Algorithms for programmers
ideas and source code

This document is work in progress: read the "important remarks" near the beginning

Jörg Arndt
arndt@jjj.de

Draft version[1] of 2008-January-19

[1]The latest version and the accompanying software is online at http://www.jjj.de/fxt/.

Contents

Important remarks about this document **xi**

I Low level algorithms 1

1 Bit wizardry **3**
1.1 Trivia . . . 3
1.2 Operations on individual bits . . . 8
1.3 Operations on low bits or blocks of a word . . . 9
1.4 Isolating blocks of bits and single bits . . . 12
1.5 Computing the index of a single set bit . . . 14
1.6 Operations on high bits or blocks of a word . . . 16
1.7 Functions related to the base-2 logarithm . . . 18
1.8 Counting bits and blocks of a word . . . 19
1.9 Bit set lookup . . . 21
1.10 Avoiding branches . . . 22
1.11 Bit-wise rotation of a word . . . 26
1.12 Functions related to bit-wise rotation * . . . 27
1.13 Reversing the bits of a word . . . 30
1.14 Bit-wise zip . . . 35
1.15 Gray code and parity . . . 36
1.16 Bit sequency . . . 42
1.17 Powers of the Gray code . . . 43
1.18 Invertible transforms on words . . . 45
1.19 Moves of the Hilbert curve . . . 51
1.20 The Z-order . . . 53
1.21 Scanning for zero bytes . . . 54
1.22 2-adic inverse and square root . . . 55
1.23 Radix −2 representation . . . 56
1.24 A sparse signed binary representation . . . 59
1.25 Generating bit combinations . . . 61
1.26 Generating bit subsets of a given word . . . 63
1.27 Binary words as subsets in lexicographic order . . . 64
1.28 Minimal-change bit combinations . . . 69
1.29 Fibonacci words . . . 71
1.30 Binary words and parentheses strings * . . . 74
1.31 Error detection by hashing: the CRC . . . 77
1.32 Permutations via primitives . . . 81
1.33 CPU instructions often missed . . . 84

2 Permutations **85**
2.1 The revbin permutation . . . 85

2.2 The radix permutation . . . 89
2.3 In-place matrix transposition . . . 89
2.4 Revbin permutation and matrix transposition * . . . 91
2.5 The zip permutation . . . 93
2.6 The reversed zip permutation . . . 95
2.7 The XOR permutation . . . 96
2.8 The Gray code permutation . . . 97
2.9 The reversed Gray code permutation . . . 101
2.10 Decomposing permutations * . . . 102
2.11 General permutations and their operations . . . 104

3 Sorting and searching 115
3.1 Sorting . . . 115
3.2 Binary search . . . 117
3.3 Index sorting . . . 118
3.4 Pointer sorting . . . 120
3.5 Sorting by a supplied comparison function . . . 121
3.6 Determination of unique elements . . . 124
3.7 Unique elements with inexact types . . . 125
3.8 Determination of equivalence classes . . . 127
3.9 Determination of monotonicity and convexity * . . . 131
3.10 Heapsort . . . 134
3.11 Counting sort and radix sort . . . 134
3.12 Searching in unsorted arrays . . . 137

4 Data structures 141
4.1 Stack (LIFO) . . . 141
4.2 Ring buffer . . . 143
4.3 Queue (FIFO) . . . 144
4.4 Deque (double-ended queue) . . . 146
4.5 Heap and priority queue . . . 148
4.6 Bit-array . . . 152
4.7 Finite-state machines . . . 154
4.8 Emulation of coroutines . . . 156

II Combinatorial generation 159

5 Conventions and considerations 161
5.1 About representations and orders . . . 161
5.2 Ranking, unranking, and counting . . . 162
5.3 Characteristics of the algorithms . . . 162
5.4 Optimization techniques . . . 162
5.5 Remarks about the C++ implementations . . . 164

6 Combinations 165
6.1 Lexicographic and co-lexicographic order . . . 166
6.2 Order by prefix shifts (cool-lex) . . . 169
6.3 Minimal-change order . . . 170
6.4 The Eades-McKay strong minimal-change order . . . 172
6.5 Two-close orderings via endo/enup moves . . . 175
6.6 Recursive generation of certain orderings . . . 179

7 Compositions 183

7.1 Co-lexicographic order . . . 183
7.2 Co-lexicographic order for compositions into exactly k parts . . . 185
7.3 Compositions and combinations . . . 187
7.4 Minimal-change orders . . . 188

8 Subsets **191**
8.1 Lexicographic order . . . 191
8.2 Minimal-change order . . . 193
8.3 Ordering with De Bruijn sequences . . . 197
8.4 Shifts-order for subsets . . . 199
8.5 k-subsets where k lies in a given range . . . 200

9 Mixed radix numbers **207**
9.1 Counting order . . . 207
9.2 Gray code order . . . 210
9.3 gslex order . . . 213
9.4 endo order . . . 216
9.5 Gray code for endo order . . . 217

10 Permutations **219**
10.1 Lexicographic order . . . 219
10.2 Co-lexicographic order . . . 221
10.3 Factorial representations of permutations . . . 222
10.4 An order from reversing prefixes . . . 231
10.5 Minimal-change order (Heap's algorithm) . . . 234
10.6 Lipski's Minimal-change orders . . . 236
10.7 Strong minimal-change order (Trotter's algorithm) . . . 239
10.8 Minimal-change orders from factorial numbers . . . 244
10.9 Orders where the smallest element always moves right . . . 250
10.10 Single track orders . . . 254
10.11 Star-transposition order . . . 259
10.12 Derangement order . . . 260
10.13 Recursive algorithm for cyclic permutations . . . 263
10.14 Minimal-change order for cyclic permutations . . . 265
10.15 Permutations with special properties . . . 267

11 Subsets and permutations of a multiset **275**
11.1 Subsets of a multiset . . . 275
11.2 Permutations of a multiset . . . 276

12 Gray codes for strings with restrictions **281**
12.1 Fibonacci words . . . 282
12.2 Generalized Fibonacci words . . . 284
12.3 Digit x followed by at least x zeros . . . 287
12.4 Generalized Pell words . . . 288
12.5 Sparse signed binary words . . . 290
12.6 Strings with no two successive nonzero digits . . . 292
12.7 Strings with no two successive zeros . . . 294
12.8 Binary strings without substrings 1x1 . . . 295
12.9 Binary strings without substrings 1xy1 . . . 296

13 Parenthesis strings **299**
13.1 Co-lexicographic order . . . 299
13.2 Gray code via restricted growth strings . . . 301

13.3 The number of parenthesis strings: Catalan numbers . . . 306
13.4 Increment-i RGS and k-ary trees . . . 307

14 Integer partitions **311**
14.1 Recursive solution of a generalized problem . . . 311
14.2 Iterative algorithm . . . 313
14.3 Partitions into m parts . . . 315
14.4 The number of integer partitions . . . 316

15 Set partitions **319**
15.1 The number of set partitions: Stirling set numbers and Bell numbers . . . 320
15.2 Generation in minimal-change order . . . 321

16 A string substitution engine **331**

17 Necklaces and Lyndon words **335**
17.1 Generating all necklaces . . . 336
17.2 The number of binary necklaces . . . 343
17.3 The number of binary necklaces with fixed content . . . 344

18 Hadamard and conference matrices **347**
18.1 Hadamard matrices via LFSR . . . 347
18.2 Hadamard matrices via conference matrices . . . 349
18.3 Conference matrices via finite fields . . . 351

19 Searching paths in directed graphs **355**
19.1 Representation of digraphs . . . 356
19.2 Searching full paths . . . 357
19.3 Conditional search . . . 362
19.4 Edge sorting and lucky paths . . . 366
19.5 Gray codes for Lyndon words . . . 367

III Fast orthogonal transforms 373

20 The Fourier transform **375**
20.1 The discrete Fourier transform . . . 375
20.2 Summary of definitions of Fourier transforms * . . . 376
20.3 Radix-2 FFT algorithms . . . 378
20.4 Saving trigonometric computations . . . 383
20.5 Higher radix FFT algorithms . . . 385
20.6 Split-radix Fourier transforms . . . 392
20.7 Symmetries of the Fourier transform . . . 395
20.8 Inverse FFT for free . . . 397
20.9 Real valued Fourier transforms . . . 398
20.10 Multidimensional Fourier transforms . . . 404
20.11 The matrix Fourier algorithm (MFA) . . . 406

21 Algorithms for fast convolution **409**
21.1 Convolution . . . 409
21.2 Correlation . . . 414
21.3 Weighted Fourier transforms and convolutions . . . 417
21.4 Convolution using the MFA . . . 419
21.5 The z-transform (ZT) . . . 422
21.6 Prime length FFTs . . . 426

22 The Walsh transform and its relatives **429**
22.1 The Walsh transform: Walsh-Kronecker basis . . . 429
22.2 Eigenvectors of the Walsh transform * . . . 432
22.3 The Kronecker product . . . 433
22.4 A variant of the Walsh transform * . . . 436
22.5 Higher radix Walsh transforms . . . 437
22.6 Localized Walsh transforms . . . 440
22.7 Dyadic (XOR) convolution . . . 445
22.8 The Walsh transform: Walsh-Paley basis . . . 447
22.9 Sequency ordered Walsh transforms . . . 448
22.10 Slant transform . . . 454
22.11 Arithmetic transform . . . 455
22.12 Reed-Muller transform . . . 459
22.13 The OR-convolution, and the AND-convolution . . . 462

23 The Haar transform **465**
23.1 The 'standard' Haar transform . . . 465
23.2 In-place Haar transform . . . 467
23.3 Non-normalized Haar transforms . . . 469
23.4 Transposed Haar transforms . . . 471
23.5 The reversed Haar transform . . . 473
23.6 Relations between Walsh and Haar transforms . . . 475
23.7 Nonstandard splitting schemes * . . . 478

24 The Hartley transform **483**
24.1 Definition and symmetries . . . 483
24.2 Radix-2 FHT algorithms . . . 484
24.3 Complex FT by HT . . . 489
24.4 Complex FT by complex HT and vice versa . . . 490
24.5 Real FT by HT and vice versa . . . 491
24.6 Higher radix FHT algorithms . . . 492
24.7 Convolution via FHT . . . 493
24.8 Negacyclic convolution via FHT . . . 496
24.9 Localized FHT algorithms . . . 497
24.10 Two-dimensional FHTs . . . 499
24.11 Discrete cosine transform (DCT) by HT . . . 500
24.12 Discrete sine transform (DST) by DCT . . . 501
24.13 Automatic generation of transform code . . . 502
24.14 Eigenvectors of the Fourier and Hartley transform * . . . 504

25 Number theoretic transforms (NTTs) **507**
25.1 Prime moduli for NTTs . . . 507
25.2 Implementation of NTTs . . . 509
25.3 Convolution with NTTs . . . 514

26 Fast wavelet transforms **515**
26.1 Wavelet filters . . . 515
26.2 Implementation . . . 517
26.3 Moment conditions . . . 518

IV Fast arithmetic **521**

27 Fast multiplication and exponentiation **523**

27.1 Asymptotics of algorithms . . . 523
27.2 Splitting schemes for multiplication . . . 524
27.3 Fast multiplication via FFT . . . 532
27.4 Radix/precision considerations with FFT multiplication . . . 534
27.5 The sum-of-digits test . . . 536
27.6 Binary exponentiation . . . 537

28 Root extraction **541**
28.1 Division, square root and cube root . . . 541
28.2 Root extraction for rationals . . . 544
28.3 Divisionless iterations for the inverse a-th root . . . 546
28.4 Initial approximations for iterations . . . 549
28.5 Some applications of the matrix square root . . . 550
28.6 Goldschmidt's algorithm . . . 555
28.7 Products for the a-th root . . . 558
28.8 Divisionless iterations for polynomial roots . . . 560

29 Iterations for the inversion of a function **563**
29.1 Iterations and their rate of convergence . . . 563
29.2 Schröder's formula . . . 564
29.3 Householder's formula . . . 566
29.4 Dealing with multiple roots . . . 568
29.5 More iterations . . . 569
29.6 Improvements by the delta squared process . . . 571

30 The arithmetic-geometric mean (AGM) **573**
30.1 The AGM . . . 573
30.2 The elliptic functions K and E . . . 575
30.3 AGM-type algorithms for hypergeometric functions . . . 578
30.4 Computation of π . . . 582
30.5 Arctangent relations for π * . . . 590

31 Logarithm and exponential function **597**
31.1 Logarithm . . . 597
31.2 Exponential function . . . 603
31.3 Logarithm and exponential function of power series . . . 606
31.4 Simultaneous computation of logarithms of small primes . . . 608

32 Numerical evaluation of power series **611**
32.1 The binary splitting algorithm for rational series . . . 611
32.2 Rectangular schemes for evaluation of power series . . . 617
32.3 The magic sumalt algorithm for alternating series . . . 621

33 Computing the elementary functions with limited resources **625**
33.1 Shift-and-add algorithms for $\log_b(x)$ and b^x . . . 625
33.2 CORDIC algorithms . . . 630

34 Recurrences and Chebyshev polynomials **635**
34.1 Recurrences . . . 635
34.2 Chebyshev polynomials . . . 645

35 Cyclotomic polynomials, Hypergeometric functions, and continued fractions **655**
35.1 Cylotomic polynomials, Möbius inversion, Lambert series . . . 655
35.2 Hypergeometric functions . . . 663
35.3 Continued fractions . . . 680

36 Synthetic Iterations * 691
36.1 A variation of the iteration for the inverse . . . 691
36.2 An iteration related to the Thue constant . . . 695
36.3 An iteration related to the Golay-Rudin-Shapiro sequence . . . 696
36.4 Iterations related to the ruler function . . . 698
36.5 An iteration related to the period-doubling sequence . . . 700
36.6 An iteration from substitution rules with sign . . . 704
36.7 Iterations related to the sum of digits . . . 704
36.8 Iterations related to the binary Gray code . . . 706
36.9 A function that encodes the Hilbert curve . . . 712
36.10 Sparse variants of the inverse . . . 715
36.11 An iteration related to the Fibonacci numbers . . . 718
36.12 Iterations related to the Pell numbers . . . 722

V Algorithms for finite fields 729

37 Modular arithmetic and some number theory 731
37.1 Implementation of the arithmetic operations . . . 731
37.2 Modular reduction with structured primes . . . 735
37.3 The sieve of Eratosthenes . . . 738
37.4 The order of an element . . . 739
37.5 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = \mathrm{GF}(p)$ . . . 741
37.6 Composite modulus: the ring $\mathbb{Z}/m\mathbb{Z}$ . . . 741
37.7 The Chinese Remainder Theorem (CRT) . . . 747
37.8 Quadratic residues . . . 749
37.9 Computation of a square root modulo m . . . 751
37.10 The Rabin-Miller test for compositeness . . . 753
37.11 Proving primality . . . 759
37.12 Complex moduli: $\mathrm{GF}(p^2)$ . . . 770
37.13 Solving the Pell equation . . . 778
37.14 Multigrades * . . . 781
37.15 Properties of the convergents of $\sqrt{2}$ * . . . 782
37.16 Multiplication of hypercomplex numbers * . . . 787

38 Binary polynomials 793
38.1 The basic arithmetical operations . . . 793
38.2 Multiplication for polynomials of high degree . . . 799
38.3 Modular arithmetic with binary polynomials . . . 805
38.4 Irreducible and primitive polynomials . . . 808
38.5 The number of irreducible and primitive polynomials . . . 823
38.6 Generating irreducible polynomials from necklaces . . . 824
38.7 Irreducible and cyclotomic polynomials * . . . 826
38.8 Factorization of binary polynomials . . . 827

39 Shift registers 833
39.1 Linear feedback shift registers (LFSR) . . . 833
39.2 Galois and Fibonacci setup . . . 836
39.3 Generating all revbin pairs . . . 837
39.4 The number of m-sequences and De Bruijn sequences . . . 838
39.5 Auto correlation of m-sequences . . . 839
39.6 Feedback carry shift register (FCSR) . . . 840
39.7 Linear hybrid cellular automata (LHCA) . . . 842
39.8 Additive linear hybrid cellular automata . . . 847

40 Binary finite fields: GF(2^n) **851**
40.1 Arithmetic and basic properties . . . 851
40.2 Minimal polynomials . . . 857
40.3 Computation of the trace vector via Newton's formula . . . 859
40.4 Solving quadratic equations . . . 861
40.5 Representation by matrices * . . . 863
40.6 Representation by normal bases . . . 865
40.7 Conversion between normal and polynomial representation . . . 873
40.8 Optimal normal bases (ONB) . . . 875
40.9 Gaussian normal bases . . . 877

A Machine used for benchmarking **883**

B The pseudo language Sprache **885**

C The pari/gp language **887**

Bibliography **895**

Index **911**

Important remarks about this document

This is a draft of a book about selected algorithms. The audience in mind are programmers who are interested in the treated algorithms and actually want to create and understand working and reasonably optimized code.

The style varies somewhat which I do not consider bad per se: While some topics (as fast Fourier transforms) need a clear and explicit introduction others (like the bit wizardry chapter) seem to be best presented by basically showing the code with just a few comments.

The pseudo language Sprache is used when I see a clear advantage to do so, mainly when the corresponding C++ does not appear to be self explanatory. Larger pieces of code are presented in C++. C programmers do not need to be shocked by the '++' as only a rather minimal set of the C++ features is used. Some of the code, especially in part 3 (Arithmetical algorithms), is given in the pari/gp language as the use of other languages would likely bury the idea in technicalities.

A printable version of this book will always stay online for free download. The referenced sources are online as part of FXT (fast transforms and low level routines [19]) and hfloat (high precision floating point algorithms [20]).

The reader is welcome to criticize and suggest improvements. Please name the draft version (date) with your feedback! **This version is of 2008-January-19.** Note that you can copy and paste from the PDF and DVI versions. Thanks go to those[1] who helped to improve this document so far!

In case you want to cite this document, please avoid referencing individual chapters or sections as their numbers (and titles) may change.

Enjoy reading!

Legal matters

[1] in particular Igal Aharonovich, Nathan Bullock, Dominique Delande, Torsten Finke, Sean Furlong, Almaz Gaifullin, Alexander Glyzov, Andreas Grünbacher, Christoph Haenel, Tony Hardie-Bick, Laszlo Hars, Jeff Hurchalla, Gideon Klimer, Dirk Lattermann, Gál László, Avery Lee, Brent Lehman, Marc Lehmann, Paul C. Leopardi, John Lien, Mirko Liss, Johannes Middeke, Doug Moore, Andrew Morris, David Nalepa, Mirosław Osys, Christoph Pacher, Scott Paine, Yves Paradis, Edith Parzefall, André Piotrowski, David García Quintas, Tony Reix, Johan Rönnblom, Thomas Schraitle, Clive Scott, Michael Somos, Ralf Stephan, Michal Staruch, Mikko Tommila, Michael Roby Wetherfield, Vinnie Winkler, Jim White, John Youngquist, Rui Zhang, and Paul Zimmermann.

"Why make things difficult, when it is possible to make them cryptic and totally illogical, with just a little bit more effort?"

– Aksel Peter Jørgensen

Part I

Low level algorithms

Chapter 1

Bit wizardry

We present low-level functions that operate on the bits of a binary word. It is often not obvious what these are good for and I do not attempt much to motivate why particular functions are presented. However, *if* you happen to have an application for a given routine you will love that it is there: the program using it may run significantly faster.

The C-type `unsigned long` is abbreviated as `ulong` as defined in [FXT: fxttypes.h]. It is assumed that `BITS_PER_LONG` reflects the size of an `unsigned long`. It is defined in [FXT: bits/bitsperlong.h] and (on sane architectures) equals the machine word size. That is, it equals 32 on 32-bit architectures and 64 on 64-bit machines. Further, the quantity `BYTES_PER_LONG` shall reflect the number of bytes in a machine word, that is, it equals `BITS_PER_LONG` divided by eight. For some functions it is assumed that `long` and `ulong` have the same number of bits.

The examples of assembler code are for the x86 and the AMD64 architecture. They should be simple enough to be understandable for readers who know assembler for any CPU.

1.1 Trivia

1.1.1 Little endian versus big endian

The order in which the bytes of an integer are stored in memory can start with the least significant byte (*little endian* machine) or with the most significant byte (*big endian* machine). The hexadecimal number `0x0D0C0B0A` will be stored in the following manner when memory addresses grow from left to right:

```
adr:   z   z+1   z+2  z+3
mem:   0D   0C   0B   0A   // big endian
mem:   0A   0B   0C   0D   // little endian
```

The difference is only visible when you cast pointers. Let `V` be the 32-bit integer with the value above. Then the result of `char c = *(char *)(&V);` will be `0x0A` (value modulo 256) on a little endian machine but `0x0D` (value divided by 2^{24}) on a big endian machine. Portable code that uses casts may need two versions, one for each endianness. Though friends of the big endian way sometimes refer to little endian as 'wrong endian', the wanted result of the shown pointer cast is much more often the modulo operation.

1.1.2 Size of pointer is size of long

On sane architectures a pointer fits into a type `long` integer. When programming for a 32-bit architecture (where the size of `int` and `long` coincide) casting pointers to integers (and back) will work. The same

code *will* fail on 64-bit machines. If you have to cast pointers to an integer type, cast them to `long`.

1.1.3 Shifts and division

With two's complement arithmetic (that is: on likely every computer you'll ever touch) division and multiplication by powers of two is right and left shift, respectively. This is true for unsigned types and for multiplication (left shift) with signed types. Division with signed types rounds toward zero, as one would expect, but right shift is a division (by a power of two) that rounds to minus infinity:

```
int a = -1;
int c = a >> 1;   // c == -1
int d = a / 2;    // d ==  0
```

The compiler still uses a shift instruction for the division, but with a 'fix' for negative values:

```
  9:test.cc   @ int foo(int a)
 10:test.cc   @ {
285 0003 8B442410              movl 16(%esp),%eax  // move argument to %eax
 11:test.cc   @     int s = a >> 1;
289 0007 89C1                  movl %eax,%ecx
290 0009 D1F9                  sarl $1,%ecx
 12:test.cc   @     int d = a / 2;
293 000b 89C2                  movl %eax,%edx
294 000d C1EA1F                shrl $31,%edx  // fix: %edx=(%edx<0?1:0)
295 0010 01D0                  addl %edx,%eax // fix: add one if a<0
296 0012 D1F8                  sarl $1,%eax
```

For unsigned types the shift would suffice. One more reason to use unsigned types whenever possible.

The assembler listing was generated from C code via the following commands:

```
# create assembler code:
c++ -S -fverbose-asm -g -O2  test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s  > test.lst
```

There are two types of *right* shifts: a so-called logical and an arithmetical shift. The logical version (`shrl` in the above fragment) always fills the higher bits with zeros, corresponding to division[1] of unsigned types. The arithmetical shift (`sarl` in the above fragment) fills in ones or zeros, according to the most significant bit of the original word.

Computing remainders modulo a power of two with unsigned types is equivalent to a bit-and using a mask:

```
ulong a = b % 32;  // == b & (32-1)
```

All of the above is done by the compiler's optimization wherever possible.

Division by (compile time) constants can be replaced by multiplications and shift. The magic machinery inside the compiler does it for you. A division by the constant 10 is compiled to:

```
  5:test.cc   @ ulong foo(ulong a)
  6:test.cc   @ {
  7:test.cc   @     ulong b = a / 10;
290 0000 8B442404              movl 4(%esp),%eax
291 0004 F7250000              mull .LC33   // value == 0xcccccccd
292 000a 89D0                  movl %edx,%eax
293 000c C1E803                shrl $3,%eax
```

Thereby it is sometimes reasonable to have separate code branches with explicit special values. Similarly, for modulo computations with a constant modulus (using modulus 10,000):

```
  8:test.cc   @ ulong foo(ulong a)
  9:test.cc   @ {
 53 0000 8B4C2404              movl 4(%esp),%ecx
 10:test.cc   @     ulong b = a % 10000;
 57 0004 89C8                  movl %ecx,%eax
 58 0006 F7250000              mull .LC0   // value == 0xd1b71759
 59 000c 89D0                  movl %edx,%eax
 60 000e C1E80D                shrl $13,%eax
 61 0011 69C01027              imull $10000,%eax,%eax
```

[1] So you can think of it as 'unsigned arithmetical' shift.

```
 62 0017 29C1                subl %eax,%ecx
 63 0019 89C8                movl %ecx,%eax
```

Algorithms to replace divisions by a constant by multiplications and shifts are given in [125].

1.1.4 A pitfall (two's complement)

```
 c=................  -c=................   c=     0  -c=     0  <--=
 c=...............1  -c=1111111111111111   c=     1  -c=    -1
 c=..............1.  -c=111111111111111.   c=     2  -c=    -2
 c=..............11  -c=111111111111111.   c=     3  -c=    -3
 c=.............1..  -c=11111111111111..   c=     4  -c=    -4
 c=.............1.1  -c=11111111111111.1   c=     5  -c=    -5
 c=.............11.  -c=11111111111111.1.  c=     6  -c=    -6
[--snip--]
 c=.111111111111.1  -c=1............11   c= 32765  -c=-32765
 c=.111111111111.1  -c=1............1.   c= 32766  -c=-32766
 c=.111111111111.1  -c=1............1.   c= 32767  -c=-32767
 c=1...............  -c=1...............   c=-32768  -c=-32768  <--=
 c=1..............1  -c=.111111111111111   c=-32767  -c= 32767
 c=1.............1.  -c=.111111111111111.  c=-32766  -c= 32766
 c=1.............11  -c=.111111111111.1.   c=-32765  -c= 32765
 c=1............1..  -c=.11111111111.11..  c=-32764  -c= 32764
 c=1............1.1  -c=.11111111111.11.1  c=-32763  -c= 32763
 c=1............11.  -c=.11111111111.1.1.  c=-32762  -c= 32762
[--snip--]
 c=111111111111..1  -c=............111   c=    -7  -c=     7
 c=111111111111.1.  -c=............11.   c=    -6  -c=     6
 c=111111111111.11  -c=............1.1   c=    -5  -c=     5
 c=111111111111..  -c=............1...   c=    -4  -c=     4
 c=111111111111.1  -c=............11.   c=    -3  -c=     3
 c=111111111111..  -c=............1...   c=    -2  -c=     2
 c=111111111111111  -c=...............1  c=    -1  -c=     1
```

Figure 1.1-A: With two's complement there is one nonzero value that is its own negative.

In two's complement zero is not the only number that is equal to its negative. With a data type of n bits the value with just the highest bit set (the most negative value) also has this property. Figure 1.1-A (the output of [FXT: bits/gotcha-demo.cc]) shows the situation for words of sixteen bits. This is the reason why innocent looking code like

```
if ( x<0 )  x = -x;
// assume x positive here (WRONG!)
```

can simply fail.

1.1.5 Another pitfall (shifts in the C-language)

A shift by more than `BITS_PER_LONG`−1 is undefined by the C-standard. Therefore the following function can fail if `k` is zero:

```
inline ulong first_comb(ulong k)
// Return the first combination of (i.e. smallest word with) k bits,
// i.e.  00..001111..1 (k low bits set)
{
    ulong t = ~0UL >> ( BITS_PER_LONG - k );
    return t;
}
```

Compilers usually emit just a shift instruction which on certain CPUs does *not* give zero if the shift is equal to or greater than `BITS_PER_LONG`. This is why the line

```
if ( k==0 )  t = 0;  // shift with BITS_PER_LONG is undefined
```

has to be inserted just before the `return` statement.

1.1.6 Shortcuts

To test whether at least one of `a` and `b` equals zero use `if ( !(a && b) )`. This works for signed and unsigned integers. Checking whether both are zero can be done using `if ( (a|b)==0 )`. This

obviously generalizes for several variables as `if ( (a|b|c|..|z)==0 )`. Test whether exactly one of two variables is zero using `if ( (!a) ^ (!b) )` .

1.1.7 Toggling between values

In order to toggle an integer `x` between two values `a` and `b` use:

```
precalculate:  t  = a ^ b;
toggle:        x ^= t;   // a <--> b
```

the equivalent trick for floating point types is

```
precalculate:  t = a + b;
toggle:        x = t - x;
```

Here an overflow could occur with `a` and `b` in the legal range (but both close to overflow). This should, however, not happen with sane programs.

1.1.8 Next or previous even or odd value

Compute the next or previous even or odd value via [FXT: bits/evenodd.h]:

```
static inline ulong next_even(ulong x)  { return x+2-(x&1); }
static inline ulong prev_even(ulong x)  { return x-2+(x&1); }

static inline ulong next_odd(ulong x)  { return x+1+(x&1); }
static inline ulong prev_odd(ulong x)  { return x-1-(x&1); }
```

The following functions return the unmodified argument if it has the required property, else the nearest such value:

```
static inline ulong next0_even(ulong x)  { return x+(x&1); }
static inline ulong prev0_even(ulong x)  { return x-(x&1); }

static inline ulong next0_odd(ulong x)  { return x+1-(x&1); }
static inline ulong prev0_odd(ulong x)  { return x-1+(x&1); }
```

1.1.9 Testing whether bit-subset

The following function tests whether a word u, as a bit-set, is a subset of another word e [FXT: bits/bitsubsetq.h]:

```
inline bool is_subset(ulong u, ulong e)
// Return whether u is a bit-subset of e.
{
    return  ( (u & e)==u );
}
```

Should u contain any bits not set in e then these bits are deleted in the AND-operation and the test for equality will fail.

1.1.10 Integer versus float multiplication

The floating point multiplier gives the highest bits of the product. Integer multiplication gives the result modulo 2^b where b is the number of bits of the integer type used. As an example we square the number 1010101 using a 32-bit integer type and floating point types with 24-bit and 53-bit mantissa:

```
  a = 111111111
a*a = 12345678987654321 // true result

a*a = 1653732529            // result with 32-bit integer multiplication
(a*a)%(2**32) = 1653732529  // ... which is modulo (2**bits_per_int)

a*a = 1.2345679481405440e+16 // result with float multiplication (24 bit mantissa)
a*a = 1.2345678987654320e+16 // result with float multiplication (53 bit mantissa)
```

1.1.11 Double precision float to signed integer conversion

Conversion of double precision floats that have a 53-bit mantissa to signed integers via [13, p.52-53]

```
#define DOUBLE2INT(i, d)  { double t = ((d) + 6755399441055744.0); i = *((int *)(&t)); }
double x;
int i = 123;
DOUBLE2INT(i, x);
```

can be a faster alternative to

```
double x = 123.0;
int i;
i = x;
```

The constant used is $6755399441055744 = 2^{52} + 2^{51}$. The method is machine dependent as it relies on the binary representation of the floating point mantissa. Here it is assumed that, firstly, the floating point number has a 53-bit mantissa with the most significant bit (that is always one with normalized numbers) omitted, and secondly, the address of the number points to the mantissa.

1.1.12 Optimization considerations

Never ever think that some code is the 'fastest possible', there always another trick that can still improve performance. Many factors can have an influence on performance like number of CPU registers or cost of branches. Code that performs well on one machine might perform badly on another. The old trick to swap variables without using a temporary

```
        //   a=0, b=0    a=0, b=1    a=1, b=0    a=1, b=1
a ^= b; //     0    0      1    1      1    0      0    1
b ^= a; //     0    0      1    0      1    1      0    1
a ^= b; //     0    0      1    0      0    1      1    1

equivalent to:
tmp = a;  a = b;  b = tmp;
```

is pretty much out of fashion today. However in some specific context (like extreme register pressure) it may be the way to go.

The only way to find out which version of a function is faster is to actually do profiling (timing). The performance does depend on the stream of instructions before the machine code (we assume that all of these low-level functions get inlined). Studying the generated CPU instructions does help to understand what is going on but can never replace profiling.

The code surrounding a specific function can have a massive impact on performance. That is, benchmarks for just the isolated routine can only give a rough indication. Profile your application and also test whether the second best (when isolated) routine is the fastest.

Never just replace the unoptimized version of some code fragment when introducing a streamlined one. Keep the original in the source. In case something nasty happens (think of low level software failures when porting to a different platform) you'll be very thankful for the chance to temporarily use the slow but correct version.

Study the optimization recommendations for your CPU (like [13] for the AMD64). It doesn't hurt to see the corresponding documentation for other architectures.

Proper documentation is an absolute must for optimized code, just assume that nobody will be able to read and understand it from the supplied source alone. The experience of not being able to understand code you have written some time ago helps a lot in this matter.

More techniques for optimization are given in section 5.4 on page 162.

1.2 Operations on individual bits

1.2.1 Testing, setting, and deleting bits

The following functions should be self explanatory. Following the spirit of the C language there is no check whether the indices used are out of bounds. That is, if any index is greater or equal `BITS_PER_LONG`, the result is undefined [FXT: bits/bittest.h]:

```
inline ulong test_bit(ulong a, ulong i)
// Return zero if bit[i] is zero,
//  else return one-bit word with bit[i] set.
{
    return  (a & (1UL << i));
}
```

The following version returns either zero or one:

```
static inline bool test_bit01(ulong a, ulong i)
// Return whether bit[i] is set.
{
    return ( 0 != test_bit(a, i) );
}

inline ulong set_bit(ulong a, ulong i)
// Return a with bit[i] set.
{
    return  (a | (1UL << i));
}

inline ulong delete_bit(ulong a, ulong i)
// Return a with bit[i] cleared.
{
    return  (a & ~(1UL << i));
}

inline ulong change_bit(ulong a, ulong i)
// Return a with bit[i] changed.
{
    return  (a ^ (1UL << i));
}
```

1.2.2 Copying a bit

In order to copy a bit from one position to another we generate a one exactly if the bits at the two positions differ. Then an XOR changes the target bit if needed [FXT: bits/bitcopy.h]:

```
inline ulong copy_bit(ulong a, ulong isrc, ulong idst)
// Copy bit at [isrc] to position [idst].
// Return the modified word.
{
    ulong x = ((a>>isrc) ^ (a>>idst)) & 1;  // one if bits differ
    a ^= (x<<idst); // change if bits differ
}
```

The situation is more tricky if the bit positions are given as (one bit) masks:

```
inline ulong mask_copy_bit(ulong a, ulong msrc, ulong mdst)
// Copy bit according at src-mask (msrc)
//  to the bit according to the dest-mask (mdst).
// Both msrc and mdst must have exactly one bit set.
// Return the modified word.
{
    ulong x = mdst;
    if ( msrc & a )  x = 0;  // zero if source bit set
    x ^= mdst;  // ==mdst if source bit set, else zero
    a &= ~mdst;  // clear dest bit
    a |= x;
    return a;
}
```

The compiler generates branch-free code as the conditional assignment is compiled to a `cmov` (conditional move) assembler instruction. If one or both masks have several bits set the routine will set all bits of `mdst` if any of the bits in `msrc` is one else clear all bits of `mdst`.

1.2.3 Swapping two bits

A function to swap two bits of a word [FXT: bits/bitswap.h]:

```
static inline ulong bit_swap(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// k1==k2 is allowed (a is unchanged then)
{
    ulong x = ((a>>k1) ^ (a>>k2)) & 1;  // one if bits differ
    a ^= (x<<k2); // change if bits differ
    a ^= (x<<k1); // change if bits differ
    return  a;
}
```

When it is known that the bits do have different values the following routine can be used:

```
static inline ulong bit_swap_01(ulong a, ulong k1, ulong k2)
// Return a with bits at positions [k1] and [k2] swapped.
// Bits must have different values (!)
// (i.e. one is zero, the other one)
// k1==k2 is allowed (a is unchanged then)
{
    return  a ^ ( (1UL<<k1) ^ (1UL<<k2) );
}
```

1.3 Operations on low bits or blocks of a word

The underlying idea of functions that operate on the lowest set bit is that addition and subtraction of 1 always changes a burst of bits at the lower end of the word. The following functions are given in [FXT: bits/bitlow.h].

Isolation of the lowest set bit is achieved via

```
static inline ulong lowest_bit(ulong x)
// Return word where only the lowest set bit in x is set.
// Return 0 if no bit is set.
{
    return  x & -x;  // use: -x == ~x + 1
}
```

The lowest zero (unset bit) of some word `x` is then trivially isolated using the equivalent of `lowest_bit( ~x )`:

```
static inline ulong lowest_zero(ulong x)
// Return word where only the lowest unset bit in x is set.
// Return 0 if all bits are set.
{
    x = ~x;
    return  x & -x;
}
```

Alternatively, one can use either of

```
    return  (x ^ (x+1)) & ~x;
    return  ((x ^ (x+1)) >> 1 ) + 1;
```

The sequence of returned values for $x = 0, 1, \ldots$ is the binary ruler function, the highest power of two that divides $x + 1$:

```
   x:  ==   x        lowest_zero(x)
   0:  == ........    .......1
   1:  == .......1    ......1.
   2:  == ......1.    .....1..
   3:  == ......11    .....1..
   4:  == .....1..    ....1...
   5:  == .....1.1    ......1.
   6:  == .....11.    .......1
   7:  == .....111    ....1...
   8:  == ....1...    .......1
   9:  == ....1..1    ......1.
  10:  == ....1.1.    .......1
```

Clearing the lowest set bit in a word can be achieved via

```
static inline ulong delete_lowest_bit(ulong x)
// Return word were the lowest bit set in x is cleared.
// Return 0 for input == 0.
{
    return  x & (x-1);
}
```

while setting the lowest unset bit is done by

```
static inline ulong set_lowest_zero(ulong x)
// Return word were the lowest unset bit in x is set.
// Return ~0 for input == ~0.
{
    return  x | (x+1);
}
```

Isolate the burst of low bits/zeros as follows:

```
static inline ulong low_bits(ulong x)
// Return word where all the (low end) ones are set.
// Example:  01011011 --> 00000011
// Return 0 if lowest bit is zero:
//        10110110 --> 0
{
    if ( ~0UL==x )  return ~0UL;
    return (((x+1)^x) >> 1);
}
```

and

```
static inline ulong low_zeros(ulong x)
// Return word where all the (low end) zeros are set.
// Example:  01011000 --> 00000111
// Return 0 if all bits are set.
{
    if ( 0==x )  return ~0UL;
    return (((x-1)^x) >> 1);
}
```

Isolation of the lowest block of ones (which may have zeros to the right of it) can be achieved via:

```
static inline ulong lowest_block(ulong x)
// Isolate lowest block of ones.
// e.g.:
// x   = *****011100
// l   = 00000000100
// y   = *****100000
// x^y = 00000111100
// ret = 00000011100
{
    ulong l = x & -x;  // lowest bit
    ulong y = x + l;
    x ^= y;
    return  x & (x>>1);
}
```

Extracting the *index* (position) of the lowest bit is easy when the corresponding assembler instruction is used [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_bsf(ulong x)
// Bit Scan Forward
{
    asm ("bsfq %0, %0" : "=r" (x) : "0" (x));
```

```
    return x;
}
```

Without the assembler instruction an algorithm that uses proportional $\log_2$(BITS_PER_LONG) can be used, so the resulting function can be implemented as[2] (64-bit version)

```
static inline ulong lowest_bit_idx(ulong x)
// Return index of lowest bit set.
// Examples:
//    ***1 --> 0
//    **10 --> 1
//    *100 --> 2
// Return 0 (also) if no bit is set.
{
    ulong r = 0;
    x &= -x;  // isolate lowest bit
    if ( x & 0xffffffff00000000UL )  r += 32;
    if ( x & 0xffff0000ffff0000UL )  r += 16;
    if ( x & 0xff00ff00ff00ff00UL )  r += 8;
    if ( x & 0xf0f0f0f0f0f0f0f0UL )  r += 4;
    if ( x & 0xccccccccccccccccUL )  r += 2;
    if ( x & 0xaaaaaaaaaaaaaaaaUL )  r += 1;
#endif // BITS_USE_ASM
    return r;
}
```

The function returns zero for two inputs, one and zero. If one needs a special value for the input zero, add a statement like

```
    if ( 1>=x )  return  x-1; // 0 if 1, ~0 if 0
```

as first line of the function.

Occasionally one wants to set a rising or falling edge at the position of the lowest bit:

```
static inline ulong lowest_bit_01edge(ulong x)
// Return word where a all bits from  (including) the
//   lowest set bit to bit 0 are set.
// Return 0 if no bit is set.
{
    if ( 0==x )  return 0;
    return  x^(x-1);
}

static inline ulong lowest_bit_10edge(ulong x)
// Return word where a all bits from (including) the
//   lowest set bit to most significant bit are set.
// Return 0 if no bit is set.
{
    if ( 0==x )  return 0;
    x ^= (x-1);
    // here  x == lowest_bit_01edge(x);
    return  ~(x>>1);
}
```

The following function returns the parity of the lowest bit in a binary word

```
static inline ulong lowest_bit_idx_parity(ulong x)
{
    x &= -x;  // isolate lowest bit
    return  (x & 0xaaaaaaaaaaaaaaaaUL);
}
```

The sequence of values for $x = 0, 1, 2, \ldots$ is

```
0010000101010001000100010101000101010001010100010001000101010100010...
```

This is the complement of the *period-doubling sequence*, entry A035263 of [214]. See section 36.5.1 on page 701 for the connection to the towers of Hanoi puzzle.

[2] thanks go to Nathan Bullock for communicating this improved version.

1.4 Isolating blocks of bits and single bits

We give functions for the creation or extraction of bit-blocks, single bits and related tasks.

1.4.1 Creating bit-blocks

The following functions are given in [FXT: bits/bitblock.h].

```
static inline ulong bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  x << p;
}
```

A version with indices wrapping around is

```
static inline ulong cyclic_bit_block(ulong p, ulong n)
// Return word with length-n bit block starting at bit p set.
// The result is possibly wrapped around the word boundary.
// Both p and n are effectively taken modulo BITS_PER_LONG.
{
    ulong x = (1UL<<n) - 1;
    return  (x<<p) | (x>>(BITS_PER_LONG-p));
}
```

1.4.2 Isolating single bits or zeros

The following functions are given in [FXT: bits/bitmisc.h].

```
static inline ulong single_bits(ulong x)
// Return word were only the single bits from x are set.
{
    return  x & ~( (x<<1) | (x>>1) );
}
```

```
static inline ulong single_zeros(ulong x)
// Return word were only the single zeros from x are set.
{
    return  single_bits( ~x );
}
```

```
static inline ulong single_values(ulong x)
// Return word were only the single bits and the
//   single zeros from x are set.
{
    return  (x ^ (x<<1)) & (x ^ (x>>1));
}
```

1.4.3 Isolating single bits or zeros at the word boundary

```
static inline ulong border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie next to a zero.
{
    return  x & ~( (x<<1) & (x>>1) );
}
```

```
static inline ulong border_values(ulong x)
// Return word were those bits/zeros from x are set
//   that lie next to a zero/bit.
{
    ulong g = x ^ (x>>1);
    g |= (g<<1);
    return   g | (x & 1);
}
```

1.4.4 Isolating bits at zero-one transitions

```
static inline ulong high_border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie right to (i.e. in the next lower bin of) a zero.
{
    return  x & ( x ^ (x>>1) );
}

static inline ulong low_border_bits(ulong x)
// Return word were only those bits from x are set
//   that lie left to (i.e. in the next higher bin of) a zero.
{
    return  x & ( x ^ (x<<1) );
}
```

1.4.5 Isolating bits or zeros at block boundaries

```
static inline ulong block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at the border of a block of at least 2 bits.
{
    return  x & ( (x<<1) ^ (x>>1) );
}

static inline ulong low_block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at left of a border of a block of at least 2 bits.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_bits()
    return  t & (x>>1);
}

static inline ulong high_block_border_bits(ulong x)
// Return word were only those bits from x are set
//   that are at right of a border of a block of at least 2 bits.
{
    ulong t = x & ( (x<<1) ^ (x>>1) );  // block_border_bits()
    return  t & (x<<1);
}

static inline ulong block_bits(ulong x)
// Return word were only those bits from x are set
//   that are part of a block of at least 2 bits.
{
    return  x & ( (x<<1) | (x>>1) );
}
```

1.4.6 Isolating the interior of bit blocks

```
static inline ulong block_values(ulong x)
// Return word were only those bits/values are set
//   that do not lie next to an opposite value.
{
    return  ~single_values(x);
}

static inline ulong interior_bits(ulong x)
// Return word were only those bits from x are set
//   that do not have a zero to their left or right.
{
    return  x & ( (x<<1) & (x>>1) );
}

static inline ulong interior_values(ulong x)
// Return word were only those bits/zeros from x are set
//   that do have a zero/bit to their left or right.
{
    return  ~border_values(x);
}
```

1.5 Computing the index of a single set bit

In the function `lowest_bit_idx()` we first isolated the lowest bit of a word `x` by first setting `x&=-x`. At this point, `x` contains just one set bit (or `x==0`). The following lines in the routine implement an algorithm that computes the index of the single bit set. This section gives some alternative techniques to compute the index of a single-bit word.

1.5.1 Cohen's trick

A nice trick is presented in [83]: for N-bit words find a number m so that all powers of two are different modulo m. That is, the order of two modulo m must be greater or equal to N. We use a table `mt[]` of size m that contains the powers of two: `mt[(2**j) mod m] = j` for $j > 0$ and a special value for $j = 0$. To look up the index of a one-bit-word `x` it is reduced modulo m and `mt[x]` is returned.

```
modulus m=11
   k =  0  1  2  3  4  5  6  7
mt[k]=  0  0  1  8  2  4  9  7

Lowest bit == 0:       x= .......1 =   1    x % m=  1 ==> lookup = 0
Lowest bit == 1:       x= ......1. =   2    x % m=  2 ==> lookup = 1
Lowest bit == 2:       x= .....1.. =   4    x % m=  4 ==> lookup = 2
Lowest bit == 3:       x= ....1... =   8    x % m=  8 ==> lookup = 3
Lowest bit == 4:       x= ...1.... =  16    x % m=  5 ==> lookup = 4
Lowest bit == 5:       x= ..1..... =  32    x % m= 10 ==> lookup = 5
Lowest bit == 6:       x= .1...... =  64    x % m=  9 ==> lookup = 6
Lowest bit == 7:       x= 1....... = 128    x % m=  7 ==> lookup = 7
```

Figure 1.5-A: Determination of the position of a single bit with 8-bit words.

We demonstrate the method for $N = 8$ where $m = 11$ is the smallest number with the required property. The setup routine for the table is

```
const ulong m = 11; // the modulus
ulong mt[m+1];
static void mt_setup()
{
    mt[0] = 0;  // special value for the zero word
    ulong t = 1;
    for (ulong i=1; i<m; ++i)
    {
        mt[t] = i-1;
        t *= 2;
        if ( t>=m )  t -= m;  // modular reduction
    }
}
```

The entry in `mt[0]` will be accessed when the input is the zero word. One can use a special value that the algorithm will return for input zero. Here we simply used zero in order to always have the same return value as with `lowest_bit_idx()`. The computation of the index can then be achieved by

```
inline ulong m_lowest_bit_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x %= m;   // power of two modulo m
    return  mt[x];  // lookup
}
```

The code is given in the demo [FXT: bits/modular-lookup-demo.cc], the output with $N = 8$ (edited for size) is shown in figure 1.5-A. The following moduli $m(N)$ can be used for N-bit words:

```
N:  4   8  16   32   64   128   256    512    1024
m:  5  11  19   37   67   131   269    523    1061
```

The modulus $m(N)$ is the smallest prime greater than N such that 2 is a primitive root modulo $m(N)$:

```
    for (n=2, 10,  N=2^n;   \\ N bits per word
        forprime (z=N, N+9999,
```

```
            if ( znorder(Mod(2,z))>=N, print(N,": ",z);break() )
        )
    )
```

1.5.2 Using De Bruijn sequences

The following method (given in [166]) is even more elegant, it uses binary De Bruijn sequences of size N. A binary De Bruijn sequence of length 2^N contains all binary words of length N (see section 39.1 on page 833). These are the sequences for 32 and 64 bit, as binary words:

```
#if BITS_PER_LONG == 32
const ulong db = 0x4653ADFUL;
// == 00000100011001010011101011011111
const ulong s = 32-5;
#else
const ulong db = 0x218A392CD3D5DBFUL;
// == 0000001000011000101000111001001011001101001111010101110110111111
const ulong s = 64-6;
#endif
```

```
  db=...1.111  (De Bruijn sequence)
      k  =  0  1  2  3  4  5  6  7
  dbt[k] =  0  1  2  4  7  3  6  5

  Lowest bit == 0:    x = .......1   db * x = ...1.111   shifted = ........ == 0 ==> lookup = 0
  Lowest bit == 1:    x = ......1.   db * x = ..1.111.   shifted = .......1 == 1 ==> lookup = 1
  Lowest bit == 2:    x = .....1..   db * x = .1.111..   shifted = ......1. == 2 ==> lookup = 2
  Lowest bit == 3:    x = ....1...   db * x = 1.111...   shifted = .....1.1 == 5 ==> lookup = 3
  Lowest bit == 4:    x = ...1....   db * x = .111....   shifted = ......11 == 3 ==> lookup = 4
  Lowest bit == 5:    x = ..1.....   db * x = 111.....   shifted = .....111 == 7 ==> lookup = 5
  Lowest bit == 6:    x = .1......   db * x = 11......   shifted = .....11. == 6 ==> lookup = 6
  Lowest bit == 7:    x = 1.......   db * x = 1.......   shifted = .....1.. == 4 ==> lookup = 7
```

Figure 1.5-B: Computing the position of the single set bit in 8-bit words with a De Bruijn sequence.

Let w_i be the i-th sub-word from the left (high end). We create a table so that the entry with index w_i points to i:

```
ulong dbt[BITS_PER_LONG];
static void dbt_setup()
{
    for (ulong i=0; i<BITS_PER_LONG; ++i)  dbt[ (db<<i)>>s ] = i;
}
```

The computation of the index involves a multiplication and a table lookup:

```
inline ulong db_lowest_bit_idx(ulong x)
{
    x &= -x;  // isolate lowest bit
    x *= db;  // multiplication by a power of two is a shift
    x >>= s;  // use log_2(BITS_PER_LONG) highest bits
    return  dbt[x];  // lookup
}
```

The used sequences must start with at least $\log_2(N) - 1$ zeros because in the line `x *= db` the word `x` is shifted (not rotated). The code is given in the demo [FXT: bits/debruijn-lookup-demo.cc], the output with $N = 8$ (edited for size, dots denote zeros) is shown in figure 1.5-B.

1.5.3 Using floating point numbers

Floating point numbers are normalized so that the highest bit in the mantissa is one. Therefore if one converts an integer into a float then the position of the *highest* set bit can be read off the exponent. By isolating the lowest bit before that operation its index can be found by the same trick. However, the conversion between integers and floats is usually slow. Further, the technique is highly machine dependent.

1.6 Operations on high bits or blocks of a word

For the functions operating on the highest bit there is not a way as trivial as with the equivalent task with the lower end of the word. With a bit-reverse CPU-instruction available life would be significantly easier. However, almost no CPU seems to have it. The following functions are given in [FXT: bits/bithigh.h].

Isolation of the highest set bit is achieved via the bit-scan instruction when it is available [FXT: bits/bitasm-i386.h]:

```
static inline ulong asm_bsr(ulong x)
// Bit Scan Reverse
{
    asm ("bsrl %0, %0" : "=r" (x) : "0" (x));
    return x;
}
```

else one may use

```
static inline ulong highest_bit_01edge(ulong x)
// Return word where a all bits from (including) the
//   highest set bit to bit 0 are set.
// Return 0 if no bit is set.
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  x;
}
```

so the resulting code is

```
static inline ulong highest_bit(ulong x)
// Return word where only the highest bit in x is set.
// Return 0 if no bit is set.
{
#if defined  BITS_USE_ASM
    if ( 0==x )  return 0;
    x = asm_bsr(x);
    return  1UL<<x;
#else
    x = highest_bit_01edge(x);
    return  x ^ (x>>1);
#endif // BITS_USE_ASM
}
```

Trivially, the highest zero can be isolated using `highest_bit(~x)`. Thereby

```
static inline ulong set_highest_zero(ulong x)
// Return word were the highest unset bit in x is set.
// Return ~0 for input == ~0.
{
    return  x | highest_bit( ~x );
}
```

Finding the index of the highest set bit uses the equivalent algorithm as with the lowest set bit:

```
static inline ulong highest_bit_idx(ulong x)
// Return index of highest bit set.
// Return 0 if no bit is set.
{
#if defined  BITS_USE_ASM
    return  asm_bsr(x);
#else // BITS_USE_ASM

    if ( 0==x )  return  0;

    ulong r = 0;
#if  BITS_PER_LONG >= 64
    if ( x & (~0UL<<32) )  { x >>= 32;  r += 32; }
#endif
    if ( x & 0xffff0000 )  { x >>= 16;  r += 16; }
    if ( x & 0x0000ff00 )  { x >>=  8;  r +=  8; }
```

```
        ------------------------------------------------------------
        ................1111....1111.111 = 0xf0f7  == word
        ................1............... = highest_bit
        ................1111111111111111 = highest_bit_01edge
        1111111111111111................ = highest_bit_10edge
                                      15 = highest_bit_idx
        ................................ = low_zeros
        ............................111 = low_bits
        ................................1 = lowest_bit
        ................................1 = lowest_bit_01edge
        11111111111111111111111111111111 = lowest_bit_10edge
                                       0 = lowest_bit_idx
        ............................111 = lowest_block
        ................1111....1111.11. = delete_lowest_bit
        ............................1... = lowest_zero
        ................1111....11111111 = set_lowest_zero
        ................................ = high_bits
        1111111111111111................ = high_zeros
        1............................... = highest_zero
        1...............1111....1111.111 = set_highest_zero
        ------------------------------------------------------------
        11111111111111111....1111....1... = 0xffff0f08  == word
        1............................... = highest_bit
        11111111111111111111111111111111 = highest_bit_01edge
        1............................... = highest_bit_10edge
                                     31 = highest_bit_idx
        ............................111 = low_zeros
        ................................ = low_bits
        ............................1... = lowest_bit
        ............................1111 = lowest_bit_01edge
        11111111111111111111111111111... = lowest_bit_10edge
                                      3 = lowest_bit_idx
        ............................1... = lowest_block
        11111111111111111....1111........ = delete_lowest_bit
        ................................1 = lowest_zero
        11111111111111111....1111....1..1 = set_lowest_zero
        1111111111111111................ = high_bits
        ................................ = high_zeros
        ................1............... = highest_zero
        1111111111111111...1111....1... = set_highest_zero
        ------------------------------------------------------------
```

Figure 1.6-A: Operations on the highest and lowest bits (and blocks) of a binary word for two different 32-bit input words. Dots denote zeros.

```
    if ( x & 0x000000f0 )  { x >>=  4;  r +=  4; }
    if ( x & 0x0000000c )  { x >>=  2;  r +=  2; }
    if ( x & 0x00000002 )  {            r +=  1; }
    return r;
#endif // BITS_USE_ASM
}
```

Isolation of the high zeros goes like

```
static inline ulong high_zeros(ulong x)
// Return word where all the (high end) zeros are set.
// e.g.:  00011001 --> 11100000
// Returns 0 if highest bit is set:
//        11011001 --> 00000000
{
    x |= x>>1;
    x |= x>>2;
    x |= x>>4;
    x |= x>>8;
    x |= x>>16;
#if  BITS_PER_LONG >= 64
    x |= x>>32;
#endif
    return  ~x;
}
```

The high bits can be isolated using arithmetical right shift

```
static inline ulong high_bits(ulong x)
// Return word where all the (high end) ones are set.
// e.g.  11001011 --> 11000000
// Returns 0 if highest bit is zero:
```

```
//       01110110 --> 00000000
{
    long y = (long)x;
    y &= y>>1;
    y &= y>>2;
    y &= y>>4;
    y &= y>>8;
    y &= y>>16;
#if  BITS_PER_LONG >= 64
    y &= y>>32;
#endif
    return  (ulong)y;
}
```

In case arithmetical shifts are more expensive than unsigned shifts, instead use

```
static inline ulong high_bits(ulong x)
{
    return  high_zeros( ~x );
}
```

A demonstration of selected functions operating on the highest or lowest bit (or block) of binary words is given in [FXT: bits/bithilo-demo.cc]. A part of the output is shown in figure 1.6-A.

1.7 Functions related to the base-2 logarithm

The following functions are given in [FXT: bits/bit2pow.h].

The function `ld()` that shall return $\lfloor \log_2(x) \rfloor$ can be implemented using the obvious algorithm:

```
static inline ulong ld(ulong x)
// Return k so that 2^k <= x < 2^(k+1)
// If x==0 then 0 is returned (!)
{
    ulong k = 0;
    while ( x>>=1 )  { ++k; }
    return k;
}
```

And then, `ld()` is the same as `highest_bit_idx()`, so one can use

```
static inline ulong ld(ulong x)
{
    return  highest_bit_idx(x);
}
```

The bit-wise algorithm can be faster if the average result is known to be small.

The function `one_bit_q()` can be used to determine whether its argument is a power of two:

```
static inline bool one_bit_q(ulong x)
// Return whether x \in {1,2,4,8,16,...}
{
    ulong m = x-1;
    return  (((x^m)>>1) == m);
}
```

The following function does the same except that it returns `true` also for the zero argument:

```
static inline bool is_pow_of_2(ulong x)
// Return whether x == 0(!) or x == 2**k
{
    return  !(x & (x-1));
}
```

Occasionally useful in FFT based computations (where the length of the available FFTs is often restricted to powers of two) are

```
static inline ulong next_pow_of_2(ulong x)
// Return x if x=2**k
// else return 2**ceil(log_2(x))
// Exception: returns 0 for x==0
{
```

```
    if ( is_pow_of_2(x) )  return x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
#if BITS_PER_LONG == 64
    x |= x >> 32;
#endif
    return  x + 1;
}
```

and

```
static inline ulong next_exp_of_2(ulong x)
// Return k if x=2**k else return k+1.
// Exception: returns 1 for x==0
{
    ulong ldx = ld(x);
    ulong n = 1UL<<ldx;  // n<=x
    if ( n==x )  return  ldx;
    else         return  ldx+1;
}
```

The following version of `next_pow_of_2()` can be faster than the one given above if assembler inlines are used:

```
static inline ulong next_pow_of_2(ulong x)
{
    if ( is_pow_of_2(x) )  return x;
    ulong n = 1UL<<ld(x);  // n<x
    return  n<<1;
}
```

1.8 Counting bits and blocks of a word

If your CPU does not have a bit count instruction (sometimes called 'population count') then you might use an algorithm given in [FXT: bits/bitcount.h]. The following functions need proportional to $\log_2$(`BITS_PER_LONG`) operations:

```
static inline ulong bit_count(ulong x)
// Return number of bits set
{
    x = (0x55555555UL & x) + (0x55555555UL & (x>> 1));  // 0-2 in 2 bits
    x = (0x33333333UL & x) + (0x33333333UL & (x>> 2));  // 0-4 in 4 bits
    x = (0x0f0f0f0fUL & x) + (0x0f0f0f0fUL & (x>> 4));  // 0-8 in 8 bits
    x = (0x00ff00ffUL & x) + (0x00ff00ffUL & (x>> 8));  // 0-16 in 16 bits
    x = (0x0000ffffUL & x) + (0x0000ffffUL & (x>>16));  // 0-31 in 32 bits
    return x;
}
```

The underlying idea is to do a search via bit masks. The code can be improved to either

```
    x  = ((x>>1) & 0x55555555UL) + (x & 0x55555555UL);  // 0-2 in 2 bits
    x  = ((x>>2) & 0x33333333UL) + (x & 0x33333333UL);  // 0-4 in 4 bits
    x  = ((x>>4) + x) & 0x0f0f0f0fUL;                   // 0-8 in 4 bits
    x +=  x>> 8;                                        // 0-16 in 8 bits
    x +=  x>>16;                                        // 0-32 in 8 bits
    return  x & 0xff;
```

or (taken from [11])

```
    x -=  (x>>1) & 0x55555555UL;
    x  = ((x>>2) & 0x33333333UL) + (x & 0x33333333UL);
    x  = ((x>>4) + x) & 0x0f0f0f0fUL;
    x *= 0x01010101UL;
    return  x>>24;
```

Which of the latter two versions is faster mainly depends on the speed of integer multiplication.

For 64-bit words the masks have to be adapted and one more step must be added (example corresponding to the second variant above):

```
    x = ((x>>1) & 0x5555555555555555UL) + (x & 0x5555555555555555)UL;  // 0-2 in 2 bits
    x = ((x>>2) & 0x3333333333333333UL) + (x & 0x3333333333333333)UL;  // 0-4 in 4 bits
    x = ((x>>4) + x) & 0x0f0f0f0f0f0f0f0fUL;                           // 0-8 in 4 bits
    x +=  x>> 8;                                                       // 0-16 in 8 bits
    x +=  x>>16;                                                       // 0-32 in 8 bits
    x +=  x>>32;                                                       // 0-64 in 8 bits
    return  x & 0xff;
```

The following code (communicated by Johan Rönnblom [priv.comm.]) may be advantageous for 32-bit systems where loading constants is somewhat expensive:

```
inline uint CountBits32(uint a)
{
    uint mask = 011111111111UL;
    a = (a - ((a&~mask)>>1)) - ((a>>2)&mask);
    a += a>>3;
    a = (a & 070707) + ((a>>18) & 070707);
    a *= 010101;
    return  ((a>>12) & 0x3f);
}
```

The following algorithm avoids all branches and may be useful when branches are expensive:

```
static inline ulong bit_count_01(ulong x)
// Return number of bits in a word
// for words of the special form 00...0001...11
{
    ulong ct = 0;
    ulong a;
#if  BITS_PER_LONG == 64
    a = (x & (1UL<<32)) >> (32-5);  // test bit 32
    x >>= a;  ct += a;
#endif
    a = (x & (1UL<<16)) >> (16-4);  // test bit 16
    x >>= a;  ct += a;

    a = (x & (1UL<<8)) >> (8-3);  // test bit 8
    x >>= a;  ct += a;

    a = (x & (1UL<<4)) >> (4-2);  // test bit 4
    x >>= a;  ct += a;

    a = (x & (1UL<<2)) >> (2-1);  // test bit 2
    x >>= a;  ct += a;

    a = (x & (1UL<<1)) >> (1-0);  // test bit 1
    x >>= a;  ct += a;

    ct += x & 1; // test bit 0

    return ct;
}
```

1.8.1 Sparse counting

When the (average input) word is known to have only a few bits set the following sparse count variant can be advantageous:

```
static inline ulong bit_count_sparse(ulong x)
// Return number of bits set.
{
    if ( 0==x )  return 0;
    ulong n = 0;
    do  { ++n; }  while ( x &= (x-1) );
    return  n;
}
```

The loop will execute once for each set bit.

1.8.2 Counting blocks

The number of bit-blocks in a binary word can computed by the following function:

```
static inline ulong bit_block_count(ulong x)
// Return number of bit blocks.
```

```
// E.g.:
// ..1..11111...111.  -> 3
// ...1..11111...111  -> 3
// ......1.....1.1..  -> 3
// .........111.1111  -> 2
{
    return  bit_count( (x^(x>>1)) ) / 2 + (x & 1);
}
```

Similarly, the number of blocks with two or more bits can be counted via:

```
static inline ulong bit_block_ge2_count(ulong x)
// Return number of bit blocks with at least 2 bits.
// E.g.:
// ..1..11111...111.  -> 2
// ...1..11111...111  -> 2
// ......1.....1.1..  -> 0
// .........111.1111  -> 2
{
    return  bit_block_count( x & ( (x<<1) & (x>>1) ) );
}
```

1.8.3 GCC builtins *

Newer versions of the C compiler of the GNU Compiler Collection (GCC [112], starting with version 3.4) offer a function `__builtin_popcountl(ulong)` that counts the bits of an unsigned long integer.

We list a few such functions, taken from [113]:

```
int __builtin_ffs (unsigned int x)
    Returns one plus the index of the least significant 1-bit of x,
    or if x is zero, returns zero.

int __builtin_clz (unsigned int x)
    Returns the number of leading 0-bits in x, starting at the
    most significant bit position.  If x is 0, the result is undefined.

int __builtin_ctz (unsigned int x)
    Returns the number of trailing 0-bits in x, starting at the
    least significant bit position.  If x is 0, the result is undefined.

int __builtin_popcount (unsigned int x)
    Returns the number of 1-bits in x.

int __builtin_parity (unsigned int x)
    Returns the parity of x, i.e. the number of 1-bits in x modulo 2.
```

The names of corresponding versions for arguments of type unsigned long are obtained by adding '`l`' (ell) to the names.

1.9 Bit set lookup

There is a nice trick to determine whether a given number is contained in a given subset of the set $\{0, 1, 2, \ldots, \texttt{BITS\_PER\_LONG}-1\}$. As an example, in order to determine whether `x` is a prime less than 32, one can use the function

```
ulong m = (1UL<<2) | (1UL<<3) | (1UL<<5) | ... | (1UL<<31);  // precomputed
static inline ulong is_tiny_prime(ulong x)
{
    return  m & (1UL << x);
}
```

The same idea applied to lookup tiny factors [FXT: bits/tinyfactors.h]:

```
static inline bool is_tiny_factor(ulong x, ulong d)
// For x,d < BITS_PER_LONG (!)
// return whether d divides x  (1 and x included as divisors)
// no need to check whether d==0
//
{
    return  ( 0 != ( (tiny_factors_tab[x]>>d) & 1 ) );
```

```
}
```

The function uses the precomputed array [FXT: bits/tinyfactors.cc]:

```
extern const ulong tiny_factors_tab[] =
{
                   0x0UL,  // x = 0:           ( bits: ........)
                   0x2UL,  // x = 1:  1        ( bits: ......1.)
                   0x6UL,  // x = 2:  1 2      ( bits: .....11.)
                   0xaUL,  // x = 3:  1 3      ( bits: ....1.1.)
                  0x16UL,  // x = 4:  1 2 4    ( bits: ...1.11.)
                  0x22UL,  // x = 5:  1 5      ( bits: ..1...1.)
                  0x4eUL,  // x = 6:  1 2 3 6  ( bits: .1..111.)
                  0x82UL,  // x = 7:  1 7      ( bits: 1.....1.)
                 0x116UL,  // x = 8:  1 2 4 8
                 0x20aUL,  // x = 9:  1 3 9
[--snip--]
             0x20000002UL,  // x = 29:  1 29
             0x4000846eUL,  // x = 30:  1 2 3 5 6 10 15 30
             0x80000002UL,  // x = 31:  1 31
#if  ( BITS_PER_LONG > 32 )
            0x100010116UL,  // x = 32:  1 2 4 8 16 32
            0x20000080aUL,  // x = 33:  1 3 11 33
[--snip--]
    0x2000000000000002UL,  // x = 61:  1 61
    0x4000000080000006UL,  // x = 62:  1 2 31 62
    0x800000000020028aUL   // x = 63:  1 3 7 9 21 63
#endif // ( BITS_PER_LONG > 32 )
};
```

Bit arrays of arbitrary size are discussed in section 4.6 on page 152.

1.10 Avoiding branches

Branches are expensive operations with many CPUs, especially if the CPU pipeline is very long. The function in this section avoid branches, they are given in [FXT: bits/branchless.h].

The following function returns $\max(0, x)$. That is, zero is returned for negative input, else the unmodified input:

```
static inline long max0(long x)
{
    return  x & ~(x >> (BITS_PER_LONG-1));
}
```

There is no restriction on input range. The trick used is that with negative x the arithmetic shift will give a word of all ones which is then negated and the AND-operation deletes all bits. Similarly:

```
static inline long min0(long x)
// Return min(0, x), i.e. return zero for positive input
{
    return  x & (x >> (BITS_PER_LONG-1));
}
```

Computation of the average $(x + y)/2$ of two arguments x and y. The function gives the correct value even if $(x + y)$ does not fit into a machine word:

```
static inline ulong average(ulong x, ulong y)
// Return (x+y)/2
// Use the fact that x+y == ((x&y)<<1) + (x^y)
//    that is:       sum ==  carries   + sum_without_carries
{
    return  (x & y) + ((x ^ y) >> 1);
}
```

If it is known that $x \geq y$ then one can alternatively use the statement `return y+(x-y)/2`.

The following `upos_*()` functions only work for a limited range. The highest bit must not be set in order to have the highest bit emulate the carry flag. Branchless computation of the absolute difference $|a - b|$:

```
static inline ulong upos_abs_diff(ulong a, ulong b)
{
    long d1 = b - a;
    long d2 = (d1 & (d1>>(BITS_PER_LONG-1)))<<1;
    return  d1 - d2; // == (b - d) - (a + d);
}
```

Sorting of the arguments:

```
static inline void upos_sort2(ulong &a, ulong &b)
// Set {a, b} := {min(a, b), max(a,b)}
// Both a and b must not have the most significant bit set
{
    long d = b - a;
    d &= (d>>(BITS_PER_LONG-1));
    a += d;
    b -= d;
}
```

The following two functions adjust a given values when it lies outside a given range.

```
static inline long clip_range0(long x, long m)
// Code equivalent (for m>0) to:
//   if ( x<0 )  x = 0;
//   else if ( x>m )  x = m;
//   return  x;
{
    if ( (ulong)x > (ulong)m )  x = m & ~(x >> (BITS_PER_LONG-1));
    return  x;
}

static inline long clip_range(long x, long mi, long ma)
// Code equivalent to (for mi<=ma):
//   if ( x<mi )  x = mi;
//   else if ( x>ma )  x = ma;
{
    x -= mi;
    x = clip_range0(x, ma-mi);
    x += mi;
    return  x;
}
```

Johan Rönnblom gives the following versions for signed integer minimum, maximum, and absolute value, that can be advantageous for PPC (G4) CPUs:

```
#define B1  (BITS_PER_LONG-1) // bits of signed int minus one
#define MINI(x,y) (((x) &  (((int)((x)-(y)))>>B1)) + ((y) & ~(((int)((x)-(y)))>>B1)))
#define MAXI(x,y) (((x) & ~(((int)((x)-(y)))>>B1)) + ((y) &  (((int)((x)-(y)))>>B1)))
#define ABSI(x)   (((x) & ~(((int)(x))>>B1))       - ((x) &  (((int)(x))>>B1)))
```

1.10.1 Conditional swap

The following statement is compiled with a branch:

```
    if ( a<b )   { ulong t=a; a=b; b=t; } // swap if a < b
  // Here:   a in  %rcx,    b in  %rdx
  62 000e 4889C8                movq    %rcx, %rax      # X, X
  68 0011 4839D1                cmpq    %rdx, %rcx      # X, X
  69 0014 7306                  jae     .L3     #,   // the branch
  71 0016 4889D1                movq    %rdx, %rcx      # X, X
  72 0019 4889C2                movq    %rax, %rdx      # X, X
  73                    .L3:
```

As conditional assignments can be done branchless, an equivalent branchless version is:

```
    { ulong x=a^b;  if (a>=b) { x=0; }  a^=x;  b^=x; } // swap if a < b
 255 00af 4889EA                movq    %rbp, %rdx      # a, x
 257 00b2 B9000000              movl    $0, %ecx        #, tmp83
 257      00
 260 00b7 4831C2                xorq    %rax, %rdx      # b, x
 261 00ba 4839C5                cmpq    %rax, %rbp      # b, a
 262 00bd 480F43D1              cmovae  %rcx, %rdx      # x,, tmp83, x
 263 00c1 4831D5                xorq    %rdx, %rbp      # x, a
 264 00c4 4831D0                xorq    %rdx, %rax      # x, b
```

We'd like to have fewer instructions. If one tries

```
{ ulong ta=a;  if (a<b) {a=b; b=ta;} } // swap if a < b
```

the generated code is identical to the first version. Let's try [FXT: bits/cswap.h]:

```
static inline void cswap_lt(ulong &a, ulong &b)
// Branchless equivalent to:
// if ( a<b )   { ulong t=a; a=b; b=t; } // swap if a < b
{
    asm volatile("movq %0, %%r15 \n"   // t=a
                 "cmpq %0, %1 \n"      // cmp a, b
                 "cmovae %1, %0 \n"    // cond a=b
                 "cmovae %%r15, %1 \n" // cond b=t
                 ""
                 : "=r" (a), "=r" (b) // output
                 : "0" (a), "1" (b)   // input
                 : "r15" // clobber
                 );
}
```

Now the machine code looks better:

```
  // Here:   a in  %rax,    b in  %rdx
 111 0027 4989C7        movq %rax, %r15         # tmp71
 112 002a 4839C2        cmpq %rax, %rdx         # tmp71, tmp72
 113 002d 480F43C2      cmovae %rdx, %rax       # tmp72, tmp71
 114 0031 490F43D7      cmovae %r15, %rdx       # tmp72
```

Clearly, the relative speed of the three versions depends on the machine used. But it also turns out to be dependent on the surrounding code. We use bubble sort for benchmarking:

```
void bubble_sort(ulong *f, ulong n)
{
    while ( n-- > 1 )
        for (ulong k=0; k<n; ++k)  cswap_NN(f[k], f[k+1]);
}
```

Where we use the three versions of conditional swap for `cswap_NN()`. We sort an array of length 2^{15} twice with each version, once starting with an already sorted array and once with an array that is sorted in descending order. The 'plain' version wins:

```
    cswap_gt_plain(f[k], f[k+1]); //  3.58s
    cswap_gt_xor(f[k], f[k+1]);   //  6.34s
    cswap_gt(f[k], f[k+1]);       //  5.10s
```

This is due to the fact that the compiler bypasses the store when no swap happens:

```
 103 0020 488B4808  movq 8(%rax), %rcx #, tmp71
 104 0024 488B10    movq (%rax), %rdx #* f, tmp72
 105 0027 4839D1    cmpq %rdx, %rcx # tmp72, tmp71
 106 002a 7307      jae .L7 #,
 108 002c 48895008  movq %rdx, 8(%rax) # tmp72,
 109 0030 488908    movq %rcx, (%rax) # tmp71,* f
 111               .L7:
```

If we change the inner loop to

```
        for (ulong k=0; k<n; ++k)
        {
            ulong a = f[k],  b = f[k+1];
            cswap_NN(a, b);
            f[k] = a;   f[k+1] = b;
        }
```

then we obtain:

```
    cswap_gt_plain(a, b);  //  5.78s
    cswap_gt_xor(a, b);    //  6.22s
    cswap_gt(a, b);        //  4.68s
```

Our innocent looking change in the code prevented the compiler from doing its nice trick. The XOR version is (within timing precision) as slow as before. The assembler version wins because the data is already in registers. We learn that profiling is an absolute must.

1.10.2 Your compiler may be smarter than you thought

The machine code generated for

```
    x =  x & ~(x >> (BITS_PER_LONG-1));  // max0()
```

is

```
  35:   48 99                   cqto
  37:   48 83 c4 08             add    $0x8,%rsp  // stack adjustment
  3b:   48 f7 d2                not    %rdx
  3e:   48 21 d0                and    %rdx,%rax
```

The variable `x` resides in the register `rAX` both at start and end of the function. The compiler uses a special (AMD64) instruction `cqto`. Quoting [12]:

> Copies the sign bit in the rAX register to all bits of the rDX register. The effect of this instruction is to convert a signed word, doubleword, or quadword in the rAX register into a signed doubleword, quadword, or double-quadword in the rDX:rAX registers. This action helps avoid overflow problems in signed number arithmetic.

Now the equivalent

```
    x = ( x<0 ? 0 : x );  // max0()  "simple minded"
```

is compiled to:

```
  35:   ba 00 00 00 00          mov    $0x0,%edx
  3a:   48 85 c0                test   %rax,%rax
  3d:   48 0f 48 c2             cmovs  %rdx,%rax // note %edx is %rdx
```

A conditional move (`cmovs`) instruction is used here. That is, our optimized version is (on my machine) actually worse than the straightforward equivalent.

A second example is the function `clip_range()` above. It is compiled to

```
   0:   48 89 f8                mov    %rdi,%rax
   3:   48 29 f2                sub    %rsi,%rdx
   6:   31 c9                   xor    %ecx,%ecx
   8:   48 29 f0                sub    %rsi,%rax
   b:   78 0a                   js     17 <_Z2CLlll+0x17>  // the branch
   d:   48 39 d0                cmp    %rdx,%rax
  10:   48 89 d1                mov    %rdx,%rcx
  13:   48 0f 4e c8             cmovle %rax,%rcx
  17:   48 8d 04 0e             lea    (%rsi,%rcx,1),%rax
```

Now we replace the code by

```
inline long clip_range(long x, long mi, long ma)
{
    x -= mi;
    if ( x<0 )  x = 0;
//    else  // commented out to make (compiled) function really branchless
    {
        ma -= mi;
        if ( x>ma )  x = ma;
    }
    x += mi;
}
```

Then the compiler generates branchless code:

```
   0:   48 89 f8                mov    %rdi,%rax
   3:   b9 00 00 00 00          mov    $0x0,%ecx
   8:   48 29 f0                sub    %rsi,%rax
   b:   48 0f 48 c1             cmovs  %rcx,%rax
   f:   48 29 f2                sub    %rsi,%rdx
  12:   48 39 d0                cmp    %rdx,%rax
  15:   48 0f 4f c2             cmovg  %rdx,%rax
  19:   48 01 f0                add    %rsi,%rax
```

Still, with CPUs that do not have a conditional move instruction (or some branchless equivalent of it) the techniques shown in this section can be useful.

1.11 Bit-wise rotation of a word

Neither C nor C++ have a statement for bit-wise rotation of a binary word (which may be considered a missing feature). The operation can be 'emulated' via [FXT: bits/bitrotate.h]:

```
static inline ulong bit_rotate_left(ulong x, ulong r)
// Return word rotated r bits to the left
// (i.e. toward the most significant bit)
{
    return  (x<<r) | (x>>(BITS_PER_LONG-r));
}
```

As already mentioned, GCC emits exactly the CPU instruction that is *meant* here, even with non-constant argument `r`. Well done, GCC folks! Explicit use of the corresponding assembler instruction should not do any harm:

```
static inline ulong bit_rotate_right(ulong x, ulong r)
// Return word rotated r bits to the right
// (i.e. toward the least significant bit)
{
#if defined  BITS_USE_ASM    // use x86 asm code
    return asm_ror(x, r);
#else
    return  (x>>r) | (x<<(BITS_PER_LONG-r));
#endif
}
```

where we used [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_ror(ulong x, ulong r)
{
    asm ("rorq   %%cl, %0" : "=r" (x) : "0" (x), "c" (r));
    return x;
}
```

Rotations using only a part of the word length are achieved by

```
static inline ulong bit_rotate_left(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the left
// (i.e. toward the most significant bit)
// Must have  0 <= r <= ldn
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x<<r) | (x>>(ldn-r));
    x &= m;
    return  x;
}
```

and

```
static inline ulong bit_rotate_right(ulong x, ulong r, ulong ldn)
// Return ldn-bit word rotated r bits to the right
// (i.e. toward the least significant bit)
// Must have  0 <= r <= ldn
{
    ulong m = ~0UL >> ( BITS_PER_LONG - ldn );
    x &= m;
    x = (x>>r) | (x<<(ldn-r));
    x &= m;
    return  x;
}
```

Finally, the functions

```
static inline ulong bit_rotate_sgn(ulong x, long r, ulong ldn)
// Positive r --> shift away from element zero
{
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r, ldn);
    else          return  bit_rotate_right(x, (ulong)-r, ldn);
}
```

and (full-word version)

```
static inline ulong bit_rotate_sgn(ulong x, long r)
// Positive r --> shift away from element zero
{
```

```
    if ( r > 0 )  return  bit_rotate_left(x, (ulong)r);
    else          return  bit_rotate_right(x, (ulong)-r);
}
```

are sometimes convenient.

1.12 Functions related to bit-wise rotation *

We give several functions related to cyclic rotations of binary words. The following function determines whether there is a cyclic right shift of its second argument so that it matches the first argument. It is given in [FXT: bits/bitcyclic-match.h]:

```
static inline ulong bit_cyclic_match(ulong x, ulong y)
// Return  r if x==rotate_right(y, r) else return ~0UL.
// In other words: return
//   how often the right arg must be rotated right (to match the left)
// or, equivalently:
//   how often the left arg must be rotated left (to match the right)
{
    ulong r = 0;
    do
    {
        if ( x==y )  return r;
        y = bit_rotate_right(y, 1);
    }
    while ( ++r < BITS_PER_LONG );
    return ~0UL;
}
```

The functions shown work on the full length of the words, equivalents for the sub-word of the lowest `ldn` bits are given in the respective files. Just one example:

```
static inline ulong bit_cyclic_match(ulong x, ulong y, ulong ldn)
// Return  r if x==rotate_right(y, r, ldn) else return ~0UL
//  (using ldn-bit words)
{
    ulong r = 0;
    do
    {
        if ( x==y )  return r;
        y = bit_rotate_right(y, 1, ldn);
    }
    while ( ++r < ldn );
    return ~0UL;
}
```

The minimum among all cyclic shifts of a word can be computed via the following function given in [FXT: bits/bitcyclic-minmax.h]:

```
static inline ulong bit_cyclic_min(ulong x)
// Return minimum of all rotations of x
{
    ulong r = 1;
    ulong m = x;
    do
    {
        x = bit_rotate_right(x, 1);
        if ( x<m )  m = x;
    }
    while ( ++r < BITS_PER_LONG );
    return  m;
}
```

Selecting from all n-bit words those that are equal to their cyclic minimum gives the sequence of the binary length-n necklaces, see chapter 17 on page 335. For example, with 6-bit words:

word	period	word	period
`.....1`	1	`..11.1`	6
`....i1`	6	`..1111`	6
`...i11`	6	`.1.1.1`	2
`...1.1`	6	`.1.111`	6
`..1111`	6	`.11.11`	3
`..1..1`	3	`.11111`	6
`..1.11`	6	`111111`	1

The values in each right column can be computed using [FXT: bits/bitcyclic-period.h]:

```
static inline ulong bit_cyclic_period(ulong x, ulong ldn)
// Return minimal positive bit-rotation that transforms x into itself.
//  (using ldn-bit words)
// The returned value is a divisor of ldn.
{
    ulong y = bit_rotate_right(x, 1, ldn);
    return  bit_cyclic_match(x, y, ldn) + 1;
}
```

It is possible to completely avoid the rotation of partial words: let d be a divisor of the word length n. Then the rightmost $(n-1)\,d$ bits of the word obtained as `x^(x>>d)` are zero exactly if the word has period d. Thereby we can use the following function body:

```
    ulong sl = BITS_PER_LONG-ldn;
    for (ulong s=1; s<ldn; ++s)
    {
        ++sl;
        if ( 0==( (x^(x>>s)) << sl ) )  return s;
    }
    return ldn;
```

Testing for periods that are not divisors of the word length can be avoided as follows:

```
    ulong f = tiny_factors_tab[ldn];
    ulong sl = BITS_PER_LONG-ldn;
    for (ulong s=1; s<ldn; ++s)
    {
        ++sl;
        f >>= 1;
        if ( 0==(f&1) )  continue;
        if ( 0==( (x^(x>>s)) << sl ) )  return s;
    }
    return ldn;
```

The table of tiny factors used is shown in section 1.9 on page 21.

The version for `ldn==BITS_PER_LONG` can be optimized similarly:

```
static inline ulong bit_cyclic_period(ulong x)
// Return minimal positive bit-rotation that transforms x into itself.
// (same as bit_cyclic_period(x, BITS_PER_LONG) )
//
// The returned value is a divisor of the word length,
//   i.e. 1,2,4,8,...,BITS_PER_LONG.
{
    ulong r = 1;
    do
    {
        ulong y = bit_rotate_right(x, r);
        if ( x==y )  return r;
        r <<= 1;
    }
    while ( r < BITS_PER_LONG );

    return  r;  // == BITS_PER_LONG
}
```

A related function computes the cyclic distance between two words [FXT: bits/bitcyclic-dist.h]:

```
inline ulong bit_cyclic_dist(ulong a, ulong b)
// Return minimal bitcount of (t ^ b)
// where t runs through the cyclic rotations.
{
    ulong d = ~0UL;
    ulong t = a;
```

```
    do
    {
        ulong z = t ^ b;
        ulong e = bit_count( z );
        if ( e < d )  d = e;
        t = bit_rotate_right(t, 1);
    }
    while ( t!=a );
    return  d;  // not reached
}
```

The functions [FXT: bits/bitcyclic-xor.h]

```
static inline ulong bit_cyclic_rxor(ulong x)
{
    return x ^ bit_rotate_right(x, 1);
}
```

and

```
static inline ulong bit_cyclic_lxor(ulong x)
{
    return x ^ bit_rotate_left(x, 1);
}
```

return a word where the number of bits is even. In order to produce a random value with an even number of set bits one can use either variant. If the bit count shall be odd, XOR the value with one[3] afterwards.

Iterated application always ends in a cycle, two examples using 6-bit words are:

```
    .11111
    11....
    1.1...  <--= cycle start
    1111..
    1...1.
    11..11
    ..1.1.
    ..1111  <--= cycle end
    1.1...

    .111..
    ..1..1
    .11.11  <--= cycle start
    11.11.
    1.11.1  <--= cycle end
    .11.11
```

Zero is a fixed point (a period with cycle length one), a typical sequence using 8-bit words is:

```
    .1...11.1
    1...1.11
    .1..111.
    .11.1..1
    11.111.1
    .1.11..11
    1.1.1.1.
    11111111
    ........  <--= cycle start == cycle end
```

Cyclic shifts of a word produce cyclic shifts of the same cycle of words. A word and its complement produce the same result.

The inverse functions need no rotation at all, the inverse of `bit_cyclic_rxor()` is the inverse Gray code (see section 1.15 on page 36):

```
static inline ulong bit_cyclic_inv_rxor(ulong x)
// Return v so that bit_cyclic_rxor(v) == x.
{
    return  inverse_gray_code(x);
}
```

The argument `x` must have an even number of bits. If this is the case then the lowest bit of the result is zero. The complement of the returned value is also an inverse of `bit_cyclic_rxor()`.

The inverse of `bit_cyclic_lxor()` is the inverse reversed code (see section 1.15.6 on page 41):

```
static inline ulong bit_cyclic_inv_lxor(ulong x)
// Return v so that bit_cyclic_lxor(v) == x.
```

[3] Actually any value with an odd number of set bits will do for the XOR.

```
{
    return  inverse_rev_gray_code(x);
}
```

We do not need to mask out the lowest bit because for valid arguments (that have an even number of bits) the high bits of the result are zero. This function can be used to solve the quadratic equation $v^2 + v = x$ in the finite field GF(2^n) when normal bases are used, see page 867.

1.13 Reversing the bits of a word

The bits of a binary word can efficiently be reversed by a sequence of steps that reverse the order of certain blocks. For 16-bit words, we need $4 = \log_2(16)$ such steps [FXT: bits/revbin-steps-demo.cc]:

```
[ 0 1 2 3 4 5 6 7 8 9 a b c d e f ]
[ 1 0 3 2 5 4 7 6 9 8 b a d c f e ]  <--= pairs swapped
[ 3 2 1 0 7 6 5 4 b a 9 8 f e d c ]  <--= groups of 2 swapped
[ 7 6 5 4 3 2 1 0 f e d c b a 9 8 ]  <--= groups of 4 swapped
[ f e d c b a 9 8 7 6 5 4 3 2 1 0 ]  <--= groups of 8 swapped
```

1.13.1 Swapping adjacent bit blocks

We need a couple of auxiliary functions given in [FXT: bits/bitswap.h]. Pairs of adjacent bits can be swapped via

```
static inline ulong bit_swap_1(ulong x)
// Return x with neighbour bits swapped.
{
#if  BITS_PER_LONG == 32
    ulong m = 0x55555555UL;
#else
#if  BITS_PER_LONG == 64
    ulong m = 0x5555555555555555UL;
#endif
#endif
    return  ((x & m) << 1) | ((x & (~m)) >> 1);
}
```

The 64-bit branch is omitted in the following examples. Adjacent groups of 2 bits are swapped by

```
static inline ulong bit_swap_2(ulong x)
// Return x with groups of 2 bits swapped.
{
    ulong m = 0x33333333UL;
    return  ((x & m) << 2) | ((x & (~m)) >> 2);
}
```

Equivalently,

```
static inline ulong bit_swap_4(ulong x)
// Return x with groups of 4 bits swapped.
{
    ulong m = 0x0f0f0f0fUL;
    return  ((x & m) << 4) | ((x & (~m)) >> 4);
}
```

and

```
static inline ulong bit_swap_8(ulong x)
// Return x with groups of 8 bits swapped.
{
    ulong m = 0x00ff00ffUL;
    return  ((x & m) << 8) | ((x & (~m)) >> 8);
}
```

When swapping half-words (here for 32-bit architectures)

```
static inline ulong bit_swap_16(ulong x)
// Return x with groups of 16 bits swapped.
{
    ulong m = 0x0000ffffUL;
```

```
    return  ((x & m) << 16) | ((x & (m<<16)) >> 16);
}
```

GCC is clever enough to recognize that the whole operation is equivalent to a (left or right) word rotation and indeed emits just a single rotate instruction. We could also use the bit-rotate function from section 1.11 on page 26, or

```
    return (x << 16) | (x >> 16);
```

1.13.2 Bit-reversing binary words

The shown functions are taken from [FXT: bits/revbin.h]. The following is a 64-bit version of `revbin()`

```
static inline ulong revbin(ulong x)
// Return x with bitsequence reversed
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#if  BITS_PER_LONG >= 64
    x = bit_swap_32(x);
#endif
    return x;
}
```

For 32-bit machines the `bit_swap_32()` line would have to be omitted.

The steps after `bit_swap_4()` correspond to a byte-reverse operation. This operation is just one assembler instruction for many CPUs ('`bswap`'). The inline assembler with GCC for AMD64 CPUs is given in [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_bswap(ulong x)
{
    asm ("bswap %0" : "=r" (x) : "0" (x));
    return x;
}
```

We use it for byte reversion when available:

```
static inline ulong bswap(ulong x)
// Return word with reversed byte order.
{
#ifdef BITS_USE_ASM
    x = asm_bswap(x);
#else
    x = bit_swap_8(x);
    x = bit_swap_16(x);
#if  BITS_PER_LONG >= 64
    x = bit_swap_32(x);
#endif
#endif // def BITS_USE_ASM
    return x;
}
```

The function actually used for bit reversion is good for both 32 and 64 bit words:

```
static inline ulong revbin(ulong x)
{
    x = bit_swap_1(x);
    x = bit_swap_2(x);
    x = bit_swap_4(x);
    x = bswap(x);
    return x;
}
```

One can generate the masks in the process as follows:

```
static inline ulong revbin(ulong x)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
```

```
    {
        x = ( (x & m) << s ) ^ ( (x & (~m)) >> s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  x;
}
```

Note that the above function will not always beat the obvious, bit-wise algorithm:

```
static inline ulong revbin(ulong x)
{
    ulong r = 0,  ldn = BITS_PER_LONG;
    while ( ldn-- != 0 )
    {
        r <<= 1;
        r += (x&1);
        x >>= 1;
    }
    return  r;
}
```

Therefore the function

```
static inline ulong revbin(ulong x, ulong ldn)
// Return word with the ldn least significant bits
//   (i.e. bit_0 ... bit_{ldn-1})  of x reversed,
//   the other bits are set to zero.
{
    return  revbin(x) >> (BITS_PER_LONG-ldn);
}
```

should only be used when `ldn` is not too small, else replaced by the trivial algorithm.

One can also use table lookups methods so that, for example, eight bits are reversed at a time using a 256-byte table. We give the routine for full words:

```
unsigned char revbin_tab[256]; // reversed 8-bit words
ulong revbin_t(ulong x)
{
    ulong r = 0;
    for (ulong k=0; k<BYTES_PER_LONG; ++k)
    {
        r <<= 8;
        r |= revbin_tab[ x & 255 ];
        x >>= 8;
    }
    return r;
}
```

The routine can be optimized by unrolling to avoid all branches:

```
static inline ulong revbin_t(ulong x)
{
    ulong r      = revbin_tab[ x & 255 ];  x >>= 8;
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
#if BYTES_PER_LONG > 4
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
    r <<= 8;  r |= revbin_tab[ x & 255 ];  x >>= 8;
#endif
    r <<= 8;  r |= revbin_tab[ x ];
    return r;
}
```

However, reversing the first 2^{30} binary words with this routine takes (on a 64-bit machine) longer than with the routine using the `bit_swap_NN()` calls, see [FXT: bits/revbin-tab-demo.cc].

Bit-hacker's life would be easier if there was a CPU instruction for reversing a binary word.

1.13.3 Generating the bit-reversed words in order

If the bit-reversed words have to be generated in the (reversed) counting order then there is a significantly cheaper way to do the update [FXT: bits/revbin-upd.h]:

```
static inline ulong revbin_upd(ulong r, ulong h)
// Let n=2**ldn and h=n/2.
// Then, with r == revbin(x, ldn) at entry, return revbin(x+1, ldn)
// Note: routine will hang if called with r the all-ones word
{
    while ( !((r^=h)&h) )  h >>= 1;
    return  r;
}
```

Now assume we want to generate the bit-reversed words of all $N = 2^n$ words smaller than 2^n. The total number of branches with the `while`-loop can be estimated by observing that for half of the updates just one bit changes, for a quarter two bits change, three bits change for one eighth of all updates, and so on. Thereby the loop executes less than $2\,N$ times:

$$N\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\cdots+\frac{\log_2(N)}{N}\right) \;=\; N\sum_{j=1}^{\log_2(N)}\frac{j}{2^j} \;<\; 2\,N \qquad (1.13\text{-}1)$$

Observing that the updates that involve a single bit change occur at every second step we can avoid half of all branches.

For large vales of N the following method can be significantly faster if a fast routine is available for the computation of the least significant bit in a word. The underlying observation is that for a fixed word of size n there are just n different patterns of bit-changes with incrementing. We generate a lookup table of the bit-reversed patterns, `utab[]`, an array of `BITS_PER_LONG` elements:

```
inline void make_revbin_upd_tab(ulong ldn)
// Initialize lookup table used by revbin_tupd()
{
    utab[0] = 1UL<<(ldn-1);
    for (ulong k=1; k<ldn; ++k)  utab[k] = utab[k-1] | (utab[k-1]>>1);
}
```

The change patterns for $n = 5$ start as

```
    pattern  reversed pattern
     ...1     1...
     ..11     11..
     ...1     1...
     .111     111.
     ...1     1...
     ..11     11..
     ...1     1...
     1111     1111
     ...1     1...
     ..11     11..
```

The crucial observation is that the pattern with x set bits is used for the update of k to $k+1$ when the lowest zero of k is at position $x-1$:

```
                          used when the lowest
                reversed  zero of k is at index:
    utab[0]=     1....      0
    utab[1]=     11...      1
    utab[2]=     111..      2
    utab[3]=     1111.      3
    utab[4]=     11111      4
```

The update routine can now be implemented as

```
inline ulong revbin_tupd(ulong r, ulong k)
// Let r==revbin(k, ldn) then
// return revbin(k+1, ldn).
// NOTE 1: need to call make_revbin_upd_tab(ldn) before usage
//         where ldn=log_2(n)
// NOTE 2: different argument structure than revbin_upd()
{
    k = lowest_bit_idx(~k);  // lowest zero idx
    r ^= utab[k];
    return r;
}
```

The revbin-update routines are used for the revbin permutation described in section 2.1.

	30 bits	16 bits	8 bits	
Update, bit-wise	1.00	1.00	1.00	`revbin_upd()`
Update, table	0.99	1.08	1.15	`revbin_tupd()`
Full, masks	0.74	0.81	0.86	`revbin()`
Full, 8-bit table	1.77	1.94	2.06	`revbin_t()`
Full32, 8-bit table	0.83	0.90	0.96	`revbin_t_le32()`
Full16, 8-bit table	—	0.54	0.58	`revbin_t_le16()`
Full, generated masks	2.97	3.25	3.45	[page 31]
Full, bit-wise	8.76	5.77	2.50	[page 32]

Figure 1.13-A: Relative performance of the revbin-update and (full) revbin routines. The timing of the bit-wise update routine is normalized to one. Values in each column should be compared, smaller values correspond to faster routines. A column labeled "N bits" gives the timing for reversing the N least significant bits of a word.

The relative performance of the different revbin routines is shown in figure 1.13-A. As a surprise, the full-word revbin function is consistently faster than both of the update routines. This is mainly due to the fact that the machine used (see appendix A on page 883) has a byte swap instruction. As the performance of table lookups is highly machine dependent your results can be very different.

1.13.4 Alternative techniques for in-order generation

The following loop, due to Brent Lehmann [priv.comm.], also generates the bit-reversed words in succession:

```
ulong n = 32;  // a power of two
ulong p = 0, s = 0, n2 = 2*n;
do
{
    // here: s is the bit-reversed word
    p += 2;
    s ^= n - (n / (p&-p));
}
while ( p<n2 );
```

The revbin-increment is branchless but involves a division which usually is an expensive operation. With a fast bit-scan function the loop should be replaced by

```
do
{
    p += 1;
    s ^= n - (n >> (lowest_bit_idx(p)+1));
}
while ( p<n );
```

A recursive algorithm for the generation of the bit-reversed words in order is given in [FXT: bits/revbin-rec-demo.cc]:

```
ulong N;
void revbin_rec(ulong f, ulong n)
{
    // visit( f )
    for (ulong m=N>>1; m>n; m>>=1)  revbin_rec(f+m, m);
}
```

One has to call `revbin_rec(0, 0)` to generate all `N`-bit bit-reversed words.

A technique to generate all revbin pairs in a pseudo random order is given in section 39.3 on page 837.

1.14 Bit-wise zip

The bit-wise zip (bit-zip) operation moves the lower half bits to even indices and higher half bits to odd indices. For example, with 8-bit words the permutation of bits is (see section 2.5 on page 93):

```
[ a b c d A B C D ]  |--> [ a A b B c C d D ]
```

A straightforward implementation is

```
ulong bit_zip(ulong a, ulong b)
{
    ulong x = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        x |= (a & m) << s;
        ++s;
        x |= (b & m) << s;
        m <<= 1;
    }
    return  x;
}
```

Its inverse (bit-unzip) moves even indexed bits to the lower half-word and odd indexed bits to the higher half-word:

```
void bit_unzip(ulong x, ulong &a, ulong &b)
{
    a = 0;  b = 0;
    ulong m = 1, s = 0;
    for (ulong k=0; k<(BITS_PER_LONG/2); ++k)
    {
        a |= (x & m) >> s;
        ++s;
        m <<= 1;
        b |= (x & m) >> s;
        m <<= 1;
    }
}
```

The optimized versions (see [FXT: bits/bitzip.h]), using ideas similar to those in `revbin()` and `bit_count()`, are

```
static inline ulong bit_zip(ulong x)
{
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    x = butterfly_8(x);
    x = butterfly_4(x);
    x = butterfly_2(x);
    x = butterfly_1(x);
    return  x;
}
```

and

```
static inline ulong bit_unzip(ulong x)
{
    x = butterfly_1(x);
    x = butterfly_2(x);
    x = butterfly_4(x);
    x = butterfly_8(x);
#if  BITS_PER_LONG == 64
    x = butterfly_16(x);
#endif
    return  x;
}
```

Both use the `butterfly_*()`-functions which are defined in [FXT: bits/bitbutterfly.h]:

```
static inline ulong butterfly_4(ulong x)
{
#if  BITS_PER_LONG == 64
    const ulong ml = 0x0f000f000f000f00UL;
#else
    const ulong ml = 0x0f000f00UL;
#endif
```

```
    const ulong s = 4;
    const ulong mr = ml >> s;
    const ulong t = ((x & ml) >> s ) | ((x & mr) << s );
    x = (x & ~(ml | mr)) | t;
    return  x;
}
```

Laszlo Hars suggests [priv.comm.] the following routine (version for 32-bit words), which can be obtained by making the compile-time constants explicit:

```
inline uint32 bit_zip(uint32 x)
{
    x = ((x & 0x0000ff00) << 8) | ((x >> 8) & 0x0000ff00) | (x & 0xff0000ff);
    x = ((x & 0x00f000f0) << 4) | ((x >> 4) & 0x00f000f0) | (x & 0xf00ff00f);
    x = ((x & 0x0c0c0c0c) << 2) | ((x >> 2) & 0x0c0c0c0c) | (x & 0xc3c3c3c3);
    x = ((x & 0x22222222) << 1) | ((x >> 1) & 0x22222222) | (x & 0x99999999);
    return x;
}
```

Functions that zip/unzip the bits of (the lower half of) two words are

```
#define  BPLH  (BITS_PER_LONG/2)

static inline ulong bit_zip2(ulong x, ulong y)
// Two-word version:
// only the lower half of x and y are merged
{
    return  bit_zip( (y<<BPLH) + x );
}
```

and

```
static inline void bit_unzip2(ulong t, ulong &x, ulong &y)
// Two-word version:
// only the lower half of x and y are filled
{
    t = bit_unzip(t);
    y = t >> BPLH;
    x = t ^ (y<<BPLH);
}
```

1.15 Gray code and parity

The *Gray code* of a binary word can easily be computed by [FXT: bits/graycode.h]

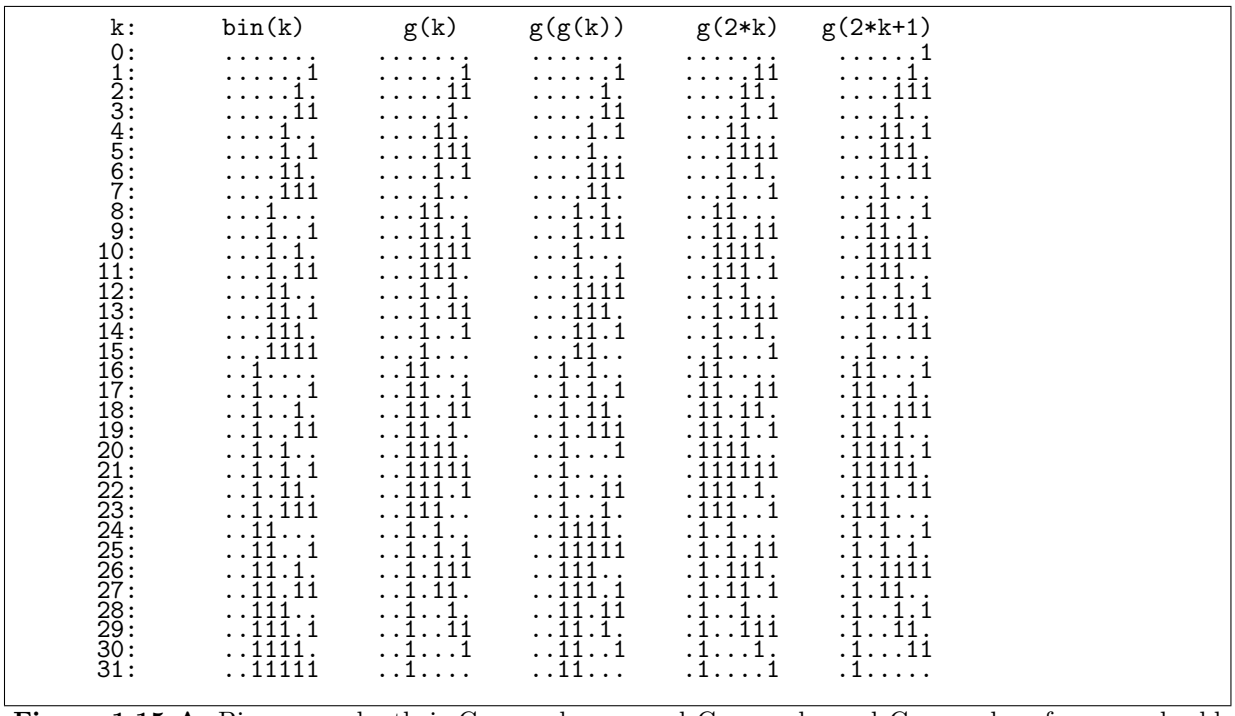
```
static inline ulong gray_code(ulong x)
{
    return  x ^ (x>>1);
}
```

Gray codes of consecutive values differ in one bit. Squared Gray codes of consecutive values differ in one or two bits. Gray codes of values that have a difference of a power of two differ in two bits. Gray codes of even/odd values have an even/odd number of bits set, respectively. This is demonstrated in [FXT: bits/gray2-demo.cc], whose output is given in figure 1.15-A.

In order to produce a random value with an even/odd number of bits set, set the lowest bit of a random number to zero/one, respectively, and take the Gray code.

Computing the inverse Gray code is slightly more expensive. Understanding the Gray code as 'bit-wise difference modulo 2' leads to the idea of computing the 'bit-wise sums modulo 2' for the inverse:

```
static inline ulong inverse_gray_code(ulong x)
{
    // VERSION 1 (integration modulo 2):
    ulong h=1, r=0;
    do
    {
        if ( x & 1 )  r^=h;
        x >>= 1;
        h = (h<<1)+1;
    }
    while ( x!=0 );
```



k:	bin(k)	g(k)	g(g(k))	g(2*k)	g(2*k+1)
0:	......	......	......	......	1
1:	1	1	1	11	....1.
2:	....1.	11	....1.	...11.	...111
3:	11	....1.	11	...1.1	...1..
4:	...1..	...11.	...1.1	..11..	..11.1
5:	...1.1	...111	...1..	..1111	..111.
6:	...11.	...1.1	...111	..1.1.	..1.11
7:	...111	...1..	...11.	..1..1	..1...
8:	..1...	..11..	..1.1.	.11...	.11..1
9:	..1..1	..11.1	..1.11	.11.11	.11.1.
10:	..1.1.	..1111	..1...	.1111.	.11111
11:	..1.11	..111.	..1..1	.111.1	.111..
12:	..11..	..1.1.	..1111	.1.1..	.1.1.1
13:	..11.1	..1.11	..111.	.1.111	.1.11.
14:	..111.	..1..1	..11.1	.1..1.	.1..11
15:	..1111	..1...	..11..	.1...1	.1....
16:	.1....	.11...	.1.1..	11....	11...1
17:	.1...1	.11..1	.1.1.1	11..11	11..1.
18:	.1..1.	.11.11	.1.11.	11.11.	11.111
19:	.1..11	.11.1.	.1.111	11.1.1	11.1..
20:	.1.1..	.1111.	.1...1	1111..	1111.1
21:	.1.1.1	.11111	.1....	111111	11111.
22:	.1.11.	.111.1	.1..11	111.1.	111.11
23:	.1.111	.111..	.1..1.	111..1	111...
24:	.11...	.1.1..	.1111.	1.1...	1.1..1
25:	.11..1	.1.1.1	.11111	1.1.11	1.1.1.
26:	.11.1.	.1.111	.111..	1.111.	1.1111
27:	.11.11	.1.11.	.111.1	1.11.1	1.11..
28:	.111..	.1..1.	.11.11	1..1..	1..1.1
29:	.111.1	.1..11	.11.1.	1..111	1..11.
30:	.1111.	.1...1	.11..1	1...1.	1...11
31:	.11111	.1....	.11...	1....1	1.....

Figure 1.15-A: Binary words, their Gray code, squared Gray code, and Gray codes of even and odd values.

```
    return r;
}
```

For n-bit words, n-fold application of the Gray code gives back the original word. Using the symbol G for the Gray code (operator) we have $G^n = \mathrm{id}$, so $G^{n-1} \circ G = \mathrm{id} = G^{-1} \circ G$. That is, applying the Gray code computation $n-1$ times gives the inverse Gray code. Thus we can simplify to

```
    // VERSION 2 (apply graycode BITS_PER_LONG-1 times):
    ulong r = BITS_PER_LONG;
    while ( --r )  x ^= x>>1;
    return x;
```

Applying the Gray code twice is identical to `x^=x>>2;`, applying it four times is `x^=x>>4;`, and the idea holds for all powers of two. This leads to the most efficient way to compute the inverse Gray code:

```
    // VERSION 3 (use: gray ** BITSPERLONG == id):
    x ^= x>>1;  // gray ** 1
    x ^= x>>2;  // gray ** 2
    x ^= x>>4;  // gray ** 4
    x ^= x>>8;  // gray ** 8
    x ^= x>>16; // gray ** 16
    // here: x = gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x>>32;  // for 64bit words
#endif
    return  x;
```

1.15.1 The parity of a binary word

The *parity* of a word is its bit-count modulo two. The inverse Gray code of a word contains at each bit position the parity of all bits of the input left from it (including itself). Thereby we use the lowest bit [FXT: bits/parity.h]:

```
static inline ulong parity(ulong x)
// return 1 if the number of set bits is even, else 0
{
```

```
    return  inverse_gray_code(x) & 1;
}
```

Be warned that the parity bit of many CPUs is the complement of the above. With the x86-architecture the parity bit only takes in account the lowest byte, therefore [FXT: bits/bitasm-i386.h]:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addl  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

The equivalent code for the AMD64 CPU is [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_parity(ulong x)
{
    x ^= (x>>32);
    x ^= (x>>16);
    x ^= (x>>8);
    asm ("addq  $0, %0  \n"
         "setnp %%al    \n"
         "movzx %%al, %0"
         : "=r" (x) : "0" (x) : "eax");
    return x;
}
```

1.15.2 Byte-wise Gray code and parity

A byte-wise Gray code can be computed using (32-bit version)

```
static inline ulong byte_gray_code(ulong x)
// Return the Gray code of bytes in parallel
{
    return  x ^ ((x & 0xfefefefe)>>1);
}
```

Its inverse is

```
static inline ulong byte_inverse_gray_code(ulong x)
// Return the inverse Gray code of bytes in parallel
{
    x ^= ((x & 0xfefefefeUL)>>1);
    x ^= ((x & 0xfcfcfcfcUL)>>2);
    x ^= ((x & 0xf0f0f0f0UL)>>4);
    return  x;
}
```

Thereby

```
static inline ulong byte_parity(ulong x)
// Return the parities of bytes in parallel
{
    return  byte_inverse_gray_code(x) & 0x01010101UL;
}
```

1.15.3 Incrementing (counting) in Gray code

Let $g(k)$ be the Gray code of a number k. We are interested in efficiently generating $g(k+1)$. Using the observation shown in figure 1.15-B we can implement a fast Gray counter if we use a spare bit to keep track of the parity of the Gray code word. The following routine does this [FXT: bits/nextgray.h]:

```
inline ulong next_gray2(ulong x)
// With input x==gray_code(2*k) the return is gray_code(2*k+2).
// Let x1 be the word x shifted right once
// and i1 its inverse Gray code.
// Let r1 be the return r shifted right once.
```

```
k:     g(k)      g(2*k)      g(k) p     diff p    set
0:   .......    .......    ...... .   ...... .    {}
1:   ......1    .....11    .....1 1   .....+ 1    {0}
2:   .....11    ....11.    ....11 .   ....+1 .    {0, 1}
3:   .....1.    ....1.1    ....1. 1   ....1- 1    {1}
4:   ....11.    ...11..    ...11. .   ...+1. .    {1, 2}
5:   ....111    ...1111    ...111 1   ...11+ 1    {0, 1, 2}
6:   ....1.1    ...1.1.    ...1.1 .   ...1-1 .    {0, 2}
7:   ....1..    ...1..1    ...1.. 1   ...1.- 1    {2}
8:   ...11..    ..11...    ..11.. .   ..+1.. .    {2, 3}
9:   ...11.1    ..11.11    ..11.1 1   ..11.+ 1    {0, 2, 3}
10:  ...1111    ..1111.    ..1111 .   ..11+1 .    {0, 1, 2, 3}
11:  ...111.    ..111.1    ..111. 1   ..111- 1    {1, 2, 3}
12:  ...1.1.    ..1.1..    ..1.1. .   ..1-1. .    {1, 3}
13:  ...1.11    ..1.111    ..1.11 1   ..1.1+ 1    {0, 1, 3}
14:  ...1..1    ..1..1.    ..1..1 .   ..1.-1 .    {0, 3}
15:  ...1...    ..1...1    ..1... 1   ..1..- 1    {3}
16:  ..11...    .11....    .11... .   .+1... .    {3, 4}
17:  ..11..1    .11..11    .11..1 1   .11..+ 1    {0, 3, 4}
```

Figure 1.15-B: The Gray code equals the Gray code of doubled value shifted to the right once. Equivalently, we can separate the lowest bit which equals the parity of the other bits. The last column shows that the changes with each increment always happen one position left of the rightmost bit.

```
// Then r1 = gray_code(i1+1).
// That is, we have a Gray code counter.
// The argument must have an even number of bits.
{
    x ^= 1;
    x ^= (lowest_bit(x) << 1);
    return x;
}
```

To obtain a Gray counter, start with `x=0`, increment with `x=next_gray2(pg)` and use the words `g=x>>1`:

```
    ulong x = 0;
    for (ulong k=0; k<n2; ++k)
    {
        ulong g = x>>1;
        x = next_gray2(x);
        // here:  g == gray_code(k);
    }
```

This is shown in [FXT: bits/bit-nextgray-demo.cc].

To start at an arbitrary (Gray code) value `g` compute

```
    x = (g<<1) ^ parity(g)
```

in order to use the statement `x=next_gray2(x)` for later increments.

If one works with a set whose elements are the set bits in the Gray code then the parity is the set size k modulo two. The increment can then be achieved as follows: if k is even then, if the first element is zero, then remove it, else prepend the element zero. If k is odd then, if the first element equals the second minus one, then remove the second element, else insert at the second position the element equal to the first element plus one. Further, the decrement is obtained by simply swapping the actions for even and odd parity.

If one works with an array that contains the elements of the set it is more convenient to actually do the described operations at the end of the array. This leads to the (loopless) algorithm for subsets in minimal-change order that is given in section 8.2 on page 193.

1.15.4 The Thue-Morse sequence

The sequence of parities of the binary words,

```
01101001100101101001011001101001100101100110100 1...
```

is called the *Thue-Morse sequence* (entry A010060 of [214]). It appears in various seemingly unrelated contexts, see [8] and section 36.1 on page 691.

The sequence can be generated with [FXT: `class thue_morse` in bits/thue-morse.h]:

```
class thue_morse
// Thue-Morse sequence
{
public:
    ulong k_;
    ulong tm_;
public:
    thue_morse(ulong k)  { init(k); }
    ~thue_morse()  { ; }
    ulong init(ulong k)
    {
        k_ = k;
        tm_ = parity(k_);
        return tm_;
    }
    ulong data()  { return tm_; }
    ulong next()
    {
        ulong x = k_ ^ (k_ + 1);
        ++k_;
        x ^= x>>1;        // highest bit that changed with increment
        x &= 0x5555555555555555UL;  // 64-bit version
        tm_ ^= ( x!=0 );  // change if highest changed bit was at even index
        return tm_;
    }
};
```

The rate of generation is about 435 million per second (5 cycles per update) [FXT: bits/thue-morse-demo.cc].

1.15.5 The Golay-Rudin-Shapiro sequence *

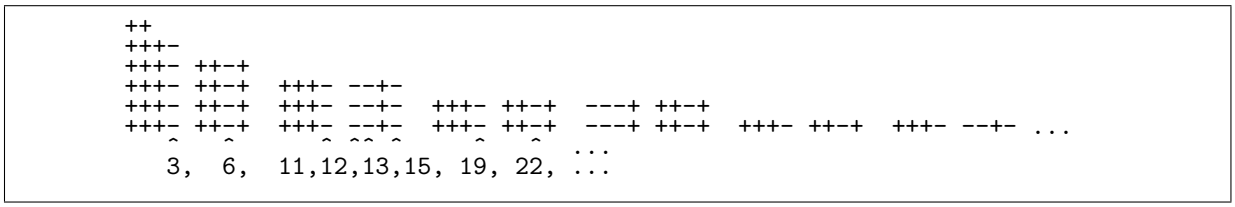


Figure 1.15-C: A construction for the GRS sequence.

The function [FXT: bits/grsnegative.h]

```
static inline ulong grs_negative_q(ulong x)
{
    return  parity( x & (x>>1) );
}
```

returns one for indices where the *Golay-Rudin-Shapiro sequence* (or *GRS sequence*) has a negative value. The function returns one for `x` in the sequence

```
3, 6, 11, 12, 13, 15, 19, 22, 24, 25, 26, 30, 35, 38, 43, 44, 45,
47, 48, 49, 50, 52, 53, 55, 59, 60, 61, 63, 67, 70, 75, 76, 77,
79, 83, 86, 88, 89, 90, 94, 96, 97, 98, 100, 101, 103, 104, 105,
106, 110, 115, 118, 120, 121, 122, 126, 131, 134, 139, 140, ...
```

This is sequence A020985 of [214], see also section 36.3 on page 696.

The sequence can be obtained by starting with a sequence of two ones and in each step appending the left half and the negated right half of the values so far, see figure 1.15-C.

The algorithm counts the bit-pairs modulo 2. Note that the sequence `[1111]` contains three bit-pairs: `[11..]`, `[.11.]` and `[..11]`. The function proves to be useful in specialized versions of the fast Fourier- and Walsh transform, see section 22.4 on page 436.

1.15.6 The reversed Gray code

```
    ------------------------------------------------------------
    111.1111....1111.............  = 0xef0f0000  == word
    1..11...1...1...1.............. = gray_code
    ..11...1...1...1.............. = rev_gray_code
    1.11.1.11111.1.1111111111111111 = inverse_gray_code
    1.1..1.1.....1.1.............. = inverse_rev_gray_code
    ------------------------------------------------------------
    ...1....1111....111111111111111  = 0x10f0ffff  == word
    ...11...1...1...1.............. = gray_code
    ..11...1...1...1.............1 = rev_gray_code
    ...11111.1.11111.1.1.1.1.1.1.1 = inverse_gray_code
    1111.....1.1.....1.1.1.1.1.1.1 = inverse_rev_gray_code
    ------------------------------------------------------------
    ......1.........................  = 0x2000000  == word
    ......11........................ = gray_code
    .....11......................... = rev_gray_code
    ......11111111111111111111111111 = inverse_gray_code
    1111111........................ = inverse_rev_gray_code
    ------------------------------------------------------------
    111111.11111111111111111111111  = 0xfdffffff  == word
    1.....11........................ = gray_code
    .....11.......................1 = rev_gray_code
    1.1.1..1.1.1.1.1.1.1.1.1.1.1.1 = inverse_gray_code
    1.1.1.11.1.1.1.1.1.1.1.1.1.1.1 = inverse_rev_gray_code
    ------------------------------------------------------------
```

Figure 1.15-D: Four examples of the Gray code, reversed Gray code and their inverses with 32-bit words.

We define the *reversed Gray code* to be the bit-reversed word of the Gray code of the bit-reversed word. That is,

```
rev_gray_code(x) := revbin(gray_code(revbin(x)))
```

It turns out that the corresponding functions are identical to the Gray code versions up to the reversed shift operations (C-language operators '`>>`' replaced by '`<<`'). Thereby, computing the reversed Gray code is as easy as [FXT: bits/revgraycode.h]:

```
static inline ulong rev_gray_code(ulong x)
{
    return  x ^ (x<<1);
}
```

Its inverse is

```
static inline ulong inverse_rev_gray_code(ulong x)
{
    // use: rev_gray ** BITSPERLONG == id:
    x ^= x<<1;  // rev_gray ** 1
    x ^= x<<2;  // rev_gray ** 2
    x ^= x<<4;  // rev_gray ** 4
    x ^= x<<8;  // rev_gray ** 8
    x ^= x<<16; // rev_gray ** 16
    // here: x = rev_gray**31(input)
    // note: the statements can be reordered at will
#if  BITS_PER_LONG >= 64
    x ^= x<<32;  // for 64bit words
#endif
    return  x;
}
```

Some examples with 32-bit words are shown in figure 1.15-D. The inverse reversed Gray code contains at each bit position the parity of all bits of the input right from it, including the bit itself. Especially, the

word parity can be found in the highest bit of the inverse reversed Gray code.

The reversed Gray code preserves the lowest set bit while the Gray code preserves the highest.

Let G^{-1} and E^{-1} be the inverse Gray- and reversed Gray code of X, respectively. Then the bit-wise sum (XOR) of G^{-1} and E^{-1} equals X if the parity of X is zero, else it equals the complement X.

We note that taking the reversed Gray code of a binary word corresponds to multiplication with the binary polynomial $x+1$, and the inverse reversed Gray code is a method for fast exact division by $x+1$, see section 38.1.6 on page 798.

1.16 Bit sequency

Functions concerned with the sequency (number of zero-one transitions) are given in [FXT: bits/bitsequency.h]. Sequency counting:

```
static inline ulong bit_sequency(ulong x)
{
    return bit_count( gray_code(x) );
}
```

The function assumes that all bits to the left of the word are zero, and all bits to right are equal to the lowest bit. For example, the sequency of the 8-bit word `[00011111]` is one. To take the lowest bit into account, add it to the sequency (then all sequencies are even).

Computation of the minimal binary word with given sequency:

```
static inline ulong first_sequency(ulong k)
// Return the first (i.e. smallest) word with sequency k,
// e.g.  00..00010101010 (seq 8)
// e.g.  00..00101010101 (seq 9)
// Must have:  0 <= k <= BITS_PER_LONG
{
    return inverse_gray_code( first_comb(k) );
}
```

A faster version is (32-bit branch only):

```
    if ( k==0 )  return 0;
    const ulong m = 0xaaaaaaaaUL;
    return  m >> (BITS_PER_LONG-k);
```

Computation of the maximal binary word with given sequency:

```
static inline ulong last_sequency(ulong k)
// Return the last (i.e. biggest) word with sequency k.
{
    return inverse_gray_code( last_comb(k) );
}
```

The functions `first_comb(k)` and `last_comb(k)` return a word with k bits set at the low and high end, respectively (see section 1.25 on page 61).

Generation of all words with a given sequency, starting with the smallest, can be achieved with a function that computes the next word with the same sequency:

```
static inline ulong next_sequency(ulong x)
// Return smallest integer with highest bit at greater or equal
// position than the highest bit of x that has the same number
// of zero-one transitions (sequency) as x.
// The value of the lowest bit is conserved.
//
// Zero is returned when there is no further sequence.
{
    x = gray_code(x);
    x = next_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

The inverse function, returning the previous word with the same sequency, is:

```
static inline ulong prev_sequency(ulong x)
{
    x = gray_code(x);
    x = prev_colex_comb(x);
    x = inverse_gray_code(x);
    return x;
}
```

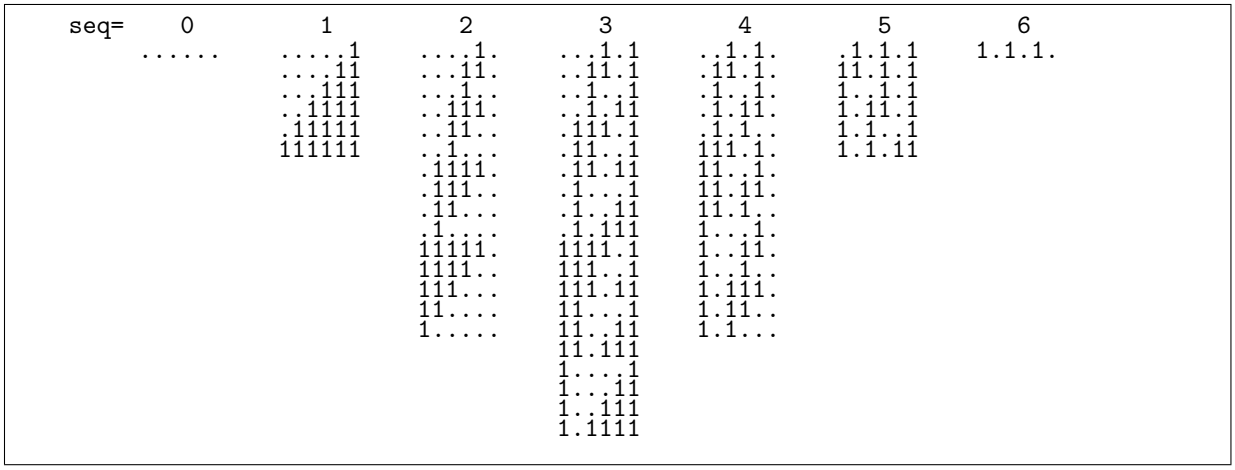


Figure 1.16-A: 6-bit words of prescribed sequency as generated by `next_sequency()`. Note that the transition at the lower end is not counted. This is consistent with sequency counting function `bit_sequency()`.

The list of all 6-bit words ordered by sequency is shown in figure 1.16-A. It was created with the program [FXT: bits/bitsequency-demo.cc].

We note that the sequency of a word can be 'complemented' as follows (32-bit version):

```
static inline ulong complement_sequency(ulong x)
// Return word whose sequency is BITS_PER_LONG - s
// where s is the sequency of x
{
    return x ^ 0xaaaaaaaaUL;
}
```

1.17 Powers of the Gray code

The Gray code is a bit-wise linear transform of a binary word. The 2^k-th power of the Gray code of x can be computed as `x ^ (x>>k)`. The e-th power can be computed as the bit-wise sum of the powers corresponding to the bits in the exponent. This motivates [FXT: bits/graypower.h]:

```
inline ulong gray_pow(ulong x, ulong e)
// Return (gray_code**e)(x)
// gray_pow(x, 1) == gray_code(x)
// gray_pow(x, BITS_PER_LONG-1) == inverse_gray_code(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 )  x ^= x >> s;  // gray ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
```

The Gray code $g = [g_0, g_1, \ldots, g_7]$ of a 8-bit binary word $x = [x_0, x_1, \ldots, x_7]$ can be expressed as a matrix multiplication over GF(2) (dots for zeros):

```
  g    =        G         x
 [g0]      [ 11...... ]  [x0]
 [g1]      [ .11..... ]  [x1]
 [g2]      [ ..11.... ]  [x2]
 [g3]  =   [ ...11... ]  [x3]
 [g4]      [ ....11.. ]  [x4]
 [g5]      [ .....11. ]  [x5]
 [g6]      [ ......11 ]  [x6]
 [g7]      [ .......1 ]  [x7]
```

The powers of the Gray code correspond to multiplication with powers of the matrix G:

```
1.......    11......    1.1.....    1111....    1...1...    11..11..    1.1.1.1.    11111111
.1......    .11.....    .1.1....    .1111...    .1...1..    .11..11.    .1.1.1.1    .1111111
..1.....    ..11....    ..1.1...    ..1111..    ..1...1.    ..11..11    ..1.1.1.    ..111111
...1....    ...11...    ...1.1..    ...1111.    ...1...1    ...11..1    ...1.1.1    ...11111
....1...    ....11..    ....1.1.    ....1111    ....1...    ....11..    ....1.1.    ....1111
.....1..    .....11.    .....1.1    .....111    .....1..    .....11.    .....1.1    .....111
......1.    ......11    ......1.    ......11    ......1.    ......11    ......1.    ......11
.......1    .......1    .......1    .......1    .......1    .......1    .......1    .......1
G**0=id     G**1=G      G**2        G**3        G**4        G**5        G**6        G**7=G**(-1)
```

The powers of the inverse Gray code for N-bit words (where N is a power of two) can be computed by the relation $G^e\, G^{N-e} = G^N = \mathrm{id}$.

```
inline ulong inverse_gray_pow(ulong x, ulong e)
// Return (inverse_gray_code**(e))(x)
//   == (gray_code**(-e))(x)
// inverse_gray_pow(x, 1) == inverse_gray_code(x)
// inverse_gray_pow(x, BITS_PER_LONG-1) == gray_code(x)
{
    return  gray_pow(x, -e);
}
```

The matrices corresponding to the powers of the reversed Gray code are:

```
1.......    1.......    1.......    1.......    1.......    1.......    1.......    1.......
.1......    11......    .1......    11......    .1......    11......    .1......    11......
..1.....    .11.....    1.1.....    111.....    ..1.....    .11.....    1.1.....    111.....
...1....    ..11....    .1.1....    1111....    ...1....    ..11....    .1.1....    1111....
....1...    ...11...    ..1.1...    .1111...    1...1...    1..11...    1.1.1...    11111...
.....1..    ....11..    ...1.1..    ..1111..    .1...1..    11..11..    .1.1.1..    111111..
......1.    .....11.    ....1.1.    ...1111.    ..1...1.    .11..11.    1.1.1.1.    1111111.
.......1    ......11    .....1.1    ....1111    ...1...1    ..11..11    .1.1.1.1    11111111
E**0=id     E**1=E      E**2        E**3        E**4        E**5        E**6        E**7=E**(-1)
```

We just have to reverse the shift operator in the functions:

```
inline ulong rev_gray_pow(ulong x, ulong e)
// Return (rev_gray_code**e)(x)
{
    e &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = 1;
    while ( e )
    {
        if ( e & 1 )  x ^= x << s;  // rev_gray ** s
        s <<= 1;
        e >>= 1;
    }
    return  x;
}
```

The inverse function is

```
inline ulong inverse_rev_gray_pow(ulong x, ulong e)
// Return (inverse_rev_gray_code**(e))(x)
{
    return  rev_gray_pow(x, -e);
}
```

1.18 Invertible transforms on words

The functions presented in this section are invertible 'transforms' on binary words. The names are chosen as 'some code', emphasizing the result of the transforms, similar to the convention used with the name 'Gray code'. The functions are given in [FXT: bits/bittransforms.h].

Consider (*blue code*)

```
inline ulong blue_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        a ^= ( (a&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

and (*yellow code*)

```
inline ulong yellow_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        a ^= ( (a&m) << s );
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

Both involve a computational work $\sim \log_2(b)$ where b is the number of bits per word (`BITS_PER_LONG`). The `blue_code` can be used as a fast implementation for the composition of a binary polynomial with $x+1$, see page 813. Note the names 'blue code' etc. are ad hoc terminology and not standard.

The output of the program [FXT: bits/bittransforms-blue-demo.cc] is shown in figure 1.18-A. The parity of $B(a)$ is equal to the lowest bit of a. Up to the $a = 47$ the bit-count varies by ± 1 between successive values of $B(a)$, the transition $B(47) \to B(48)$ changes the bit-count by 3. The sequence of the indices a where the bit-count changes by more than one is

```
47, 51, 59, 67, 75, 79, 175, 179, 187, 195, 203, 207, 291, 299, 339, 347, 419, 427, ...
```

The yellow code might be a good candidate for 'randomization' of binary words. The blue code maps any range $[0 \ldots 2^k - 1]$ onto itself. Both the blue code and the yellow code are involutions (self-inverse).

The transforms (*red code*)

```
inline ulong red_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u<<s);
        a ^= (v>>s);
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

and (*green code*)

```
inline ulong green_code(ulong a)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
```

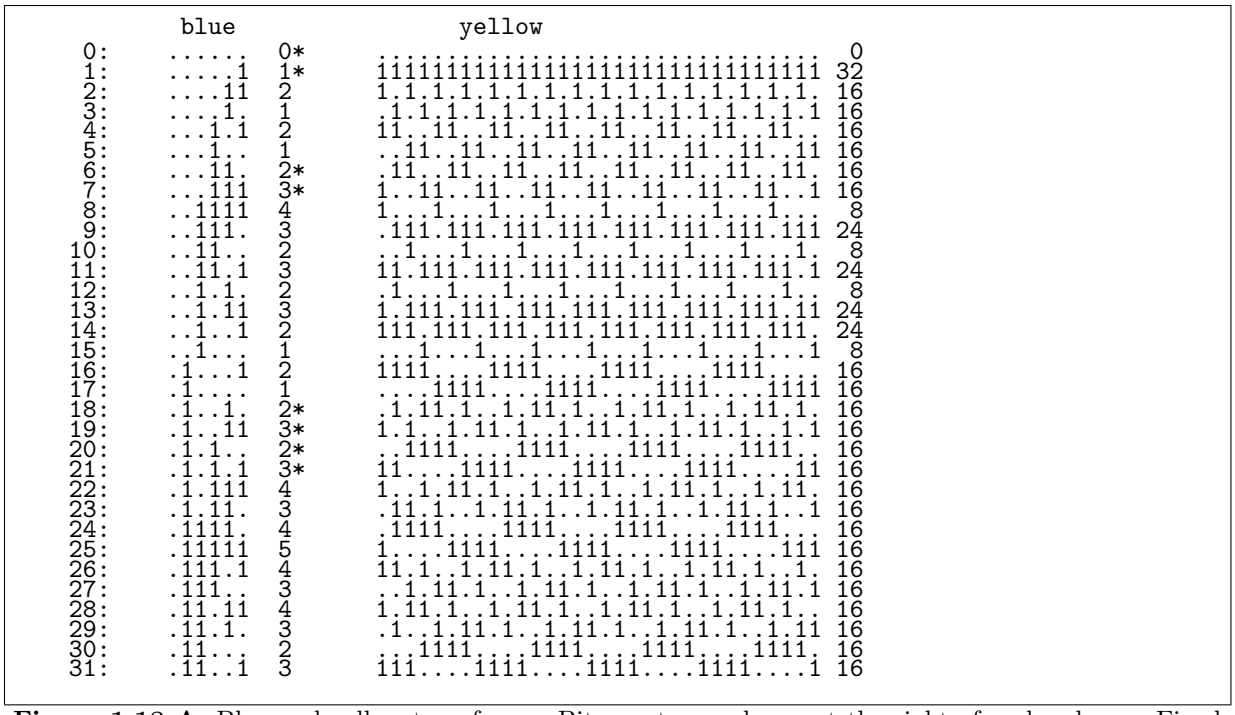


Figure 1.18-A: Blue and yellow transforms. Bit-counts are shown at the right of each column. Fixed points are marked with asterisks.

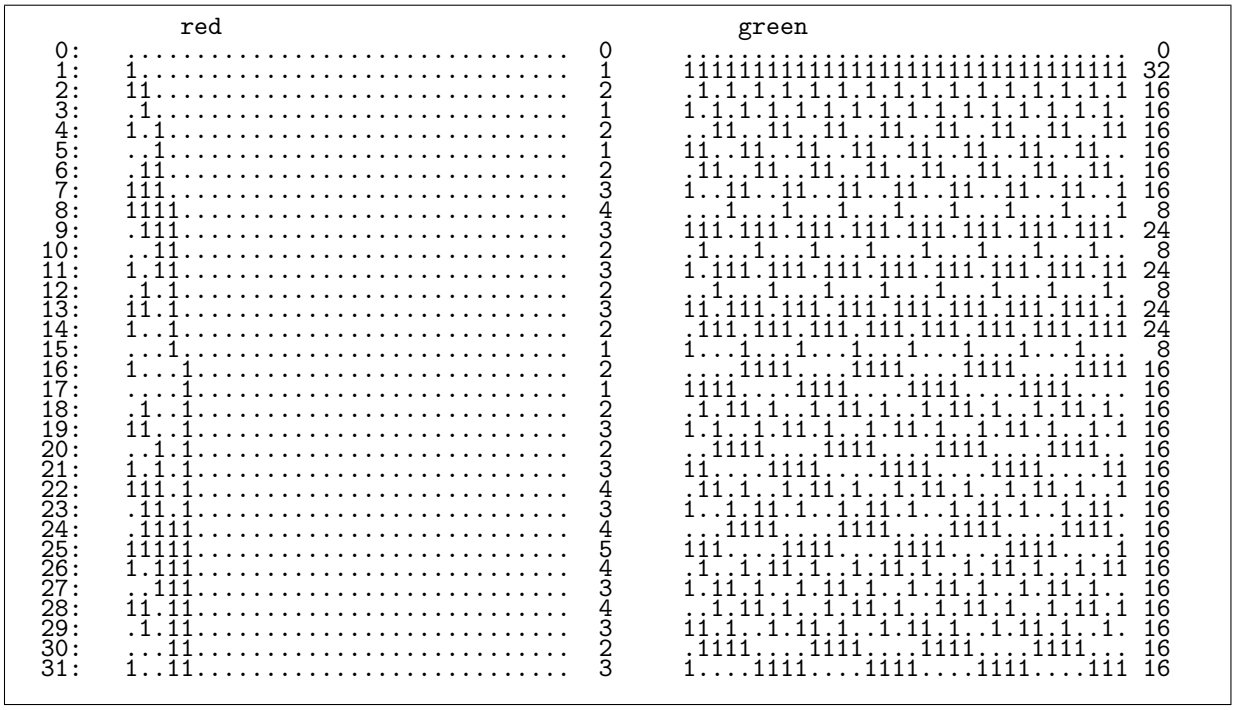


Figure 1.18-B: Red and green transforms. Bit-counts are shown at the right of each column.

```
    while ( s )
    {
        ulong u = a & m;
        ulong v = a ^ u;
        a = v ^ (u>>s);
        a ^= (v<<s);
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

are shown in figure 1.18-B, which was created with the program [FXT: bits/bittransforms-red-demo.cc].

1.18.1 Fixed points of the blue code

```
       0 = ......  :    ..........  =     0
       1 = .....1  :    .........1  =     1
       2 = ....1.  :    .......11.  =     6
       3 = ....11  :    .......111  =     7
       4 = ...1..  :    .....1.1..  =    20
       5 = ...1.1  :    .....1..1.  =    18
       6 = ...11.  :    .....1.1.1  =    21
       7 = ...111  :    .....1..11  =    19
       8 = ..1...  :    ...1111...  =   120
       9 = ..1..1  :    ...11.11..  =   108
      10 = ..1.1.  :    ...111111.  =   126
      11 = ..1.11  :    ...11.1.1.  =   106
      12 = ..11..  :    ...1111..1  =   121
      13 = ..11.1  :    ...11.11.1  =   109
      14 = ..111.  :    ...1111111  =   127
      15 = ..1111  :    ...11.1.11  =   107
      16 = .1....  :    .1...1....  =   272
      17 = .1...1  :    .1.11.1...  =   360
      18 = .1..1.  :    .1.....1..  =   260
      19 = .1..11  :    .1.11111..  =   380
      20 = .1.1..  :    .1...1.11.  =   278
      21 = .1.1.1  :    .1.11.111.  =   366
      22 = .1.11.  :    .1......1.  =   258
      23 = .1.111  :    .1.1111.1.  =   378
      24 = .11...  :    .1...1...1  =   273
      25 = .11..1  :    .1.11.1..1  =   361
      26 = .11.1.  :    .1.....1.1  =   261
      27 = .11.11  :    .1.11111.1  =   381
      28 = .111..  :    .1...1.111  =   279
      29 = .111.1  :    .1.11.1111  =   367
      30 = .1111.  :    .1......11  =   259
      31 = .11111  :    .1.1111.11  =   379
```

Figure 1.18-C: The first fixed points of the blue code. The highest bit of all fixed points lies at an even index. There are $2^{n/2}$ fixed points with highest bit at index n.

The sequence of fixed points of the blue code is (entry A118666 of [214])

```
0, 1, 6, 7, 18, 19, 20, 21, 106, 107, 108, 109, 120, 121, 126, 127, 258, 259, ...
```

If f is a fixed point then f XOR 1 is also a fixed point. Further, $2\,(f \text{ XOR } (2\,f))$ is a fixed point. These facts can be cast into a function that returns a unique fixed point for each argument [FXT: bits/blue-fixed-points.h]:

```
inline ulong blue_fixed_point(ulong s)
{
    if ( 0==s )  return 0;
    ulong f = 1;
    while ( s>1 )
    {
        f ^= (f<<1);
        f <<= 1;
        f |= (s&1);
        s >>= 1;
    }
    return f;
}
```

The output for the first few arguments is shown in figure 1.18-C. Note that the fixed points are not in ascending order. The list was created by the program [FXT: bits/bittransforms-blue-fp-demo.cc].

Now write $f(x)$ for the binary polynomial corresponding to f (see chapter 38 on page 793), if $f(x)$ is a fixed point (that is, $B\,f(x) = f(x+1) = f(x)$), then both $(x^2+x)\,f(x)$ and $1 + (x^2+x)\,f(x)$ are fixed points. The function `blue_fixed_point()` repeatedly multiplies by x^2+x and adds one if the corresponding bit of the argument is set.

The inverse function uses the fact that polynomial division by $x+1$ can be achieved with the inverse reversed Gray code (see section 1.15.6 on page 41) if the polynomial is divisible by $x+1$:

```
inline ulong blue_fixed_point_idx(ulong f)
// Inverse of blue_fixed_point()
{
    ulong s = 1;
    while ( f )
    {
        s <<= 1;
        s ^= (f & 1);
        f >>= 1;
        f = inverse_rev_gray_code(f);  // == bitpol_div(f, 3);
    }
    return s >> 1;
}
```

1.18.2 Relations between the transforms

We write B for the blue code (transform), Y for the yellow code and r for bit-reversal (the `revbin`-function). Then B and Y are connected by the relations

$$B = Y\,r\,Y = r\,Y\,r \tag{1.18-1a}$$
$$Y = B\,r\,B = r\,B\,r \tag{1.18-1b}$$
$$r = Y\,B\,Y = B\,Y\,B \tag{1.18-1c}$$

As said, B and Y are self-inverse:

$$B^{-1} = B, \qquad B\,B = \mathrm{id} \tag{1.18-2a}$$
$$Y^{-1} = Y, \qquad Y\,Y = \mathrm{id} \tag{1.18-2b}$$

The red code and the green code are not involutions ('square roots of identity') but third roots of identity (Using R for the red code, E for the green code):

$$R\,R\,R = \mathrm{id}, \qquad R^{-1} = R\,R = E \tag{1.18-3a}$$
$$E\,E\,E = \mathrm{id}, \qquad E^{-1} = E\,E = R \tag{1.18-3b}$$
$$R\,E = E\,R = \mathrm{id} \tag{1.18-3c}$$

By construction

$$R = r\,B \tag{1.18-4a}$$
$$E = r\,Y \tag{1.18-4b}$$

Relations connecting R and E are:

$$R = E\,r\,E = r\,E\,r \tag{1.18-5a}$$
$$E = R\,r\,R = r\,R\,r \tag{1.18-5b}$$
$$R = R\,E\,R \tag{1.18-5c}$$
$$E = E\,R\,E \tag{1.18-5d}$$

One has

$$r \;=\; Y\,R \;= R\,B \;= B\,E \;= E\,Y \tag{1.18-6}$$

Further

$$B \;=\; R\,Y \;= Y\,E \;= R\,B\,R \;= E\,B\,E \tag{1.18-7a}$$
$$Y \;=\; E\,B \;= B\,R \;= R\,Y\,R \;= E\,Y\,E \tag{1.18-7b}$$

$$R \;=\; B\,Y \;= B\,E\,B \;= Y\,E\,Y \tag{1.18-7c}$$
$$E \;=\; Y\,B \;= B\,R\,B \;= Y\,R\,Y \tag{1.18-7d}$$

and

$$\mathrm{id} \;=\; B\,Y\,E \;= R\,Y\,B \tag{1.18-8a}$$
$$\mathrm{id} \;=\; E\,B\,Y \;= B\,R\,Y \tag{1.18-8b}$$
$$\mathrm{id} \;=\; Y\,E\,B \;= Y\,B\,R \tag{1.18-8c}$$

The following multiplication table lists $z = y\,x$. The R in the third column of the second row says that $r\,B = R$. The letter i is used for identity (id). An asterisk says that $x\,y \neq y\,x$.

	i	r	B	Y	R	E
i	i	r	B	Y	R	E
r	r	i	R*	E*	B*	Y*
B	B	E*	i	R*	Y*	r*
Y	Y	R*	E*	i	r*	B*
R	R	Y*	r*	B*	E	i
E	E	B*	Y*	r*	i	R

1.18.3 Relations to Gray code and reversed Gray code

Write g for the Gray code, then:

$$g\,B\,g\,B \;=\; \mathrm{id} \tag{1.18-9a}$$
$$g\,B\,g \;=\; B \tag{1.18-9b}$$
$$g^{-1}\,B\,g^{-1} \;=\; B \tag{1.18-9c}$$
$$g\,B \;=\; B\,g^{-1} \tag{1.18-9d}$$

Let S_k be the operator that rotates a word by k bits (bit zero is moved to position k, use [FXT: `bit_rotate_sgn()` in bits/bitrotate.h]) then

$$Y\,S_{+1}\,Y \;=\; g \tag{1.18-10a}$$
$$Y\,S_{-1}\,Y \;=\; g^{-1} \tag{1.18-10b}$$
$$Y\,S_{k}\,Y \;=\; g^{k} \tag{1.18-10c}$$

Shift in the frequency domain is derivative in time domain. Relation 1.18-10c, together with a algorithm to generate the cycle leaders of the Gray permutation (section 2.8.1 on page 97) gives a curious method to generate the binary necklaces whose length is a power of two, described in section 17.1.6 on page 341. Let e be the operator for the reversed Gray code, then

$$B\,S_{+1}\,B \;=\; e^{-1} \tag{1.18-11a}$$
$$B\,S_{-1}\,B \;=\; e \tag{1.18-11b}$$
$$B\,S_{k}\,B \;=\; e^{-k} \tag{1.18-11c}$$

1.18.4 More transforms by symbolic powering

The idea of powering a transform (as done for the Gray code in section 1.17 on page 43) can be applied to the 'color'-transforms as exemplified for the blue code:

```
inline ulong blue_xcode(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        if ( x & 1 )  a ^= ( (a&m) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m>>s);
    }
    return  a;
}
```

The result is *not* the power of the blue code which would be pretty boring as $B\,B = \mathrm{id}$. Instead the transform (and the equivalents for Y, R and E, see [FXT: bits/bitxtransforms.h]) are more interesting: all relations between the transforms are still valid, if the symbolic exponent is identical with all terms. For example, we had $B\,B = \mathrm{id}$, now $B^x\,B^x = \mathrm{id}$ is true for all x (there are essentially `BITS_PER_LONG` different x). Similarly, $E\,E = R$ now has to be $E^x\,E^x = R^x$. That is, we have `BITS_PER_LONG` different versions of our four transforms that share their properties with the 'simple' versions. Among them `BITS_PER_LONG` transforms B^x and Y^x that are involutions and E^x and R^x that are third roots of the identity: $E^x\,E^x\,E^x = R^x\,R^x\,R^x = \mathrm{id}$.

While not powers of the simple versions, we still have $B^0 = Y^0 = R^0 = E^0 = \mathrm{id}$. Further, let e be the 'exponent' of all ones and Z be any of the transforms, then $Z^e = Z$, Writing '+' for the XOR operation, then $Z^x\,Z^y = Z^{x+y}$ and so $Z^x\,Z^y = Z$ whenever $x + y = e$.

1.18.5 The building blocks of the transforms

Consider the following transforms on two-bit words where addition is bit-wise (that is, XOR):

$$\mathrm{id}_2\ v = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \tag{1.18-12a}$$

$$r_2\,v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \tag{1.18-12b}$$

$$B_2\,v = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ b \end{bmatrix} \tag{1.18-12c}$$

$$Y_2\,v = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{1.18-12d}$$

$$R_2\,v = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a+b \end{bmatrix} \tag{1.18-12e}$$

$$E_2\,v = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a+b \\ a \end{bmatrix} \tag{1.18-12f}$$

It can easily be verified that for these the same relations hold as for id, r, B, Y, R, E. In fact the 'color-transforms', bit-reversion and (trivially) id are the transforms obtained by the repeated Kronecker-products of the matrices. The transforms are linear over GF(2):

$$Z(\alpha\,a + \beta\,b) = \alpha\,Z(a) + \beta\,Z(b) \tag{1.18-13}$$

The corresponding version of the bit-reversal is [FXT: `xrevbin()` in bits/revbin.h]:

```
inline ulong xrevbin(ulong a, ulong x)
{
    x &= (BITS_PER_LONG-1);  // modulo BITS_PER_LONG
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL >> s;
    while ( s )
    {
        if ( x & 1 )  a = ( (a & m) << s ) ^ ( (a & (~m)) >> s );
        x >>= 1;
        s >>= 1;
        m ^= (m<<s);
    }
    return  a;
}
```

Then, for example, $R^x = r^x\, B^x$ (see relation 1.18-4a on page 48). The yellow code is the bit-wise Reed-Muller transform (described in section 22.12 on page 459) of a binary word. The symbolic powering is equivalent to selecting individual levels of the transform.

1.19 Moves of the Hilbert curve

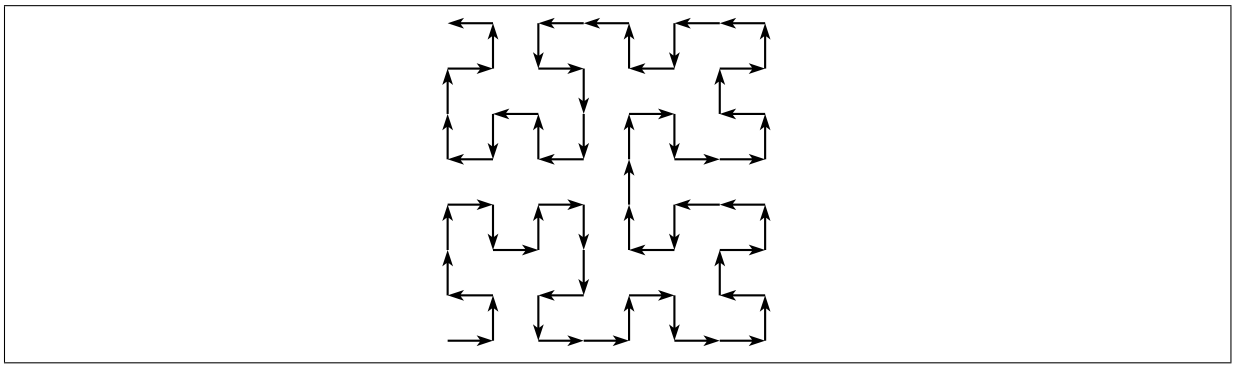


Figure 1.19-A: The Hilbert curve.

```
    dx+dy: ++-+++-+++---++++-+++-+++---++++-+++-+++----+---+---+---++++-
    dx-dy: +----+++-+++-+++-++++---+---+----++++---+---+----+++---+---+--
      dir: >^<^^>v>^>vv<v>>^>v>>^<^>^<<v<^^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^<
     turn: 0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+--
```

Figure 1.19-B: Moves and turns of the Hilbert curve.

A rendering of the Hilbert curve is shown in figure 1.19-A. An efficient algorithm that computes the direction of the n-th move of the Hilbert curve is based on the parity of the number of threes in the radix-4 representation of n (see section 36.9.1 on page 713).

Let d_x and d_y correspond to the moves at step n in the Hilbert curve. Then $d_x, d_y \in \{-1, 0, +1\}$ and exactly one of them is zero. Thereby for both $p := d_x + d_y$ and $m := d_x - d_y$ we have $p, m \in \{-1, +1\}$.

The following function computes p and returns $0, 1$ if $p = -1, +1$, respectively [FXT: bits/hilbert.h]:

```
inline ulong hilbert_p(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx+dy)==-1, else one (then: (dx+dy)==+1).
// Algorithm: count number of threes in radix 4
{
    ulong d = (t & 0x5555555555555555UL) & ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    return  parity( d );
}
```

The function can be slightly optimized as follows (64-bit version only):

```
inline ulong hilbert_p(ulong t)
{
    t &= ((t & 0xaaaaaaaaaaaaaaaaUL) >> 1);
    t ^= t>>2;
    t ^= t>>4;
    t ^= t>>8;
    t ^= t>>16;
    t ^= t>>32;
    return  t & 1;
}
```

The value of m can be computed as:

```
inline ulong hilbert_m(ulong t)
// Let dx,dy be the horizontal,vertical move
// with step t of the Hilbert curve.
// Return  zero if (dx-dy)==-1, else one (then: (dx-dy)==+1).
{
    return  hilbert_p( -t );
}
```

It remains to merge the values of p and m into a two-bit value d that encodes the direction of the move:

```
inline ulong hilbert_dir(ulong t)
// Return d encoding the following move with the Hilbert curve.
//
// d \in {0,1,2,3} as follows:
//   d : direction
//   0 : right (+x:  dx=+1, dy= 0)
//   1 : down  (-y:  dx= 0, dy=-1)
//   2 : up    (+y:  dx= 0, dy=+1)
//   3 : left  (-x:  dx=-1, dy= 0)
{
    ulong p = hilbert_p(t);
    ulong m = hilbert_m(t);
    ulong d = p ^ (m<<1);
    return  d;
}
```

To print the value of d symbolically, one can use the C++ statement `cout << ("v><^")[d];`.

The turn u between steps can be computed as

```
inline int hilbert_turn(ulong t)
// Return the turn (left or right) with the steps
//   t and t-1 of the Hilbert curve.
// Returned value is
//   0 for no turn
//  +1 for right turn
//  -1 for left turn
{
    ulong d1 = hilbert_dir(t);
    ulong d2 = hilbert_dir(t-1);
    d1 ^= (d1>>1);
    d2 ^= (d2>>1);
    ulong u = d1 - d2;
    // at this point, symbolically:  cout << ("+.-0+.-")[ u + 3 ];
    if ( 0==u )  return 0;
    if ( (long)u<0 )  u += 4;
    return  (1==u ? +1 : -1);
}
```

To print the value of u symbolically, one can use `cout << ("-0+")[d+1];`.

The values of p and m, followed by the direction and turn of the Hilbert curve are shown in figure 1.19-B. The list was created with the program [FXT: bits/hilbert-moves-demo.cc]. Figure 1.19-A was created with the program [FXT: bits/hilbert-texpic-demo.cc].

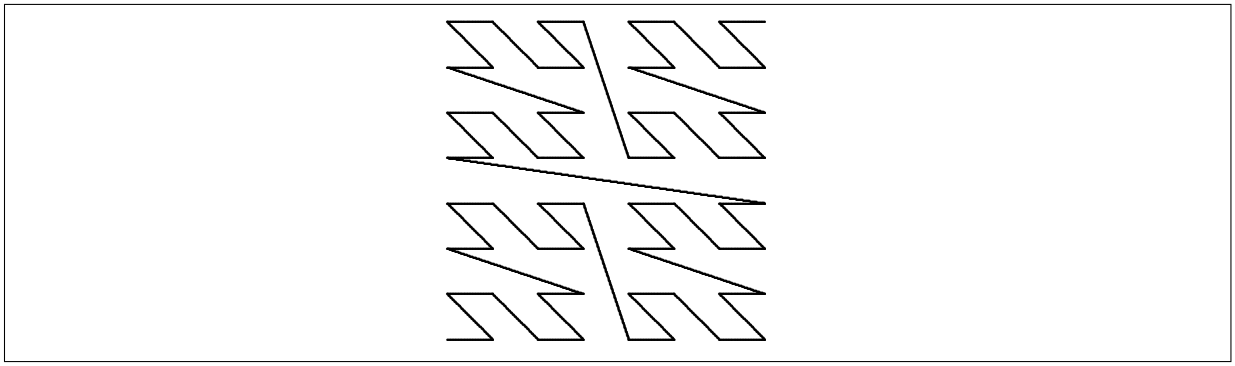


Figure 1.20-A: The Z-order curve.

1.20 The Z-order

A 2-dimensional space-filling curve that traverses all points in each quadrant before it enters the next can be obtained by the *Z-order*. Figure 1.20-A shows a rendering of the Z-order curve, it was created with the program [FXT: bits/zorder-texpic-demo.cc]. The conversion between a linear parameter to a pair coordinates achieved by separating the bits at the even and odd indices. The simple routine is [FXT: bits/zorder.h]:

```
inline void lin2zorder(ulong t, ulong &x, ulong &y)  { bit_unzip2(t, x, y); }
```

The routine `bit_unzip2()` is described in section 1.14 on page 35. The inverse is

```
inline ulong zorder2lin(ulong x, ulong y)  { return bit_zip2(x, y); }
```

From any coordinate pair the next pair can be computed with the following (constant amortized time) routine:

```
inline void zorder_next(ulong &x, ulong &y)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= ~x;
        y ^= b;  b &= ~y;
        b <<= 1;
    }
    while ( b );
}
```

The previous pair is obtained similarly:

```
inline void zorder_prev(ulong &x, ulong &y)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= x;
        y ^= b;  b &= y;
        b <<= 1;
    }
    while ( b );
}
```

The routines are written in a way that generalizes trivially to more dimensions:

```
inline void zorder3d_next(ulong &x, ulong &y, ulong &z)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= ~x;
        y ^= b;  b &= ~y;
        z ^= b;  b &= ~z;
        b <<= 1;
    }
    while ( b );
}
```

```
inline void zorder3d_prev(ulong &x, ulong &y, ulong &z)
{
    ulong b = 1;
    do
    {
        x ^= b;  b &= x;
        y ^= b;  b &= y;
        z ^= b;  b &= z;
        b <<= 1;
    }
    while ( b );
}
```

Unlike the Hilbert curve there are steps where the curve advances more than one unit.

1.21 Scanning for zero bytes

The function (32-bit version)

```
static inline ulong contains_zero_byte(ulong x)
{
    return ((x-0x01010101UL)^x) & (~x) & 0x80808080UL;
}
```

from [FXT: bits/zerobyte.h] determines if any sub-byte of the argument is zero. It returns zero when x contains no zero-byte and nonzero when it does. The idea is to subtract one from each of the bytes and then look for bytes where the borrow propagated all the way to the most significant bit. In order to scan for other values than zero (e.g. 0xa5) one can use `contains_zero_byte( x ^ 0xa5a5a5a5UL )`.

Using the simplified version

```
    return ((x-0x01010101UL) ^ x) & 0x80808080UL;
```

gives false alarms when a byte equals 0x80. For one byte (in hex, omitting prefixes '0x'):

```
      x-01          = 80-01   = 7f
     (x-01)^x       = 7f ^ 80 = ff
    ((x-01)^x) & 80 = ff & 80 = 80   != 0
```

For strings where the high bit of every byte is known to be zero (for example ASCII-strings) the simple version can be used.

The function [FXT: aux1/bytescan.cc]

```
ulong long_strlen(const char *str)
// Return length of string starting at str.
{
    ulong x;
    const char *p = str;

    // Alignment: scan bytes up to word boundary:
    while ( (ulong)p % BYTES_PER_LONG )
    {
        if ( 0 == *p )  return  (ulong)(p-str);
        ++p;
    }

    x = *(ulong *)p;
    while ( ! contains_zero_byte(x) )
    {
        p += BYTES_PER_LONG;
        x = *(ulong *)p;
    }

    // now a zero byte is somewhere in x:
    while ( 0 != *p )  { ++p; }

    return  (ulong)(p-str);
}
```

may be a win for very long strings and word sizes of 64 or more bits.

1.22 2-adic inverse and square root

1.22.1 Computation of the inverse

The 2-adic inverse can be computed using an iteration (see section 28.1.5 on page 543) with quadratic convergence. The number to be inverted has to be odd [FXT: bits/bit2adic.h]:

```
inline ulong inv2adic(ulong x)
// Return inverse modulo 2**BITS_PER_LONG
// x must be odd
// The number of correct bits is doubled with each step
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 3, 6, 12, 24, 48, 96, ... bits (or better)
{
    if ( 0==(x&1) )  return 0;  // not invertible
    ulong i = x;  // correct to three bits at least
    ulong p;
    do
    {
        p = i * x;
        i *= (2UL - p);
    }
    while ( p!=1 );
    return  i;
}
```

Let m be the modulus (a power of two), then the computed value i is the inverse of x modulo m: $i \equiv x^{-1} \bmod m$. It can be used for the so-called *exact division*: to compute the quotient a/x for a number a that is known to be divisible by x, simply multiply by i. This works because $a = b\,x$ (a is divisible by x), so $a\,i \equiv b\,x\,i \equiv b \bmod m$.

1.22.2 Exact division by $C = 2^k \pm 1$

We use the relation (for power series)

$$\frac{A}{C} \;=\; \frac{A}{1-Y} \;=\; A\,(1+Y)\,(1+Y^2)\,(1+Y^4)\,(1+Y^8)\,\ldots\,(1+Y^{2^n}) \mod x^{2^{n+1}} \qquad (1.22\text{-}1)$$

where $Y = 1 - C$. The relation can be used for efficient exact division over $\mathbb{Z}$ by $C = 2^k \pm 1$. For $C = 2^k + 1$ use

$$\frac{A}{C} \;=\; A\,(1-2^k)\,(1+2^{k\,2})\,(1+2^{k\,4})\,(1+2^{k\,8})\,\cdots\,(1+2^{k\,2^u}) \mod 2^N \qquad (1.22\text{-}2)$$

where $k\,2^u \geq N$. For $C = 2^k - 1$ use ($A/C = -A/-C$)

$$\frac{A}{C} \;=\; -A\,(1+2^k)\,(1+2^{k\,2})\,(1+2^{k\,4})\,(1+2^{k\,8})\,\cdots\,(1+2^{k\,2^u}) \mod 2^N \qquad (1.22\text{-}3)$$

The equivalent method for exact division by polynomials (over GF(2)) is given in section 38.1.6 on page 798.

1.22.3 Computation of the square root

With the inverse square root we choose the start value to match $\lfloor d/2 \rfloor + 1$ as that guarantees four bits of initial precision. Moreover, we get control to which of the two possible values the inverse square root is finally reached. The argument modulo 8 has to be equal to one.

```
inline ulong invsqrt2adic(ulong d)
// Return inverse square root modulo 2**BITS_PER_LONG
// Must have:  d==1 mod 8
// The number of correct bits is doubled with each step
```

```
  x   = ...............................1 =  1
  inv = ...............................1
 sqrt = ...............................1

  x   = 11111111111111111111111111111111 = -1
  inv = 11111111111111111111111111111111

  x   = ............................1. =  2

  x   = 11111111111111111111111111111111. = -2

  x   = ..........................i.i.i =  3
  inv = i.i.i.i.i.i.i.i.i.i.i.i.i.i.i.ii

  x   = 11111111111111111111111111111111.1 = -3
  inv = .1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1

  x   = ............................1.. =  4
 sqrt = ............................i.

  x   = 11111111111111111111111111111111.. = -4

  x   = ..........................1.1 =  5
  inv = ii.ii.ii.ii.ii.ii.ii.ii.ii.ii.i1.1

  x   = 11111111111111111111111111111111.11 = -5
  inv = ..11..11..11..11..11..11..11..11..11

  x   = ...........................11. =  6

  x   = 11111111111111111111111111111111.1. = -6

  x   = ........................i.i.i =  7
  inv = i.ii.ii.ii.ii.ii.ii.ii.ii.ii.ii.111

  x   = 11111111111111111111111111111111..1 = -7
  inv = .1..1..1..1..1..1..1..1..1..1..1..1
 sqrt = i..iii..i..ii...ii......i.ii.i.1

  x   = ...........................1... =  8

  x   = 11111111111111111111111111111111... = -8

  x   = ..........................1..1 =  9
  inv = ..iii..iii..iii..iii..iii..iii..i
 sqrt = iiiiiiiiiiiiiiiiiiiiiiiiiiiiiii.1
```

Figure 1.22-A: Examples of the 2-adic inverse and square root of x where $-9 \le x \le +9$. Where no inverse or square root is given, it does not exist.

```
// ==> loop is executed prop. log_2(BITS_PER_LONG) times
// precision is 4, 8, 16, 32, 64, ... bits (or better)
{
    if ( 1 != (d&7) )  return 0;  // no inverse sqrt
    // start value: if d == ****10001 ==> x := ****1001
    ulong x = (d >> 1) | 1;
    ulong p, y;
    do
    {
        y = x;
        p = (3 - d * y * y);
        x = (y * p) >> 1;
    }
    while ( x!=y );
    return  x;
}
```

The square root can be obtained by a final multiplication with d:

```
inline ulong sqrt2adic(ulong d)
// Return square root modulo 2**BITS_PER_LONG
// Must have: d==1 mod 8  or  d==4 mod 32,  d==16 mod 128
//   ... d==4**k mod 4**(k+3)
// Result undefined if condition does not hold
{
    if ( 0==d )  return  0;
    ulong s = 0;
    while ( 0==(d&1) )  { d >>= 1; ++s; }
    d *= invsqrt2adic(d);
    d <<= (s>>1);
    return   d;
}
```

Note that the 2-adic square root is something completely different from the integer square root in general. If the argument d is a perfect square then the result equals $\pm\sqrt{d}$. The output of the program [FXT: bits/bit2adic-demo.cc] is shown in figure 1.22-A. For further information on 2-adic (more generally p-adic) numbers see [155], [105], and also [152].

1.23 Radix -2 representation

The radix -2 representation of a number n is

$$n \;=\; \sum_{k=0}^{\infty} t_k \, (-2)^k \tag{1.23-1}$$

where the t_k are zero or one.

1.23.1 Conversion from binary

```
         k:     bin(k)    m=bin2neg(k)   g=gray(m)   dec(g)
         0:    ......      ......       ......       0 <= 0
         1:    ......1     ......1      ......1      1 <= 1
         2:    .....1.     ....11.      ....1.1      5
         3:    .....11     ....111      ....1..      4
         4:    ....1..     ....1..      ....11.      2
         5:    ....1.1     ....1.1      ....111      3 <= 5
         6:    ....11.     ..11.1.      ..1.111     19
         7:    ....111     ..11.11      ..1.11.     18
         8:    ...1...     ..11...      ..1.1..     20
         9:    ...1..1     ..11..1      ..1.1.1     21
        10:    ...1.1.     ..1111.      ..1...1     17
        11:    ...1.11     ..11111      ..1....     16
        12:    ...11..     ..111..      ..1..1.     14
        13:    ...11.1     ..111.1      ..1..11     15
        14:    ...111.     ..1..1.      ..11.11      7
        15:    ...1111     ..1..11      ..11.1.      6
        16:    ..1....     ..1....      ..11...      8
        17:    ..1...1     ..1...1      ..11..1      9
        18:    ..1..1.     ..1.11.      ..111.1     13
        19:    ..1..11     ..1.111      ..111..     12
        20:    ..1.1..     ..1.1..      ..1111.     10
        21:    ..1.1.1     ..1.1.1      ..11111     11 <= 21
        22:    ..1.11.     11.1.1.      1.11111     75
        23:    ..1.111     11.1.11      1.1111.     74
        24:    ..11...     11.1...      1.111..     76
        25:    ..11..1     11.1..1      1.111.1     77
        26:    ..11.1.     11.111.      1.11..1     73
        27:    ..11.11     11.1111      1.11...     72
        28:    ..111..     11.11..      1.11.1.     70
        29:    ..111.1     11.11.1      1.11.11     71
        30:    ..1111.     11...1.      1.1..11     79
        31:    ..11111     11...11      1.1..1.     78
```

Figure 1.23-A: Radix −2 representations and their Gray codes. Lines ending in '`<=N`' indicate that all values less or equal N occur in the last column up to that point.

Item 128 of [30] gives a surprisingly simple algorithm to obtain the coefficients t_k of the radix −2 representation of a binary number [FXT: bits/negbin.h]:

```
inline ulong bin2neg(ulong x)
// binary --> radix(-2)
{
    const ulong m = 0xaaaaaaaaUL; // 32 bit
    x += m;
    x ^= m;
    return  x;
}
```

An example:

```
    14 -->  ..1..1. == 16 - 2 == (-2)^4 + (-2)^1
```

The inverse routine is obtained by executing the inverse of the two steps in reversed order:

```
inline ulong neg2bin(ulong x)
// radix(-2) --> binary
// inverse of bin2neg()
{
    const ulong m = 0xaaaaaaaaUL;
    x ^= m;
    x -= m;
    return  x;
}
```

Figure 1.23-A shows the output of the program [FXT: bits/negbin-demo.cc]. The sequence of Gray codes of the radix −2 representation is a Gray code for the numbers in the range $0, \ldots, k$ for the following values of k:

$$k \;=\; 1,\, 5,\, 21,\, 85,\, 341,\, 1365,\, 5461,\, 21845,\, 87381,\, 349525,\, 1398101,\, \ldots,\, (4^n - 1)/3$$

1.23.2 Fixed points of the conversion

The sequence of fixed points of the conversion starts as

```
 0 : .........
 1 : ........i
 4 : .......1.
 5 : .......1i
16 : .....1...
17 : .....1...i
20 : .....1.1.
21 : .....1.1.i
64 : ...1......
65 : ...1.....i
68 : ...1...1..
69 : ...1...1.i
80 : ...1.1....
81 : ...1.1...i
84 : ...1.1.1..
85 : ...1.1.1.i
```

This the *Moser – De Bruijn sequence*, entry A000695 of [214]. A related power series is

$$\prod_{k=0}^{\infty}\left(1+x^{4^k}\right) \quad = \quad 1+x+x^4+x^5+x^{16}+x^{17}+x^{20}+x^{21}+x^{64}+x^{65}+x^{68}+\ldots \qquad (1.23\text{-}2)$$

The k-th fixed point can be obtained moving all bits of the binary representation of k to position $2\,x$ where $x \geq 0$ is the index of the bit under consideration:

```
inline ulong negbin_fixed_point(ulong k)
{
    ulong f = 0;
    for (ulong m=1, s=0;  m!=0;  m<<=1, ++s)  f ^= ( (k&m)<<s );
    return  f;
}
```

The sequence of radix -2 representations of 0, 1, 2, ..., interpreted as binary numbers, is entry A005351 of [214]:

```
0,1,6,7,4,5,26,27,24,25,30,31,28,29,18,19,16,17,22,23,20,21,106,107,104,105,110,111, ...
```

The corresponding sequence for the negative numbers -1, -2, -3, ... is entry A005352:

```
3,2,13,12,15,14,9,8,11,10,53,52,55,54,49,48,51,50,61,60,63,62,57,56,59,58,37,36,39,38, ...
```

More information about 'non-standard' representations of numbers can be found in [155].

1.23.3 Generating negbin words in order

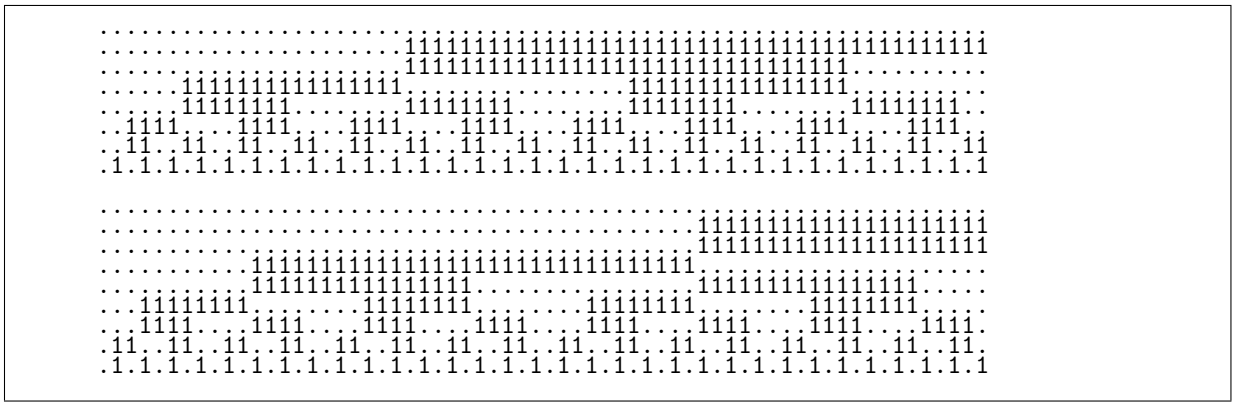


Figure 1.23-B: Radix -2 representations of the numbers $0\ldots+63$ (top), and $0\ldots-63$ (bottom).

A radix -2 representation can be incremented by the function [FXT: bits/negbin.h]:

```
inline ulong next_negbin(ulong x)
// With x the radix(-2) representation of n
// return radix(-2) representation of n+1.
```

```
{
    const ulong m = 0xaaaaaaaaUL;  // 32-bit version
    x ^= m;
    ++x;
    x ^= m;
    return x;
}
```

A version without constants is

```
    ulong s = x << 1;
    ulong y = x ^ s;
    y += 1;
    s ^= y;
    return s;
```

Decrementing can be done via

```
inline ulong prev_negbin(ulong x)
// With x the radix(-2) representation of n
// return radix(-2) representation of n-1.
{
    const ulong m = 0xaaaaaaaaUL;
    x ^= m;
    --x;
    x ^= m;
    return x;
}
```

or via

```
    const ulong m = 0x55555555UL;
    x ^= m;
    ++x;
    x ^= m;
    return x;
```

The functions are quite fast, about 440 million words per second are generated (5 cycles per increment or decrement). Figure 1.23-B shows the generated words in forward (top) and backward (bottom) order, it was created with the program [FXT: bits/negbin2-demo.cc].

1.24 A sparse signed binary representation

An algorithm to compute a representation of a number x as

$$x \;=\; \sum_{k=0}^{\infty} s_k \cdot 2^k \quad \text{where} \quad s_k \in \{-1,\, 0,\, +1\} \tag{1.24-1}$$

such that two consecutive digits s_k, s_{k+1} are never simultaneously nonzero is given in [194]. Figure 1.24-A gives the representation of several small numbers, it is the output of [FXT: bits/bin2naf-demo.cc].

We can convert the binary representation of x into a pair of binary numbers that correspond to the positive and negative digits [FXT: bits/bin2naf.h]:

```
inline void bin2naf(ulong x, ulong &np, ulong &nm)
// Compute (nonadjacent form, NAF) signed binary representation of x:
// the unique representation of x as
//   x=\sum_{k}{d_k*2^k} where d_j \in {-1,0,+1}
//   and no two adjacent digits d_j, d_{j+1} are both nonzero.
// np has bits j set where d_j==+1
// nm has bits j set where d_j==-1
// Thereby:  x = np - nm
{
    ulong xh = x >> 1;  // x/2
    ulong x3 = x + xh;  // 3*x/2
    ulong c = xh ^ x3;
    np = x3 & c;
    nm = xh & c;
}
```

Converting back to binary is trivial:

```
 0:    ......,    ......,     0 =
 1:    ......1    ......P     1 =  +1
 2:    .....1.    .....P.     2 =  +2
 3:    .....11    ....P.M     3 =  +4 -1
 4:    ....1..    ....P..     4 =  +4
 5:    ....1.1    ....P.P     5 =  +4 +1
 6:    ....11.    ...P.M.     6 =  +8 -2
 7:    ....111    ...P..M     7 =  +8 -1
 8:    ...1...    ...P...     8 =  +8
 9:    ...1..1    ...P..P     9 =  +8 +1
10:    ...1.1.    ...P.P.    10 =  +8 +2
11:    ...1.11    ..P.M.M    11 =  +16 -4 -1
12:    ...11..    ..P.M..    12 =  +16 -4
13:    ...11.1    ..P.M.P    13 =  +16 -4 +1
14:    ...111.    ..P..M.    14 =  +16 -2
15:    ...1111    ..P...M    15 =  +16 -1
16:    ..1....    ..P....    16 =  +16
17:    ..1...1    ..P...P    17 =  +16 +1
18:    ..1..1.    ..P..P.    18 =  +16 +2
19:    ..1..11    ..P.P.M    19 =  +16 +4 -1
20:    ..1.1..    ..P.P..    20 =  +16 +4
21:    ..1.1.1    ..P.P.P    21 =  +16 +4 +1
22:    ..1.11.    .P.M.M.    22 =  +32 -8 -2
23:    ..1.111    .P.M..M    23 =  +32 -8 -1
24:    ..11...    .P.M...    24 =  +32 -8
25:    ..11..1    .P.M..P    25 =  +32 -8 +1
26:    ..11.1.    .P.M.P.    26 =  +32 -8 +2
27:    ..11.11    .P..M.M    27 =  +32 -4 -1
28:    ..111..    .P..M..    28 =  +32 -4
29:    ..111.1    .P..M.P    29 =  +32 -4 +1
30:    ..1111.    .P...M.    30 =  +32 -2
31:    ..11111    .P....M    31 =  +32 -1
32:    .1.....    .P.....    32 =  +32
```

Figure 1.24-A: Sparse signed binary representations (nonadjacent from, NAF). The symbols '`P`' and '`M`' are respectively used for +1 and −1, dots denote zeros.

```
inline ulong naf2bin(ulong np, ulong nm)
{
    return ( np - nm );
}
```

The representation is one example of a *nonadjacent form* (NAF). A method for the computation of certain nonadjacent forms (w-NAF) is given in [183]. A Gray code for the signed binary words is described in section 12.5 on page 290.

```
 0:    .......,    .......,     0 =
 1:    .......1    .......P     1 =  +1
 2:    ......1.    ......P.     2 =  +2
 4:    .....1..    .....P..     4 =  +4
 5:    .....1.1    .....P.P     5 =  +4 +1
 8:    ....1...    ....P...     8 =  +8
 9:    ....1..1    ....P..P     9 =  +8 +1
10:    ....1.1.    ....P.P.    10 =  +8 +2
16:    ...1....    ...P....    16 =  +16
17:    ...1...1    ...P...P    17 =  +16 +1
18:    ...1..1.    ...P..P.    18 =  +16 +2
20:    ...1.1..    ...P.P..    20 =  +16 +4
21:    ...1.1.1    ...P.P.P    21 =  +16 +4 +1
32:    ..1.....    ..P.....    32 =  +32
33:    ..1....1    ..P....P    33 =  +32 +1
34:    ..1...1.    ..P...P.    34 =  +32 +2
36:    ..1..1..    ..P..P..    36 =  +32 +4
37:    ..1..1.1    ..P..P.P    37 =  +32 +4 +1
40:    ..1.1...    ..P.P...    40 =  +32 +8
41:    ..1.1..1    ..P.P..P    41 =  +32 +8 +1
42:    ..1.1.1.    ..P.P.P.    42 =  +32 +8 +2
64:    .1......    .P......    64 =  +64
```

Figure 1.24-B: The numbers whose negative part in the NAF representation is zero.

When a binary word contains no consecutive ones then the negative part of the NAF representation is zero. The sequence of values is [0, 1, 2, 4, 5, 8, 9, 10, 16, ...], entry A003714 of [214], see figure 1.24-B. The numbers are called the *Fibbinary numbers*.

1.25 Generating bit combinations

1.25.1 Co-lexicographic (colex) order of the subsets

```
          word =     set          = set (reversed)
         ...111 = { 0, 1, 2 }    = { 2 ,1 ,0 }
         ..1.11 = { 0, 1, 3 }    = { 3 ,1 ,0 }
         ..11.1 = { 0, 2, 3 }    = { 3 ,2 ,0 }
         ..111. = { 1, 2, 3 }    = { 3 ,2 ,1 }
         .1..11 = { 0, 1, 4 }    = { 4 ,1 ,0 }
         .1.1.1 = { 0, 2, 4 }    = { 4 ,2 ,0 }
         .1.11. = { 1, 2, 4 }    = { 4 ,2 ,1 }
         .11..1 = { 0, 3, 4 }    = { 4 ,3 ,0 }
         .11.1. = { 1, 3, 4 }    = { 4 ,3 ,1 }
         .111.. = { 2, 3, 4 }    = { 4 ,3 ,2 }
         1...11 = { 0, 1, 5 }    = { 5 ,1 ,0 }
         1..1.1 = { 0, 2, 5 }    = { 5 ,2 ,0 }
         1..11. = { 1, 2, 5 }    = { 5 ,2 ,1 }
         1.1..1 = { 0, 3, 5 }    = { 5 ,3 ,0 }
         1.1.1. = { 1, 3, 5 }    = { 5 ,3 ,1 }
         1.11.. = { 2, 3, 5 }    = { 5 ,3 ,2 }
         11...1 = { 0, 4, 5 }    = { 5 ,4 ,0 }
         11..1. = { 1, 4, 5 }    = { 5 ,4 ,1 }
         11.1.. = { 2, 4, 5 }    = { 5 ,4 ,2 }
         111... = { 3, 4, 5 }    = { 5 ,4 ,3 }
```

Figure 1.25-A: Combinations $\binom{6}{3}$ in co-lexicographic order. The reversed sets are sorted.

Given a binary word with k bits set the following routine computes the binary word that is the next combination of k bits in co-lexicographic order (see figure 1.25-A). When considering the delta sets (leftmost column), the ordering would be lexicographic. So a more precise term for our ordering is subset-colex (for sets written with elements in increasing order). The delta-set-colex ordering would be the same sequence as shown, but bit-reversed.

The underlying mechanism to determine the successor is to determine the lowest block of ones and move its highest bit one position up. The rest of the block is then moved to the low end of the word [FXT: bits/bitcombcolex.h]:

```
inline ulong next_colex_comb(ulong x)
{
    ulong r = x & -x;       // lowest set bit
    x += r;                 // replace lowest block by a one left to it

    if ( 0==x )  return 0;  // input was last combination

    ulong z = x & -x;       // first zero beyond lowest block
    z -= r;                 // lowest block  (cf. lowest_block())

    while ( 0==(z&1) )  { z >>= 1; }  // move block to low end of word
    return  x | (z>>1);     // need one bit less of low block
}
```

One could replace the while-loop by a bit scan and shift combination. The combinations $\binom{32}{20}$ are generated at a rate of about 165 million per second. The rate is about 135 M/s for the combinations $\binom{32}{12}$.

The following routine computes the predecessor of a combination:

```
inline ulong prev_colex_comb(ulong x)
// Inverse of next_colex_comb()
{
    x = next_colex_comb( ~x );
    if ( 0!=x )  x = ~x;
    return  x;
}
```

The first and last combination can be computed via

```
inline ulong first_comb(ulong k)
```

```
// Return the first combination of (i.e. smallest word with) k bits,
// i.e.  00..001111..1 (k low bits set)
// Must have:  0 <= k <= BITS_PER_LONG
{
    ulong t = ~0UL >> ( BITS_PER_LONG - k );
    if ( k==0 )  t = 0;  // shift with BITS_PER_LONG is undefined
    return t;
}
```

and

```
inline ulong last_comb(ulong k, ulong n=BITS_PER_LONG)
// return the last combination of (biggest n-bit word with) k bits
// i.e.  1111..100..00 (k high bits set)
// Must have:  0 <= k <= n <= BITS_PER_LONG
{
    return  first_comb(k) << (n - k);
}
```

The `if`-statement in `first_comb()` is needed because a shift by more than `BITS_PER_LONG`−1 is undefined by the C-standard, see section 1.1.5 on page 5.

A variant of the presented (colex-) algorithm appears in [30, item 175]. The variant used here avoids the division of the HAKMEM-version and was given by Doug Moore and Glenn Rhoads. The listing in figure 1.25-A can be created with the program [FXT: bits/bitcombcolex-demo.cc]:

```
ulong n = 6,  k = 3;
ulong last = last_comb(k, n);
ulong g = first_comb(k);
ulong gg = 0;
do
{
    // visit combination given as word g
    gg = g;
    g =  next_colex_comb(g);
}
while ( gg!=last );
```

1.25.2 Lexicographic (lex) order of the subsets

lex (5, 3)		colex (5, 2)	
word =	set	word =	set
..111	= { 0, 1, 2 }	...11	= { 0, 1 }
.1.11	= { 0, 1, 3 }	..1.1	= { 0, 2 }
1..11	= { 0, 1, 4 }	..11.	= { 1, 2 }
.11.1	= { 0, 2, 3 }	.1..1	= { 0, 3 }
1.1.1	= { 0, 2, 4 }	.1.1.	= { 1, 3 }
11..1	= { 0, 3, 4 }	.11..	= { 2, 3 }
.111.	= { 1, 2, 3 }	1...1	= { 0, 4 }
1.11.	= { 1, 2, 4 }	1..1.	= { 1, 4 }
11.1.	= { 1, 3, 4 }	1.1..	= { 2, 4 }
111..	= { 2, 3, 4 }	11...	= { 3, 4 }

Figure 1.25-B: Combinations $\binom{5}{3}$ in lexicographic order (left columns). The sets are sorted. The binary words corresponding to lexicographic order are obtained from the bit-reversed complements of the words corresponding to the co-lexicographic order of combinations $\binom{5}{2} = \binom{5}{5-3}$.

The binary words corresponding to combinations $\binom{n}{k}$ in lexicographic order can be obtained as the bit-reversed complements of the words for the combinations $\binom{n}{n-k}$ in colex order, see figure 1.25-B. Note a more precise term for the order is subset-lex (for sets written with elements in increasing order). The sequence is identical to the delta-set-colex order backwards.

The program [FXT: bits/bitcomblex-demo.cc] shows how to compute the subset-lex sequence efficiently:

```
ulong n = 5,  k = 3;
ulong x = first_comb(n-k);        // first colex (n-k choose n)
const ulong m = first_comb(n);    // aux mask
const ulong l = last_comb(k, n);  // last colex
```

```
    ulong ct = 0;
    ulong y;
    do
    {
        y = revbin(~x, n) & m;  // lex order
        // visit combination given as word y
        x = next_colex_comb(x);
    }
    while ( y != 1 );
```

The bit-reversal routine `revbin()` is shown in section 1.13 on page 30. Sections 6.1.1 on page 166 and section 6.1.2 give iterative algorithms for combinations (represented by arrays) in lex and colex order, respectively.

1.26 Generating bit subsets of a given word

1.26.1 Counting order

In order to generate all bit-subsets of a binary word one can use the sparse counting idea shown in section 1.8 on page 20. The implementation is [FXT: `class bit_subset` in bits/bitsubset.h]:

```
class bit_subset
{
public:
    ulong u_;  // current subset
    ulong v_;  // the full set

public:
    bit_subset(ulong v) : u_(0), v_(v)  { ; }
    ~bit_subset() { ; }
    ulong current()  const  { return u_; }
    ulong next()  { u_ = (u_ - v_) & v_;  return u_; }
    ulong prev()  { u_ = (u_ - 1 ) & v_;  return u_; }

    ulong first(ulong v)  { v_=v;  u_=0;  return u_; }
    ulong first()  { first(v_);  return u_; }

    ulong last(ulong v)  { v_=v;  u_=v;  return u_; }
    ulong last()  { last(v_);  return u_; }
};
```

With the word `[...11.1.]` the following sequence of words is produced by subsequent `next()`-calls:

```
.....1.
....1..
....1.1.
...1....
...1...1.
...11...
...11.1.
........
```

A block of ones at the right will result in the binary counting sequence, see [FXT: bits/bitsubset-demo.cc].

1.26.2 Gray code order

We use a method to isolate the changing bit from counting order that does not depend on shifting:

```
    *******0111  =  u
    *******1000  =  u+1
    00000001111  = (u+1) ^ u
    00000001000  = ((u+1) ^ u) & (u+1)  <--= bit to change
```

The method still works if the lowest one are separated by any amount of zeros. In fact, we want to find the single bit that changed from zero to one. The bits that are switched from zero to one in the transition from the word A to B can also be isolated via `X=B&~A`. The implementation is [FXT: `class bit_subset` in bits/bitsubset-gray.h]:

```
class bit_subset_gray
{
public:
```

```
    bit_subset S_;
    ulong g_;  // subsets in Gray code order
    ulong h_;  // highest bit in S_.v_;  needed for the prev() method
public:
    bit_subset_gray(ulong v) : S_(v), g_(0), h_(highest_bit(v))  { ; }
    ~bit_subset_gray()  { ; }
    ulong current()  const { return g_; }
    ulong next()
    {
        ulong u0 = S_.current();
        if ( u0 == S_.v_ )  return first();
        ulong u1 = S_.next();
        ulong x = ~u0 & u1;
        g_ ^= x;
        return g_;
    }
    ulong first(ulong v)  { S_.first(v);  h_=highest_bit(v);  g_=0;  return g_; }
    ulong first()  { S_.first();  g_=0;  return g_; }
  [--snip--]
```

With the word `[...11.1.]` the following sequence of words is produced by subsequent `next()`-calls:

```
    .....1.
    ....1.1.
    ...1....
    ...11...
    ...11.1.
    ...1..1.
    ...1....
    ........
```

A block of ones at the right will result in the binary Gray code sequence, see [FXT: bits/bitsubset-gray-demo.cc]. The method `prev()` computes the previous word in the sequence, note the swapped roles of the variables `u0` and `u1`:

```
  [--snip--]
    ulong prev()
    {
        ulong u1 = S_.current();
        if ( u1 == 0 )  return last();
        ulong u0 = S_.prev();
        ulong x = ~u0 & u1;
        g_ ^= x;
        return g_;
    }
    ulong last(ulong v)  { S_.last(v);  h_=highest_bit(v); g_=h_;  return g_; }
    ulong last()  { S_.last();  g_=h_;  return g_; }
};
```

1.27 Binary words as subsets in lexicographic order

1.27.1 Next and previous word in lexicographic order

The (bit-reversed) binary words in lexicographic order with respect to the subsets can be generated by successive calls to the following function [FXT: bits/bitlex.h]:

```
static inline ulong next_lexrev(ulong x)
// Return next word in subset-lex order.
{
    ulong x0 = x & -x;  // lowest bit
    if ( 1!=x0 )  // easy case: set bit right of lowest bit
    {
        x0 >>= 1;
        x ^= x0;
        return  x;
    }
    else  // lowest bit at word end
    {
        x ^= x0;  // delete lowest bit
        x0 = x & -x;  // new lowest bit ...
```

```
 1:  1...  =  8    {0}
 2:  11..  = 12    {0, 1}
 3:  111.  = 14    {0, 1, 2}
 4:  1111  = 15    {0, 1, 2, 3}
 5:  11.1  = 13    {0, 1, 3}
 6:  1.1.  = 10    {0, 2}
 7:  1.11  = 11    {0, 2, 3}
 8:  1..1  =  9    {0, 3}
 9:  .1..  =  4    {1}
10:  .11.  =  6    {1, 2}
11:  .111  =  7    {1, 2, 3}
12:  .1.1  =  5    {1, 3}
13:  ..1.  =  2    {2}
14:  ..11  =  3    {2, 3}
15:  ...1  =  1    {3}
```

Figure 1.27-A: Binary words corresponding to non-empty subsets of the 4-element set in lexicographic order with respect to subsets. Note the first element of the subsets corresponds to the highest set bit.

```
        x0 >>= 1;  x -= x0;  // ... is moved one to the right
        return  x;
    }
}
```

The bit-reversed representation was chosen because the isolation of the lowest bit is often cheaper than the same operation on the highest bit. Starting with a one-bit word at position $n-1$ one generates the 2^n bit-subsets of length n. The function is used as follows [FXT: bits/bitlex-demo.cc]:

```
ulong n = 4;  // n-bit binary words
ulong x = 1UL<<(n-1);  // first subset
do
{
    // visit word x
}
while ( (x=next_lexrev(x)) );
```

The output for of the program is shown in figure 1.27-A. About 225 million words per second are generated. An equivalent routine for arrays is given in section 8.1.2 on page 192.

```
 [0: ......  =  0 *]   16: .1...1  = 17      32: 1....1  = 33      48: 11..11  = 51
  1: .....1  =  1 *    17: .1..11  = 19      33: 1...11  = 35      49: 11..1.  = 50
  2: ....11  =  3      18: .1..1.  = 18 *    34: 1...1.  = 34 *    50: 11.1.1  = 53
  3: ....1.  =  2      19: .1.1.1  = 21      35: 1..1.1  = 37      51: 11.111  = 55
  4: ...1.1  =  5      20: .1.111  = 23      36: 1..111  = 39      52: 11.11.  = 54
  5: ...111  =  7      21: .1.11.  = 22      37: 1..11.  = 38      53: 11.1..  = 52
  6: ...11.  =  6 *    22: .1.1..  = 20      38: 1..1..  = 36      54: 111..1  = 57
  7: ...1..  =  4      23: .11..1  = 25      39: 1.1..1  = 41      55: 111.11  = 59
  8: ..1..1  =  9      24: .11.11  = 27      40: 1.1.11  = 43      56: 111.1.  = 58
  9: ..1.11  = 11      25: .11.1.  = 26      41: 1.1.1.  = 42      57: 1111.1  = 61
 10: ..1.1.  = 10 *    26: .111.1  = 29      42: 1.11.1  = 45      58: 111111  = 63
 11: ..11.1  = 13      27: .11111  = 31      43: 1.1111  = 47      59: 11111.  = 62
 12: ..1111  = 15      28: .1111.  = 30      44: 1.111.  = 46      60: 1111..  = 60 *
 13: ..111.  = 14      29: .111..  = 28      45: 1.11..  = 44      61: 111...  = 56
 14: ..11..  = 12      30: .11...  = 24      46: 1.1...  = 40      62: 11....  = 48
 15: ..1...  =  8      31: .1....  = 16      47: 11...1  = 49      63: 1.....  = 32
```

Figure 1.27-B: Binary words corresponding to the subsets of the 6-element set, as generated by `prev_lexrev()`. Fixed points are marked with asterisk.

The following function goes backward:

```
static inline ulong prev_lexrev(ulong x)
// Return previous word in subset-lex order.
{
    ulong x0 = x & -x;  // lowest bit
    if ( x & (x0<<1) )  // easy case: next higher bit is set
    {
        x ^= x0;  // delete lowest bit
        return x;
    }
    else
    {
```

```
        x += x0;  // move lowest bit to the left
        x |= 1;   // set rightmost bit
        return x;
    }
}
```

About 250 million words are generated per second. Starting with zero one obtains, by repeated calls to `prev_lexrev()`, a sequence of words that just before the 2^n-th call has visited every word of length n. The generated sequence of words corresponding to subsets of the 6-element set is shown in figure 1.27-B. The sequence 1, 3, 2, 5, 7, 6, 4, 9,... in the right column is entry A108918 of [214]. It turns out to be useful for a special version of fast Walsh transforms described in section 22.6 on page 444.

1.27.2 Conversion between binary- and lex-ordered words

A little contemplation on the structure of the binary words in lexicographic order leads to the routine that allows random access to the k-th lex-rev word (unrank algorithm) [FXT: bits/bitlex.h]:

```
inline ulong negidx2lexrev(ulong k)
{
    ulong z = 0;
    ulong h = highest_bit(k);
    while ( k )
    {
        while ( 0==(h&k) )  h >>= 1;
        z ^= h;
        ++k;
        k &= h - 1;
    }
    return  z;
}
```

Let the inverse function be $T(x)$, then we have $T(0) = 0$ and, with $h(x)$ being the highest power of two not greater than x,

$$T(x) \quad = \quad h(x) - 1 + \begin{cases} T\,(x - h(x)) & \text{if} \quad x - h(x) \neq 0 \\ h(x) & \text{else} \end{cases} \tag{1.27-1}$$

We obtain the rank algorithm by starting with the lowest bit:

```
inline ulong lexrev2negidx(ulong x)
{
    if ( 0==x )  return 0;
    ulong h = x & -x;  // lowest bit
    ulong r = (h-1);
    while ( x^=h )
    {
        r += (h-1);
        h = x & -x;  // next higher bit
    }
    r += h;  // highest bit
    return  r;
}
```

1.27.3 Minimal decompositions into terms $2^k - 1$ *

The least number of terms needed in the sum $x = \sum_k 2^k - 1$ equals the number of bits of the lex-word as shown in figure 1.27-C. The number can be computed as

```
c = bit_count( negidx2lexrev( x ) );
```

Alternatively, one can subtract the greatest integer of the form $2^k - 1$ until x is zero and count the number of subtractions. The sequence of these numbers is entry A100661 of [214]:

```
1,2,1,2,3,2,1,2,3,2,3,4,3,2,1,2,3,2,3,4,3,2,3,4,3,4,5,4,3,2,1,2,3,2,3,...
```

The following function can be used to compute the sequence:

```
   ....1 1      ....1 =  1 =  1
   ...11 2      ...1. =  2 =  1 + 1
   ...1. 1      ...11 =  3 =  3
   ..1.1 2      ..1.. =  4 =  3 + 1
   ..111 3      ..1.1 =  5 =  3 + 1 + 1
   ..11. 2      ..11. =  6 =  3 + 3
   ..1.. 1      ..111 =  7 =  7
   .1..1 2      .1... =  8 =  7 + 1
   .1.11 3      .1..1 =  9 =  7 + 1 + 1
   .1.1. 2      .1.1. = 10 =  7 + 3
   .11.1 3      .1.11 = 11 =  7 + 3 + 1
   .1111 4      .11.. = 12 =  7 + 3 + 1 + 1
   .111. 3      .11.1 = 13 =  7 + 3 + 3
   .11.. 2      .111. = 14 =  7 + 7
   .1... 1      .1111 = 15 = 15
   1...1 2      1.... = 16 = 15 + 1
   1..11 3      1...1 = 17 = 15 + 1 + 1
   1..1. 2      1..1. = 18 = 15 + 3
   1.1.1 3      1..11 = 19 = 15 + 3 + 1
   1.111 4      1.1.. = 20 = 15 + 3 + 1 + 1
   1.11. 3      1.1.1 = 21 = 15 + 3 + 3
   1.1.. 2      1.11. = 22 = 15 + 7
   11..1 3      1.111 = 23 = 15 + 7 + 1
   11.11 4      11... = 24 = 15 + 7 + 1 + 1
   11.1. 3      11..1 = 25 = 15 + 7 + 3
   111.1 4      11.1. = 26 = 15 + 7 + 3 + 1
   11111 5      11.11 = 27 = 15 + 7 + 3 + 1 + 1
   1111. 4      111.. = 28 = 15 + 7 + 3 + 3
   111.. 3      111.1 = 29 = 15 + 7 + 7
   11... 2      1111. = 30 = 15 + 15
   1.... 1      11111 = 31 = 31
```

Figure 1.27-C: Binary words in subset-lex order and their bit counts (left columns). The least number of terms of the form $2^k - 1$ needed in the sum $x = \sum_k 2^k - 1$ (right columns) equals the bit count.

```
void S(ulong f, ulong n) // A100661
{
    static int s = 0;
    ++s;
    cout << s << ",";
    for (ulong m=1; m<n; m<<=1)  S(f+m, m);
    --s;
    cout << s << ",";
}
```

When called with arguments $f = 0$ and $n = 2^k$ it prints the first $2^{k+1} - 1$ numbers of the sequence followed by a zero.

A generating function of the sequence is given by

$$Z(x) \quad := \quad \frac{-1 + 2\,(1-x)\,\prod_{n=1}^{\infty}\left(1 + x^{2^n-1}\right)}{(1-x)^2} \quad = \tag{1.27-2}$$

$$1 + 2x + x^2 + 2x^3 + 3x^4 + 2x^5 + x^6 + 2x^7 + 3x^8 + 2x^9 + 3x^{10} + 4x^{11} + 3x^{12} + 2x^{13} + \ldots$$

1.27.4 The sequence of fixed points *

The sequence of fixed points of the conversion to and from indices is 0, 1, 6, 10, 18, 34, 60, 66, 92, 108, 116, 130, 156, 172, 180, 204, 212, 228, 258, 284, 300, 308, 332, 340, 356, 396, 404, 420, 452, 514, 540, 556, ... is sequence A079471 of [214]. Their values as bit patterns are shown in figure 1.27-D. The crucial observation is that a word is a fixed point exactly if (it equals zero or) its bit-count equals 2^j where j is the index of the lowest set bit.

Now we can find out whether x is a fixed point of the sequence by the following function:

```
static inline bool is_lexrev_fixed_point(ulong x)
// Return whether x is a fixed point in the prev_lexrev() - sequence
{
    if ( x & 1 )
    {
        if ( 1==x )  return  true;
```

```
          0:  ..........          514:    .1.....1.
          1:  .........1          540:    .1....111..
          6:  .......11.          556:    .1...1.11..
         10:  .......1.1.      [--snip--]
         18:  ......1..1.       1556:   .11...1.1..
         34:  .....1...1.       1572:   .11...1..1..
         60:  .....1111..       1604:   .11..1...1..
         66:  ....1....1.       1668:   .11.1....1..
         92:  ....1.111..       1796:   .111.....1..
        108:  ....11.11..       2040:   .11111111..
        116:  ....111.1..       2050:   1.........1.
        130:  ...1.....1.       2076:   1......111..
        156:  ...1..111..       2092:   1.....1.11..
        172:  ...1.1.11..       2100:   1.....11.1..
        180:  ...1.11.1..       2124:   1....1..11..
        204:  ...11..11..       2132:   1....1.1.1..
        212:  ...11.1.1..       2148:   1....11..1..
        228:  ...111..1..      [--snip--]
        258:  ..1......1.       4644:  1..1...1..1..
        284:  ..1...111..       4676:  1..1..1...1..
        300:  ..1..1.11..       4740:  1..1.1....1..
        308:  ..1..11.1..       4868:  1..11.....1..
        332:  ..1.1..11..       5112:  1..1111111...
        340:  ..1.1.1.1..       5132:  1.1......11..
        356:  ..1.11..1..       5140:  1.1.....1.1..
        396:  ..11...11..       5156:  1.1....1..1..
        404:  ..11..1.1..       5188:  1.1...1...1..
        420:  ..11.1..1..       5252:  1.1..1....1..
        452:  ..111...1..       5380:  1.1.1.....1..
```

Figure 1.27-D: Fixed points of the binary to lex-rev conversion.

```
        else        return  false;
    }
    else
    {
        ulong w = bit_count(x);
        if ( w != (w & -w) )  return  false;
        if ( 0==x )  return  true;
        return  0 != ( (x & -x) & w );
    }
}
```

One can also use either of the following tests:

```
    x == negidx2lexrev(x)
    x == lexrev2negidx(x)
```

1.27.5 Recursive generation and relation to a power series *

The following function generates the bit-reversed binary words in reversed lexicographic order:

```
void C(ulong f, ulong n, ulong w)
{
    for (ulong m=1; m<n; m<<=1)  C(f+m, m, w^m);
    print_bin("  ", w, 10);  // visit
}
```

Calling `C(0, 64, 0)` we obtain the list of words shown in figure 1.27-B with the all-zeros word moved to the last position. A slight modification of the function

```
void A(ulong f, ulong n)
{
    cout << "1,";
    for (ulong m=1; m<n; m<<=1)  A(f+m, m);
    cout << "0,";
}
```

generates the power series (sequence A079559 of [214])

$$\prod_{n=1}^{\infty}\left(1+x^{2^n-1}\right) \;=\; 1+x+x^3+x^4+x^7+x^8+x^{10}+x^{11}+x^{15}+x^{16}+\ldots \qquad (1.27\text{-}3)$$

Calling `A(0, 32)` we obtain:

```
1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,0,0,1,1,0,1,1,0,0,0,0, ...
```

Indeed, the lowest bit of the k-th word of the bit-reversed sequence in reversed lexicographic order equals the $(k-1)$-st coefficient in the power series. The sequence can also be generated with a string substitution engine (see chapter 16 on page 331):

```
Start: 1
Rules:
  0 --> 0
  1 --> 110
-------------
0:   (#=2)
  1
1:   (#=4)
  110
2:   (#=8)
  1101100
3:   (#=16)
  1101100110110000
4:   (#=32)
  11011001101100011011001101100000
5:   (#=64)
  1101100110110000110110011011000011011001101100011011001101100000
```

We note that the sequence of sums, prepended by one,

$$1 + x\,\frac{\prod_{n=1}^{\infty}\left(1+x^{2^n-1}\right)}{1-x} \quad = \quad 1 + 1\,x + 2\,x^2 + 2\,x^3 + 3\,x^4 + 4\,x^5 + 4\,x^6 + \ldots \qquad (1.27\text{-}4)$$

has series coefficients

```
1, 1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 7, 8, 8, 8, 8, 9, 10, 10, 11, 12, 12, 12, 13, ...
```

This sequence is entry A046699 of [214]. We have $a(1) = a(2) = 1$ and the sequence satisfies the peculiar recurrence

$$a(n) \quad = \quad a(n - a(n-1)) + a(n - 1 - a(n-2)) \qquad \text{for} \quad n > 2 \qquad (1.27\text{-}5)$$

1.28 Minimal-change bit combinations

The wonderful routine [FXT: bits/bitcombminchange.h]

```
static inline ulong igc_next_minchange_comb(ulong x)
// Return the inverse graycode of the next combination in minchange order.
// Input must be the inverse graycode of the current combination.
{
    ulong g = rev_gray_code(x);
    ulong i = 2;
    ulong cb; // ==candidateBits;
    do
    {
        ulong y = (x & ~(i-1)) + i;
        ulong j = lowest_bit(y) << 1;
        ulong h = !!(y & j);
        cb = ((j-h) ^ g) & (j-i);
        i = j;
    }
    while ( 0==cb );

    return  x + lowest_bit(cb);
}
```

together with

```
static inline ulong igc_last_comb(ulong k, ulong n)
// Return the (inverse graycode of the) last combination
// as in igc_next_minchange_comb()
{
    if ( 0==k )  return 0;
    ulong f = first_sequency(k);
    ulong c = first_comb(n);
```

```
    return c ^ (f>>1);
    // =^=  (by Doug Moore)
//    return  ((1UL<<n) - 1) ^ (((1UL<<k) - 1) / 3);
}
```

(the function `first_sequency()` is defined in section 1.16 on page 42) can be used as

```
static inline ulong next_minchange_comb(ulong x, ulong last)
// not efficient, just to explain the usage
// of igc_next_minchange_comb()
// Must have: last==igc_last_comb(k, n)
{
    x = inverse_gray_code(x);
    if ( x==last )  return 0;
    x = igc_next_minchange_comb(x);
    return  gray_code(x);
}
```

Each combination is different from the preceding one in exactly two positions. For example, using $n = 5$ and $k = 3$ we obtain

```
  x       inverse_gray_code(x)
..111       ..1.1 == first_sequency(k)
.11.1       .1..1
.111.       .1.11
.1.11       .11.1
11..1       1...1
11.1.       1..11
111..       1.111
1.1.1       11..1
1.11.       11.11
1..11       111.1 == igc_last_comb(k, n)
```

The same run of bit combinations could be obtained by going through the Gray codes and omitting all words where the bit-count is not equal to k. The algorithm shown here, however, is much more efficient.

For reasons of efficiency one may prefer code as

```
ulong last = igc_last_comb(k, n);
ulong c, nc = first_sequency(k);
do
{
    c = nc;
    nc = igc_next_minchange_comb(c);
    ulong g = gray_code(c);
    // Here g contains the bitcombination
}
while ( c!=last );
```

which avoids the repeated computation of the inverse Gray code.

```
   n = 6  k = 2            n = 6  k = 3            n = 6  k = 4
....11  ....1.  ....1.  ...111  ...1.1  ...1..  ..1111  ..1.1.  ..1...
...11.  ...1..  ....1.  ..11.1  ..1..1  ....1.  .11.11  .1..1.  ....1.
...1.1  ...11.  ....1.  ..111.  ..1.11  ....1.  .1111.  .1.1..  ....1.
..11..  ..1...  ...1..  ..1.11  ..11.1  ...1..  .111.1  .1.11.  ...1..
..1.1.  ..11..  ....1.  .11..1  .1...1  ....1.  .1.111  .11.1.  ..1...
..1..1  ..111.  ....1.  .11.1.  .1..11  ...1..  11..11  1...1.  ....1.
.11...  .1....  ..1...  .111..  .1.111  ....1.  11.11.  1..1..  ....1.
.1.1..  .11...  ...1..  .1.1.1  .11..1  ....1.  11.1.1  1..11.  ....1.
.1..1.  .111..  ....1.  .1.11.  .11.11  ....1.  1111..  1.1...  ...1..
.1...1  .1111.  ....1.  .1..11  .111.1  ...1..  111.1.  1.11..  ....1.
11....  1.....  .1....  11...1  1....1  ....1.  111..1  1.111.  ...1..
1.1...  11....  ..1...  11..1.  1...11  ...1..  1.1.11  11..1.  ....1.
1..1..  111...  ...1..  11.1..  1..111  ..1...  1.111.  11.1..  ....1.
1...1.  1111..  ....1.  111...  1.1111  ....1.  1.11.1  11.11.  ...1..
1....1  11111.  ....1.  1.1..1  11...1  ....1.  1..111  111.1.  ..1...
                        1.1.1.  11..11  ...1..
                        1.11..  11.111  ....1.
                        1..1.1  111..1  ....1.
                        1..11.  111.11  ...1.
                        1...11  1111.1  ...1..
```

Figure 1.28-A: Minimal-change combinations, their inverse Gray codes and the differences of the inverse Gray codes. The differences are powers of two.

The algorithm in `igc_next_minchange_comb()` uses the fact that the difference of two (inverse Gray codes of) successive combinations is always a power of two, see figure 1.28-A. See also [FXT:

bits/bitcombminchange-demo.cc]. Using this observation one can derive a different version that checks the pattern of the change:

```
static inline ulong igc_next_minchange_comb(ulong x)
// Alternative version.
{
    ulong gx = gray_code( x );
    ulong i = 2;
    do
    {
        ulong y = x + i;
        i <<= 1;
        ulong gy = gray_code( y );
        ulong r = gx ^ gy;
        // Check that change consists of exactly one bit
        // of the new and one bit of the old pattern:
        if ( is_pow_of_2( r & gy ) && is_pow_of_2( r & gx ) )  break;
        // is_pow_of_2(x):=((x & -x) == x)  returns 1 also for x==0.
        // But this cannot happen for both tests at the same time
    }
    while ( 1 );
    return  y;
}
```

This version is the fastest, the combinations $\binom{32}{12}$ are generated at a rate of about 96 million per second, the combinations $\binom{32}{20}$ at a rate of about 83 million per second.

Here is another version which needs the number of set bits as a second parameter:

```
static inline ulong igc_next_minchange_comb(ulong x, ulong k)
// Alternative version, uses the fact that the difference
// of two successive x is the smallest possible power of 2.
{
    ulong y, i = 2;
    do
    {
        y = x + i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

The routine will be fast if the CPU used has a bitcount instruction. The necessary modification for the generation of the previous combination is trivial:

```
static inline ulong igc_prev_minchange_comb(ulong x, ulong k)
// Returns the inverse graycode of the previous combination in minchange order.
// Input must be the inverse graycode of the current combination.
// With input==first the output is the last for n=BITS_PER_LONG
{
    ulong y, i = 2;
    do
    {
        y = x - i;
        i <<= 1;
    }
    while ( bit_count( gray_code(y) ) != k );
    return  y;
}
```

1.29 Fibonacci words

A Fibonacci word is a word that does not contain two successive ones. Whether a given binary word is a Fibonacci word can be tested with the function [FXT: bits/fibrep.h]

```
inline bool is_fibrep(ulong f)
{
    return  ( 0==(f&(f>>1)) );
}
```

1.29.1 Lexicographic order

```
 0: .......    11: ...1.1.    22: .1....1    33: .1.1.1.1    44: 1..1..1.
 1: ......1    12: ...1.1.1   23: .1...1.    34: 1......    45: 1..1.1..
 2: .....1.    13: ..1....    24: .1...1..   35: 1.....1    46: 1..1.1.1
 3: .....1..   14: ..1....1   25: .1...1.1   36: 1....1.    47: 1.1.....
 4: .....1.1   15: ..1...1.   26: .1..1...   37: 1....1..   48: 1.1....1
 5: ....1...   16: ..1..1..   27: .1..1..1   38: 1....1.1   49: 1.1...1.
 6: ....1..1   17: ..1..1.1   28: .1..1.1.   39: 1...1...   50: 1.1..1..
 7: ....1.1.   18: ..1.1...   29: .1.1....   40: 1...1..1   51: 1.1..1.1
 8: ...1....   19: ..1.1..1   30: .1.1...1   41: 1...1.1.   52: 1.1.1...
 9: ...1...1   20: ..1.1.1.   31: .1.1..1.   42: 1..1....   53: 1.1.1..1
10: ...1..1.   21: .1......   32: .1.1.1..   43: 1..1...1   54: 1.1.1.1.
```

Figure 1.29-A: All 55 Fibonacci words with 8 bits in lexicographic order.

To generate all Fibonacci words use the following functions from [FXT: bits/fibrep.h]. For forward order (see figure 1.29-A):

```
inline ulong next_fibrep(ulong x)
// With x the Fibonacci representation of n
// return Fibonacci representation of n+1.
{
    // 2 examples:          //  ex. 1             //  ex.2
    //                      // x == [*]0 010101   // x == [*]0 01010
    ulong y = x | (x>>1);   // y == [*]? 011111   // y == [*]? 01111
    ulong z = y + 1;        // z == [*]? 100000   // z == [*]? 10000
    z = z & -z;             // z == [0]0 100000   // z == [0]0 10000
    x ^= z;                 // x == [*]0 110101   // x == [*]0 11010
    x &= ~(z-1);            // x == [*]0 100000   // x == [*]0 10000
    return x;
}
```

The routine can be used as shown in [FXT: bits/fibrep2-demo.cc]:

```
    ulong n = 7;
    const ulong f = 1UL << n;
    ulong t = 0;
    do
    {
        // visit(t)
        t = next_fibrep(t);
    }
    while ( t!=f );
```

The reversed order can be obtained via

```
    ulong f = 1UL << n;
    while ( f )
    {
        ulong t = prev_fibrep(f);
        f = t;
        // visit(t)
    }
```

which uses the function (64-bit version)

```
inline ulong prev_fibrep(ulong x)
// With x the Fibonacci representation of n
// return Fibonacci representation of n-1.
{
    ulong i = lowest_bit_idx(x);
    x ^= (1UL<<i);
    ++i;
    x ^= (0x5555555555555555UL >> (BITS_PER_LONG-i));
    return x;
}
```

The forward version generates about 200 million words per second, the backward version about 135 million words per second.

```
     j:        k(j)       k(j)-k(j-1)   x=bin2neg(k)     gray(x)
     1:    ....11...1     .........     ...111...1     ...1..1..1 =  27
     2:    ....11....     .........1    ...111....     ...1..1...  =  26
     3:    ....1.1111     ........1     ...111..11     ...1..1.1.  =  28
     4:    ....1.11..     ........11    ...11111..     ...1....1.  =  23
     5:    ....1.1.11     .........1    ...1111111     ...1......  =  21
     6:    ....1.1.1.     .........1    ...111111.     ...1.....1  =  22
     7:    ....1.1..1     .........1    ...1111..1     ...1...1.1  =  25
     8:    ....1.1...     .........1    ...1111...     ...1...1..  =  24
     9:    ....1...11     .......1.1    ...11..111     ...1.1.1..  =  32
    10:    ....1...1.     .........1    ...11..11.     ...1.1.1.1  =  33
    11:    ....1....1     .........1    ...11....1     ...1.1...1  =  30
    12:    ....1.....     .........1    ...11.....     ...1.1....  =  29
    13:    .....11111     ..........1   ...11...11     ...1.1..1.  =  31
    14:    ......11..     .....1..11    .....111..     .....1..1.  =  10
    15:    ......1.11     .........1    .....11111     .....1....  =   8
    16:    ......1.1.     .........1    .....1111.     .....1...1  =   9
    17:    ......1..1     .........1    .....11..1     .....1.1.1  =  12
    18:    ......1...     .........1    .....11...     .....1.1..  =  11
    19:    ........11     ........1.1   .......111     .......1..  =   3
    20:    ........1.     .........1    .......11.     .......1.1  =   4
    21:    .........1     .........1    .........1     .........1  =   1
    22:    ..........     .........1    ..........     ..........  =   0
    23:    1111111111     .........1    ........11     ........1.  =   2
    24:    11111111..     ........11    ......11..     ......1.1.  =   7
    25:    1111111.11     .........1    ......1111     ......1...  =   5
    26:    1111111.1.     .........1    ......111.     ......1..1  =   6
    27:    111111...1     ......1..1    ....11...1     ....1.1..1  =  19
    28:    111111....     .........1    ....11....     ....1.1...  =  18
    29:    11111.1111     .........1    ....11..11     ....1.1.1.  =  20
    30:    11111.11..     ........11    ....1111..     ....1...1.  =  15
    31:    11111.1.11     .........1    ....111111     ....1.....  =  13
    32:    11111.1.1.     .........1    ....11111.     ....1....1  =  14
    33:    11111.1..1     .........1    ....111..1     ....1..1.1  =  17
    34:    11111.1...     .........1    ....111...     ....1..1..  =  16
```

Figure 1.29-B: Gray code for the binary Fibonacci words (rightmost column).

1.29.2 Gray code order

A Gray code for the binary Fibonacci words (shown in figure 1.29-B) can be obtained by using the Gray code of the radix -2 representations (see section 1.23 on page 56) of binary words whose difference is of the form

```
      1    ...............1
      3    ..............11
      5    .............1.1
      9    ............1..1
     19    ...........11..11
     37    ..........1..1.1
     73    .........1..1..1
    147    ........1..1..11
    293    .......1..1..1.1
    585    ......1..1..1..1
   1171    .....1..1..1..11
   2341    ....1..1..1..1.1
   4681    ...1..1..1..1..1
   9363    ..1..1..1..1..11
```

The algorithm is to try these values as increments starting from the least, same as for the minimal-change combination described in section 1.28 on page 69. The next valid word is encountered if it is a valid Fibonacci word, that is, if it does not contain two consecutive set bits. The implementation is [FXT: `class bit_fibgray` in bits/bitfibgray.h]:

```
class bit_fibgray
// Fibonacci Gray code with binary words.
{
public:
    ulong x_;  // current Fibonacci word
    ulong k_;  // aux
    ulong fw_, lw_;  // first and last Fibonacci word in Gray code
    ulong mw_;  // max(fw_, lw_)
    ulong n_;  // Number of bits
public:
    bit_fibgray(ulong n)
    {
        n_ = n;
        fw_=0;
        for (ulong m=(1UL<<(n-1)); m!=0; m>>=3)  fw_ |= m;
```

```
        lw_ = fw_ >> 1;
        if ( 0==(n&1) )  { ulong t=fw_; fw_=lw_; lw_=t; }
        mw_ = ( lw_->fw_ ? lw_ : fw_ );
        x_ = fw_;

        k_ = inverse_gray_code(fw_);
        k_ = neg2bin(k_);
    }

    ~bit_fibgray() {;}

    ulong next()
    // Return next word in Gray code.
    // Return ~0 if current word is the last one.
    {
        if ( x_ == lw_ )  return ~0UL;

        ulong s = n_;
        while ( 1 )
        {
            --s;
            ulong c = 1 | (mw_ >> s);
            ulong i = k_ - c;
            ulong x = bin2neg(i);
            x ^= (x>>1);

            if ( 0==(x&(x>>1)) )
            {
                k_ = i;
                x_ = x;
                return x;
            }
        }
    }
};
```

More than 100 million words per second are generated. The program [FXT: bits/bitfibgray-demo.cc] shows how to use the class, figure 1.29-B was created with it. Section 12.1 on page 282 gives a recursive algorithm for Fibonacci words in Gray code order.

1.30 Binary words and parentheses strings *

```
 0    .... P   [empty string]        .....   [empty string]
 1    ...1 P   ()                    ....1   ()
 2    ..1.                           ...11   (())
 3    ..11 P   (())                  ..1.1   ()()
 4    .1..                           ..111   ((()))
 5    .1.1 P   ()()                  .1.11   (()())
 6    .11.                           .11.1   ()(())
 7    .111 P   ((()))                .1111   (((())))
 8    1...                           1..11   (())()
 9    1..1                           1.1.1   ()()()
10    1.1.                           1.111   ((()()))
11    1.11 P   (()())                11.11   (()(()))
12    11..                           111.1   ()((()))
13    11.1 P   ()(())                11111   ((((()))))
14    111.
15    1111 P   (((())))
```

Figure 1.30-A: Left: some of the 4-bit binary words can be interpreted as a string parentheses (marked with 'P'). Right: all 5-bit words that correspond to well-formed parentheses strings.

A subset of the binary words can be interpreted as a (well formed) string of parentheses. The 4-bit binary words that have this property are marked with a 'P' in figure 1.30-A (left) [FXT: bits/parenword-demo.cc]. The strings are constructed by scanning the word from the low end and printing a '(' with each one and a ')' with each zero. In order to find out when to terminate one adds up +1 for each opening parenthesis and −1 for a closing parenthesis. When the ones in the binary word have been scanned then s closing parentheses have to be added where s is the value of the sum [FXT: bits/parenwords.h]:

```
inline void parenword2str(ulong x, char *str)
```

```
{
    int s = 0;
    ulong j = 0;
    for (j=0; x!=0; ++j)
    {
        s += ( x&1 ? +1 : -1 );
        str[j] = ")("[x&1];
        x >>= 1;
    }
    while ( s-- > 0 )  str[j++] = ')';  // finish string
    str[j] = 0;  // terminate string
}
```

The 5-bit binary words that are valid 'paren words' together with the corresponding strings are shown in figure 1.30-A (right). Note that the lower bits in the word (right end) correspond to the beginning of the string (left end). If a negative value for the sums occurs at any time of the computation then the word is not a paren word. We use that fact to create a routine that determines whether a word is a paren word:

```
inline bool is_parenword(ulong x)
{
    int s = 0;
    for (ulong j=0; x!=0; ++j)
    {
        s += ( x&1 ? +1 : -1 );
        if ( s<0 )  break;  // invalid word
        x >>= 1;
    }
    return  (s>=0);
}
```

The sequence

```
  1, 3, 5, 7, 11, 13, 15, 19, 21, 23, 27, 29, 31, 39, 43, 45, 47, 51, 53, 55, 59, 61, 63, ...
```

of nonzero integers x so that `is_parenword(x)` returns `true` is entry A036991 of [214]. If we fix the number of paren pairs then the following functions generate the least and biggest valid binary word. The function simply returns a word with a block of n ones at the low end:

```
inline ulong first_parenword(ulong n)
// Return least binary word corresponding to n pairs of parens
// Example, n=5:  .....11111   ((((()))))
{
    return first_comb(n);
}
```

The last paren word is the word with a sequence of n blocks '`01`' at the low end:

```
inline ulong last_parenword(ulong n)
// Return biggest binary word corresponding to n pairs of parens.
// Must have: 1 <= n <= BITS_PER_LONG/2.
// Example, n=5:  .1.1.1.1.1   ()()()()()
{
    return  0x5555555555555555UL >> (BITS_PER_LONG-2*n);
}
```

```
  ......11111 = (((((())))))    ...1...1111 = (((()))())    ..1....1111 = (((())))()
  .....1.1111 = (((()())))    ...1..1.111 = (((())()))    ..1...1.111 = (((()))()
  .....11.111 = (((())()))    ...1..11.11 = (((()()))    ..1...11.11 = (((()())))
  .....111.11 = (((()())))    ...1..111.1 = ((()))())    ..1...111.1 = ((()()))()
  .....1111.1 = ((()()()))    ...1.1..111 = ((())()))    ..1..1..111 = ((())())()
  ....1..1111 = ((())(())))    ...1.1.1.11 = ((()()()))    ..1..1.1.11 = ((()()))()
  ....1.1.111 = ((())()()))    ...1.1.11.1 = ((())()))    ..1..1.11.1 = ((())()))()
  ....1.11.11 = ((()())()))    ...1.11..11 = ((()))()))    ..1..11..11 = ((()))())()
  ....1.111.1 = ((()()()))    ...1.11.1.1 = ((()()()))    ..1..11.1.1 = ((()()))()
  ....11..111 = (((()))()))    ...11...111 = (((())))())    ..1.1...111 = (((())))()
  ....11.1.11 = ((()()())))    ...11..1.11 = ((()))())    ..1.1..1.11 = ((()())())
  ....11.11.1 = ((())()()))    ...11..11.1 = ((()))()))    ..1.1..11.1 = ((()))()()
  ....111..11 = ((()))()))    ...11.1..11 = ((()))()))    ..1.1.1..11 = ((()))()()
  ....111.1.1 = ((()()()))    ...11.1.1.1 = ((()()()))    ..1.1.1.1.1 = ((()()()))
```

Figure 1.30-B: The 42 binary words corresponding to all valid pairings of 5 parentheses, in colex order.

The sequence of all binary words corresponding to n pairs of parens in colex order can be generated with the following (slightly cryptic) function:

```
inline ulong next_parenword(ulong x)
// Next (colex order) binary word that is a paren word.
{
    if ( x & 2 )  // Easy case, move highest bit of lowest block to the left:
    {
        ulong b = lowest_zero(x);
        x ^= b;
        x ^= (b>>1);
        return x;
    }
    else // Gather all low "01"s and split lowest nontrival block:
    {
        if ( 0==(x & (x>>1)) )  return 0;
        ulong w = 0;  // word where the bits are assembled
        ulong s = 0;  // shift for lowest block
        ulong i = 1;  // == lowest_bit(x)
        do  // collect low "01"s:
        {
            x ^= i;
            w <<= 1;
            w |= 1;
            ++s;
            i <<= 2;  // == lowest_bit(x);
        }
        while ( 0==(x&(i<<1)) );

        ulong z = x ^ (x+i);  // lowest block
        x ^= z;
        z &= (z>>1);
        z &= (z>>1);
        w ^= (z>>s);
        x |= w;
        return x;
    }
}
```

The program [FXT: bits/parenword-colex-demo.cc] shows how to create a list of binary words corresponding to n pairs of parens (code slightly shortened):

```
    ulong n = 4;  // Number of paren pairs
    ulong pn = 2*n+1;
    char *str = new char[n+1];  str[n] = 0;
    ulong x = first_parenword(n);
    while ( x )
    {
        print_bin("  ", x, pn);
        parenword2str(x, str);
        cout << " = " << str << endl;

        x = next_parenword(x);
    }
```

Its output with $n = 5$ is shown in figure 1.30-B. The $1,767,263,190$ paren words for $n = 19$ are generated at a rate of about 153 million words per second. Chapter 13 on page 299 gives a different formulation of the algorithm.

Knuth gives [157] a very elegant routine for generating the next paren word, the comments are `MMIX` instructions:

```
inline ulong next_parenword(ulong x)
{
    const ulong m0 = -1UL/3;
    ulong t = x ^ m0;               // XOR t, x, m0;
    if ( (t&x)==0 )  return 0;      // current is last
    ulong u = (t-1) ^ t;            // SUBU u, t, 1;  XOR u, t, u;
    ulong v = x | u;                // OR v, x, u;
    ulong y = bit_count( u & m0 );  // SADD y, u, m0;
    ulong w = v + 1;                // ADDU w, v, 1;
    t = v & ~w;                     // ANDN t, v, w;
    y = t >> y;                     // SRU y, t, y;
    y += w;                         // ADDU y, w, y;
    return y;
}
```

The routine is slower, however, about 81 million words per second are generated. A bit-count instruction

in hardware would speed it up significantly. By treating the case of easy update separately as in the other version, a rate of about 137 million words per second is obtained.

1.31 Error detection by hashing: the CRC

A *hash value* is an element from a set H that is computed via a *hash function* f that maps any (finite) sequence of input data to H.

For the sake of simplicity we now consider the case that the input sequences are of fixed size so they are in a fixed set, say S. We further assume that the set S is (much) bigger than H.

$$f : S \to H, \qquad s \mapsto h \tag{1.31-1}$$

where $s \in S$ and $h \in H$.

Two input sequences with different hash values are necessarily different. But, as the hash function maps a larger set to a smaller one, there are different input sequences with identical hash values.

A trivial example is the set $H = \{0, 1\}$ together with a function that count binary digits modulo two, the parity function. Another example is the sum-of-digits test (see section 27.5 on page 536) used to check the multiplication of large numbers. In the test we compute the value of a multi digit number decimal $s \in S$ modulo nine. The crucial additional property of this hash is that with $f(A) = a$, $f(B) = b$, $f(C) = c$ (were A, B, C are decimal numbers), then $A \cdot B = C$ implies $a \cdot b = c$.

A hash function f that is actually useful should have the *mixing* property: it should map the elements $s \in S$ 'randomly' to H. With the sum-of-digits test we could have used rather arbitrary moduli for the hash function. With one exception: the value modulo ten as hash would be pretty useless. No change in any digit except for the last could ever be detected.

The so-called *cyclic redundancy check* (CRC) is a hash where the hash values are binary words of fixed length. The hash function (basically) computes $h = s \bmod c$ where s is a binary polynomial build from the input sequence and c is a binary polynomial that is primitive (see chapter 38 on page 793). We will use polynomials c of degree 64 so the hash values (CRCs) are 64-bit words.

An C++ implementation is given as [FXT: `class crc64` in bits/crc64.h]:

```
class crc64
// 64-bit CRC (cyclic redundancy check)
{
public:
    uint64 a_;  // internal state (polynomial modulo c)
    uint64 c_;  // a binary primitive polynomial
    // (non-primitive c lead to smaller periods)
    // The leading coefficient needs not be present.
    uint64 h_;  // auxiliary

    static const uint64 cc[];  // 16 "random" 64-bit primitive polynomials

public:
    crc64(uint64 c=0)
    {
        if ( 0==c )  c = 0x1bULL; // =^= 64,4,3,1,0 (default)
        init(c);
    }

    ~crc64()  {;}

    void init(uint64 c)
    {
        c_ = c;
        c_ >>= 1;
        h_ = 1ULL<<63;
        c_ |= h_;  // leading coefficient
        reset();
    }

    void reset()  { set_a(~0ULL); }  // all ones
```

```
    void set_a(uint64 a)  { a_=a; }
    uint64 get_a()  const  { return a_; }
   [--snip--]
```

Note that a nonzero initial state (member variable `a`) is used: starting with zero will only go to a nonzero state with the first nonzero bit in the input sequence. That is, input sequences differing only by initial runs of zeros would get the same CRC.

After an instance is created one can feed in bits via `bit_in(b)` where lowest bit of `b` must contain the bit to be used:

```
   [--snip--]
    void shift()
    {
        bool s = (a_ & 1);
        a_ >>= 1;
        if ( 0!=s )  a_ ^= c_;
    }
    uint64 bit_in(unsigned char b)
    {
        a_ ^= (b&1);
        shift();
        return  a_;
    }
   [--snip--]
```

When a byte is to be checksummed we can do better than just feeding in the bits one by one. This is achieved adding the byte followed by eight calls to `shift()`:

```
   [--snip--]
    uint64 byte_in(unsigned char b)
    {
#if 1
        a_ ^= b;
        shift();  shift();  shift();  shift();
        shift();  shift();  shift();  shift();
#else  // identical but slower:
        bit_in(b);  b>>=1;  // bit 0
        bit_in(b);  b>>=1;  // bit 1
        bit_in(b);  b>>=1;  // bit 2
        bit_in(b);  b>>=1;  // bit 3
        bit_in(b);  b>>=1;  // bit 4
        bit_in(b);  b>>=1;  // bit 5
        bit_in(b);  b>>=1;  // bit 6
        bit_in(b);  b>>=1;  // bit 7
#endif
        return  a_;
    }
   [--snip--]
```

The lower block implements the straightforward idea. The program [FXT: bits/crc64-demo.cc] computes the 64-bit CRC of a single byte in both ways.

Binary words are fed in byte by byte, starting from the lower end:

```
    uint64 word_in(uint64 w)
    {
        ulong k = BYTES_PER_LONG_LONG;
        while ( k-- )  { byte_in( (uchar)w );  w>>=8; }
        return  a_;
    }
```

To feed in a given number of bits of a word, use the following method:

```
    uint64 bits_in(uint64 w, uchar k)
    // Feed in the k lowest bits of w
    {
        if ( k&1 )  { a_ ^= (w&1);  w >>= 1;  shift(); }
        k >>= 1;
        if ( k&1 )  { a_ ^= (w&3);  w >>= 2;  shift();  shift(); }
        k >>= 1;
        if ( k&1 )  { a_ ^= (w&15);  w >>= 4;  shift();  shift();  shift();  shift(); }
        k >>= 1;
        while ( k-- )  { byte_in( (uchar)w );  w>>=8; }
```

```
        return  a_;
    }
```

The operation is the optimized equivalent to

```
        while ( k-- )  { bit_in( (uchar)w );  w>>=1; }
```

If two sequences differ in a single block of up to 64 bits, their CRCs will be different. The probability that two different sequences have the same CRC equals $2^{-64} \approx 5.42 \cdot 10^{-20}$. If that is not enough (and one does not want to write a CRC with more than 64 bits) then one can simply use two (or more) instances where different polynomials must be used. Sixteen 'random' primitive polynomials are given [FXT: bits/crc64.cc] as static class member:

```
const uint64 crc64::cc[] = {
    0x5a0127dd34af1e81ULL,  // [0]
    0x4ef12e145d0e3ccdULL,  // [1]
    0x16503f45acce9345ULL,  // [2]
    0x24e8034491298b3fULL,  // [3]
    0x9e4a8ad2261db8b1ULL,  // [4]
    0xb199aecfbb17a13fULL,  // [5]
    0x3f1fa2cc0dfbbf51ULL,  // [6]
    0xfb6e45b2f694fb1fULL,  // [7]
    0xd4597140a01d32edULL,  // [8]
    0xbd08ba1a2d621bffULL,  // [9]
    0xae2b680542730db1ULL,  // [10]
    0x8ec06ec4a8fe8f6dULL,  // [11]
    0xb89a2ecea2233001ULL,  // [12]
    0x8b996e790b615ad1ULL,  // [13]
    0x7eaef8397265e1f9ULL,  // [14]
    0xf368ae22deecc7c3ULL,  // [15]
};
```

These are taken from the list [FXT: data/rand64-hex-primpoly.txt]. Initialize multiple CRCs as follows:

```
    crc64 crca( crc64::cc[0] );
    crc64 crcb( crc64::cc[1] );
```

A class for 32-bit CRCs is given in [FXT: `class crc32` in bits/crc32.h]. Its usage is completely equivalent.

The CRC can easily be implemented in hardware and is, for example, used to detect errors in hard disk blocks. When a block is written its CRC is computed and stored in an extra word. When the block is read, the CRC is computed from the data and compared to the stored CRC. A mismatch indicates an error.

One property that the CRC does not have is cryptographic security. It is possible to intentionally create a data set with a prescribed CRC. With *secure hashes* (like MD5 and SHA1) it is (practically) not possible to do so. Secure hashes can be used to 'sign' data. Imagine you distribute a file (for example an binary executable) over the Internet. You want to make sure that someone downloading the file (from any source) can verify that it is not an altered version (like, in the case of an executable, a malicious program). To do so you create a (secure!) hash value which you publish on your web site. Any person can verify the authenticity of the file by computing the hash and comparing it to the published version. [Note added: recently successful attacks on both MD5 (see [236]) and SHA1 were reported].

1.31.1 Optimization via lookup tables

One can feed in an n-bit word w into the CRC in one step (instead of n steps) as follows: add w to (the CRC word) a. Save the lowest n bits of the result to a variable x. Right shift a by n bits. Add to a the entry x of an auxiliary table t. For $n = 8$ the operation can be implemented as [FXT: `class tcrc64` in bits/tcrc64.h]:

```
    uint64 byte_in(uchar b)
    {
        a_ ^= b;
        uint64 x =  t_[a_ & 255];
        a_ >>= 8;
        a_ ^= x;
        return  a_;
```

```
    }
```

The size of the table t is $2^n = 256$ words. For $n = 1$ the table would have only two entries, zero and c, the polynomial used. Then the implementation reduces to

```
    uint64 bit_in(uchar b)
    {
        a_ ^= (b&1);
        bool s = (a_ & 1);
        a_ >>= 1;
        if ( 0!=s )  a_ ^= c_;  // t[0]=0; t[1]=c_;
        return  a_;
    }
```

which is equivalent to the `bit_in()` routine of the unoptimized CRC.

The lookup table is computed upon class initialization as follows:

```
        for (ulong w=0; w<256; ++w)
        {
            set_a(0);
            for (ulong k = 0; k<8; ++k)  bit_in( (uchar)w>>k );
            t_[w] = a_;
        }
```

The class can use tables of either 16 or 256 words. When a table of size 16 is used, the computation is about 6 times faster than with the non-optimized routine. A table of size 256 gives a speedup by a factor of 12. Optimization techniques based on lookup tables are often used in practical applications, both in hardware and in software, see [58].

1.31.2 Parallel CRCs

A very fast method for checksumming is to compute the CRCs for each bit of the fed-in words in parallel. An array of 64 words is used [FXT: `class pcrc64` in bits/pcrc64.h]:

```
template <typename Type>
class pcrc64
// Parallel computation of 64-bit CRCs for each bit of the input words.
// Primitive polynomial used is x^64 + x^4 + x^3 + x^2 + 1
{
public:
    Type x_[64];    // CRC data
    // bit(i) of x_[0], x_[1], ..., x_[63] is a 64-bit CRC
    //   of bit(i) of all input words
    uint pos_;      // position of constant polynomial term
    const uint m_;  // mask to compute mod 64
```

Upon initialization all words are set to all ones:

```
public:
    pcrc64()
        : m_(63)
    {
        reset();
    }

    ~pcrc64()  { ; }

    void reset()
    {
        pos_ = 0;
        Type ff = 0;  ff = ~ff;
        for (uint k=0; k<64; ++k)  x_[k] = ff;
    }
```

The cyclic shift of the array is avoided by working modulo 64 when feeding in words:

```
    void word_in(Type w)
    {
        uint p = pos_;
        pos_ = (p+1) & m_;
        uint h = (p-1) & m_;
        Type a = x_[p & m_];  // 0
```

```
        p += 2;
        a ^= x_[p & m_];  // 2
        ++p;
        a ^= x_[p & m_];  // 3
        ++p;
        a ^= x_[p & m_];  // 4
        x_[h] = a ^ w;
    }
```

The algorithm corresponds to the Fibonacci setup of the linear feedback shift registers (see section 39.2 on page 836). There is no primitive trinomial with degree a multiple of eight so we use the pentanomial $x^{64} + x^4 + x^3 + x^2 + 1$. With an array size where a primitive trinomial exists the modulo computations would be more expensive. A unrolled routine can be used to feed in multiple words:

```
    void words_in(Type *w, ulong n)
    {
        if ( n&1 )  { word_in(w[0]); ++w; }
        n >>= 1;

        if ( n&1 )  { word_in(w[0]); word_in(w[1]); w+=2; }
        n >>= 1;

        for (ulong k=0; k<n; ++k)
        {
            word_in(w[0]);
            word_in(w[1]);
            word_in(w[2]);
            word_in(w[3]);
            w += 4;
        }
    }
```

The program [FXT: bits/pcrc64-demo.cc] feeds the numbers up to a given value into a `pcrc64<uint>`:

```
int main()
{
    Type n = 32768;
    pcrc64<Type> P;
    for (Type k=0; k<n; ++k)  P.word_in(k);

    // print array P.x_[] here
}
```

This rather untypical type of input data illustrates the independence of the bits in the array `x_[]`:

```
       0:  .................1..1.1.11.111..
       1:  11111111111111111.11.1.1111.111.
       2:  11111111111111111...11...1......
       3:  11111111111111111.1..1111....11.
       4:  11111111111111111.1..1.1111.11..
       5:  .................11...1..1.....1
       6:  .................1.1.11111.1.11.
       7:  111111111111111111.1.1.11.1..1..
       8:  1111111111111111111.1...1.11111.
       9:  111111111111111111.1111.1.1.1...
      10:  ...................111...1..1..1
      11:  11111111111111111..1111.1..1.1.1
      12:  .................11..1.11.11..1.
      13:  .................11.1.1..11.1111
    [--snip--]
      59:  .................1...1.11.11.1.1
      60:  11111111111111111.1..11.111..1.1
      61:  .................1..1111.....11.
      62:  11111111111111111..1..11..1.....
      63:  .................111..1.1111.111
```

The implementation can process about 2 GB of data per second when 64-bit types are used, 1 GB/s with 32-bit types, 500 MB/sec with 16-bit types, and about 230 MB/sec with 8-bit types.

1.32 Permutations via primitives

We give two methods to specify permutations of the bits of a binary word via one or more control words. The methods are suggestions for machine instructions that can serve as primitives for permutations of the bits of a word.

A restricted method

Generalizing the symbolic-powering idea given in section 1.18 on page 45 we can specify a subset of all permutations by selecting bit-blocks of the masks (for 32-bit words)

```
................1111111111111111
........11111111........11111111
....1111....1111....1111....1111
..11..11..11..11..11..11..11..11
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1
```

Subsets of the blocks of the masks can be determined with the bits of a 32-bit word by considering the highest bit of each block:

```
................1................    bits  15 ...
........1...............1.......    bits   7 ...
....1.....1.....1.....1.....1...    bits   3  11 ...
..1...1...1...1...1...1...1...1.    bits   1   5   9  13 ..
.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1    bits   0   2   4   6   8  10  12  14 ...
```

Thereby one can use all (except for the highest) bits of a word to select the blocks where the bits defined by the block and those left to it should be swapped. A implementation of the implied algorithm can be found in [FXT: bits/bitperm1-demo.cc]. Arrays are used to give more readable code:

```
void perm1(uchar *a, ulong ldn, const uchar *x)
// Permute a[] according to the 'control word' x[].
// The length of a[] must be 2**ldn.
{
    long n = 1L<<ldn;
    for (long s=n/2; s>0; s/=2)
    {
        for (long k=0; k<n; k+=s+s)
        {
            if ( x[k+s-1]!='0' )
            {
                // swap regions [a+k,...,a+k+s-1], [a+k+s,...,a+k+2*s-1]:
                swap(a+k, a+k+s, s);
            }
        }
    }
}
```

No attempt has been made to optimize or parallelize the algorithm. We just explore how useful a machine instruction for the permutation of bits would be.

The program uses a fixed size of 16 bits, an 'x' is printed whenever the corresponding bit is set:

```
a=0123456789ABCDEF   bits of the input word
x=0010011000110110   control word
   8:    7
   4:    3  11x
   2:    1   5x  9  13x
   1:    0   2x  4   6x  8  10x 12  14x
a=01326754CDFEAB98   result
```

The control word used leads to the Gray code permutation (see 2.8 on page 97). Assume we use words with N bits. We cannot (for $N > 2$) obtain all $N!$ permutations as we can choose between only 2^{N-1} control words. Now set the word length to $N := 2^n$. The reachable permutations are those where the intervals $[k \cdot 2^j, \ldots, (k+1) \cdot 2^j - 1]$ contain all numbers $[p \cdot 2^j, \ldots, (p+1) \cdot 2^j - 1]$ for all $j \le n$ and $0 \le k < 2^{n-j}$, choosing p for each interval arbitrarily ($0 \le p < 2^{n-j}$). For example, the lower half of the permuted array must contain a permutation of either the lower or the upper half ($j = n-1$) and each pair a_{2y}, a_{2y+1} must contain two elements $2z$, $2z+1$ ($j = 1$). The bit-reversal can be obtained using a control word with all bits set. Alas, the (important!) zip permutation (bit-zip, see section 1.14 on page 35) is unreachable.

The inverse permutation is achieved by changing a single line:

```
for (long s=1; s<n; s+=s)
```

A machine instruction could choose between the two routines via the highest bit in the control word.

A general method

All permutations of $N = 2^n$ elements can be obtained using n control words of N bits. Assume we have a machine instruction that collects bits according to a control word. An eight bit example:

```
a = abcdefgh   input data
x = ..1.11.1   control word (dots for zeros)
      cefh     bits of a where x has a one
    abdg       bits of a where x has a zero
    abdgcefh   result, bits separated according to x
```

We need n such instructions that work on all length-2^k sub-words for $1 \le k \le n$. For example, the instruction working on half words of a 16-bit word would work as

```
a = abcdefgh ABCDEFGH   input data
x = ..1.11.1 1111....   control word (dots for zeros)
      cefh      ABCD    bits of a where x has a one
    abdg     EFGH       bits of a where x has a zero
    abdgcefh EFGHABCD   result, bits separated according to x
```

Note the bits of the different sub-words are not mixed. Now all permutations can be reached if the control word for the 2^k-bit sub-words have exactly 2^{k-1} bits set in all ranges $[j \cdot 2^k, \ldots, (j+1) \cdot 2^k]$.

A control word together with the specification of the instruction used defines the action taken. The following leads to a swap of adjacent bit pairs

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.    k= 1  (2-bit sub-words)
```

while this

```
1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.1.    k= 5 (32 bit sub-words)
```

results in gathering the even and odd indexed bits in the halfwords.

A complete set of permutation primitives for 16-bit words and their effect on a symbolic array of bits (split into groups of four elements for readability) is

```
                                0123 4567 89ab cdef
11111111........    k= 4  ==> 89ab cdef 0123 4567
1111....1111....    k= 3  ==> cdef 89ab 4567 0123
11..11..11..11..    k= 2  ==> efcd ab89 6745 2301
1.1.1.1.1.1.1.1.    k= 1  ==> fedc ba98 7654 3210
```

The top primitive leads to a swap of the left and right half of the bits, the next to a swap of the halves of the half words and so on. The obtained permutation is array reversal. Note that we use array notation (least index left) here.

The resulting permutation depends on the order in which the primitives are used. Starting with full words

```
                                0123 4567 89ab cdef
1.1. 1.1. 1.1. 1.1.  k= 4  ==>  1357 9bdf 0246 8ace
1.1. 1.1. 1.1. 1.1.  k= 3  ==>  37bf 159d 26ae 048c
1.1. 1.1. 1.1. 1.1.  k= 2  ==>  7f3b 5d19 6e2a 4c08
1.1. 1.1. 1.1. 1.1.  k= 1  ==>  f7b3 d591 e6a2 c480
```

The result is different when starting with 2-bit sub-words:

```
                                0123 4567 89ab cdef
1.1. 1.1. 1.1. 1.1.  k= 1  ==>  1032 5476 98ba dcfe
1.1. 1.1. 1.1. 1.1.  k= 2  ==>  0213 4657 8a9b cedf
1.1. 1.1. 1.1. 1.1.  k= 3  ==>  2367 0145 abef 89cd
1.1. 1.1. 1.1. 1.1.  k= 4  ==>  3715 bf9d 2604 ae8c
```

There are $\binom{2z}{z}$ possibilities to have z bits set in a $2z$-bit word. There are 2^{n-k} length-2^k sub-words in a 2^n-bit word so the number of valid control words for that step is

$$\left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}}$$

The product of the number of valid words in all steps gives the number of permutations:

$$(2^n)! \;=\; \prod_{k=1}^{n} \left[\binom{2^k}{2^{k-1}}\right]^{2^{n-k}} \tag{1.32-1}$$

1.33 CPU instructions often missed

Essential

- Bit-shift and bit-rotate instructions that work properly for shifts greater or equal to the word length: the shift instruction should zero the word, the rotate instruction should take the shift modulo word length. The C-language standards leave the results for these operations undefined and compilers simply emit the corresponding assembler instructions. The resulting CPU dependent behavior is both a source of errors and makes certain optimizations impossible.
- A bit-reverse instruction. A fast byte-swap mitigates the problem, see section 1.13 on page 30.
- Instructions that return the index of highest or lowest set bit in a word.
- Fast conversion from integer to float and double (both directions).
- A fused multiply-add instruction for floats.
- Instructions for the multiplication of complex floating point numbers. For example, with 128-bit (SIMD) registers and 64-bit floats:

```
// Here: R1 == A, B;  R2 == C, D;
CMUL R1, R2;
// Here: R2 == A*C-B*D, A*D+B*C;
```

- A sum-diff instruction, such as:

```
// Here:  R1 == A;  R2 == B;
SUMDIFF R1, R2;
// Here:  R1 == A+B;  R2 == A-B;
```

  It serves as a primitive for fast orthogonal transforms.

- An instruction to swap registers. Even better, a conditional version of that.

Nice to have

- A parity bit for the complete machine word. The parity of a word is the number of bits modulo two, not the complement of it. Even better would be an instruction for the inverse Gray code, see section 1.15 on page 36.
- A bit-count instruction, see section 1.8 on page 19. This would also give the parity at bit zero.
- A random number generator: LHCAs (see section 39.7 on page 842) may be candidates.
- A conditional version of more than just the move instruction, possibly as an instruction prefix.
- An instruction to detect zero bytes in a word, see section 1.21 on page 54. The C-convention is to use a zero byte as string terminator. Performance of the string related functions in the C-library could thereby be increased significantly. Ideally the instruction should exist for different word sizes: 4-byte, 8-byte and 16-byte (possibly using SIMD extensions).
- A bit-zip and a bit-unzip instruction, see section 1.14 on page 35. Note this is polynomial squaring over GF(2).
- A bit-gather and a bit-scatter instruction, see [FXT: bits/bitgather.h] and [FXT: bits/bitseparate.h]. This would include bit-zip and its inverse.
- Primitives for permutations of bits, see section 1.32 on page 81.
- Multiplication corresponding to XOR as addition. That is, a multiplication without carries, the one used for polynomials over GF(2). See section 38.1 on page 793 and [FXT: `bitpol_mult()` in bpol/bitpol-arith.h].

Chapter 2

Permutations

In this chapter we first study several special permutations that are used as part of other algorithms like the revbin permutation used in radix-2 FFT algorithms. Then permutations in general together with the operations on them, like composition and inversion, are described.

Algorithms for the generation of all permutations of a given number of objects are given in chapter 10. The connection to mixed radix numbers in factorial base is given in section 10.3.

2.1 The revbin permutation

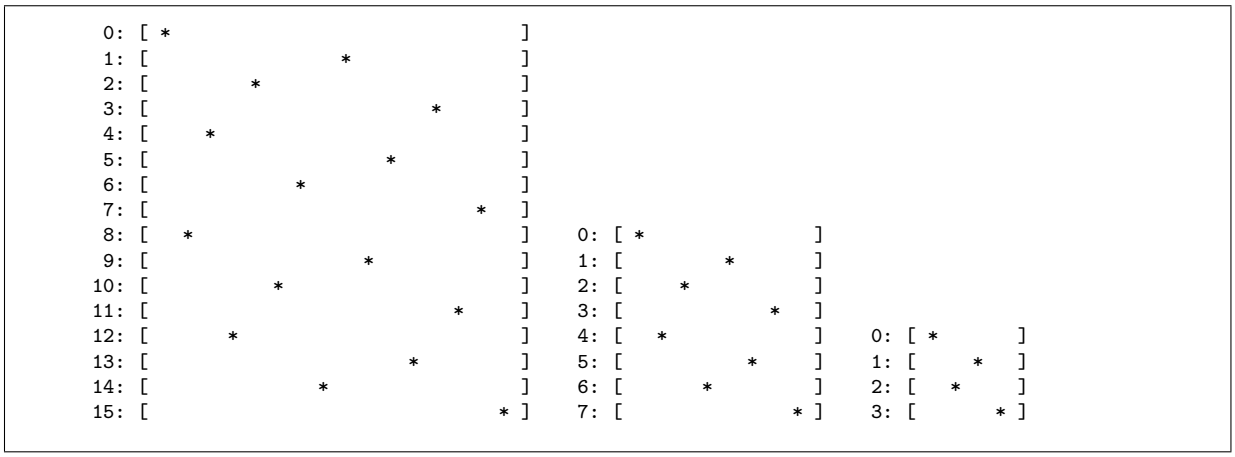


Figure 2.1-A: Permutation matrices of the revbin permutation for sizes 16, 8 and 4. The permutation is self-inverse.

The permutation that swaps elements whose binary indices are mutual reversals is called *revbin permutation* (sometimes also *bit-reversal-* or *bitrev* permutation). For example, for length $n = 256$ the element with index $x = 43_{10} = 00101011_2$ is swapped with the element whose index is $\tilde{x} = 11010100_2 = 212_{10}$. Note that $\tilde{x}$ depends on both x and on n. Pseudo code for a naive implementation is

```
procedure revbin_permute(a[], n)
// a[0..n-1] input,result
{
    for x:=0 to n-1
    {
        r := revbin(x, n)
        if r>x then  swap(a[x], a[r])
    }
}
```

The condition `r>x` before the `swap()` statement makes sure that the swapping is not undone later when the loop variable `x` has the value of the present `r`. The key ingredient for a fast permutation routine is the observation that we only need to update as bit-reversed values. Given $\tilde{x}$ we can compute $\widetilde{x+1}$ efficiently. Methods to do this are given in section 1.13.3. A faster routine will be of the form

```
procedure revbin_permute(a[], n)
// a[0..n-1] input,result
{
    if n<=2  return
    r := 0  // the reversed 0
    for x:=1 to n-1
    {
        r := revbin_upd(r, n/2)  // inline me
        if r>x then  swap(a[x], a[r])
    }
}
```

A method to generate all revbin pairs via linear feedback shift registers is given in section 39.3 on page 837.

About $(n-\sqrt{n})/2$ `swap()`-statements will be executed with the revbin permutation of n elements. That is, almost every element is moved for large n, as there are only few numbers with symmetric bit patterns:

n:	2 # swaps	# symm. pairs
2:	0	2
4:	2	2
8:	4	4
16:	12	4
32:	24	8
64:	56	8
2^{10}:	992	32
2^{20}:	$0.999 \cdot 2^{20}$	2^{10}
∞ :	$n-\sqrt{n}$	$\sqrt{n}$

The sequence

```
0, 2, 4, 12, 24, 56, 112, 238, 480, 992, 1980, 4032, 8064, 16242, 32512, 65280, ...
```

is entry A045687 of [214].

Optimizations using the symmetries of the permutation

The following symmetry can be used for further optimization: if for even $x < \frac{n}{2}$ there is a swap (for the pair x, $\tilde{x}$) then there is also a swap for the pair $n-1-x$, $n-1-\tilde{x}$. As $x < \frac{n}{2}$ and $\tilde{x} < \frac{n}{2}$ one has $n-1-x > \frac{n}{2}$ and $n-1-\tilde{x} > \frac{n}{2}$. That is, the swaps are independent. A routine that uses these observations is

```
procedure revbin_permute(a[], n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        swap(a[x], a[r])
        x := x + 1

        // x even:
        r := revbin_upd(r,n/2)  // inline me
        if r>x then
        {
            swap(a[x], a[r])
            swap(a[n-1-x], a[n-1-r])
        }
        x := x + 1
    }
}
```

The code above can be used to derive an optimized version for zero padded data:

```
procedure revbin_permute0(a[], n)
{
    if n<=2  return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        a[r] := a[x]
        a[x] := 0
        x := x + 1

        // x even:
        r := revbin_upd(r, n)  // inline me
        if r>x then  swap(a[x], a[r])
        // Omit swap of a[n-1-x] and a[n-1-r] as these are zero
        x := x + 1
    }
}
```

One can carry the scheme further, distinguishing whether $x \bmod 4 = 0$, 1, 2 or 3. This is done in the actual C++ implementation [FXT: `revbin_permute()` in perm/revbinpermute.h]. The parameters

```
#define  RBP_SYMM  4  // 1, 2, 4 (default is 4)
#define  FAST_BIT_SCAN  // define if machine has fast bit scan
```

determine how much of the symmetry is used (`RBP_SYMM`) and which flavor of the revbin-update routine is chosen (`FAST_BIT_SCAN`). With a fast bit-scan instruction the table driven version is slightly faster. We further define a convenient macro to swap elements:

```
#define  idx_swap(k, r)  { ulong kx=(k), rx=(r);  swap2(f[kx], f[rx]); }
```

The main routine uses unrolled versions of the revbin permutation for small values of n. These are given in [FXT: perm/shortrevbinpermute.h]. For example, the unrolled routine for $n = 16$ is

```
template <typename Type>
inline void revbin_permute_16(Type *f)
{
    swap2(f[1], f[8]);
    swap2(f[2], f[4]);
    swap2(f[3], f[12]);
    swap2(f[5], f[10]);
    swap2(f[7], f[14]);
    swap2(f[11], f[13]);
}
```

The code was generated with the program [FXT: perm/cycles-demo.cc], see section 2.11.3 on page 106. The routine `revbin_permute_le_64(f,n)` that is called for $n \leq 64$ selects the correct routine for the parameter n:

```
template <typename Type>
void revbin_permute(Type *f, ulong n)
{
    if ( n<=64 )
    {
        revbin_permute_le_64(f, n);
        return;
    }
}
```

If the table driven update is used, the table has to be initialized and, depending on the amount of symmetry used, also some auxiliary variables:

```
    const ulong nh = (n>>1);
#ifdef FAST_BIT_SCAN
    make_revbin_upd_tab(nh);
#endif

#if  ( RBP_SYMM >= 2 )
    const ulong n1  = n - 1;    // = 11111111
#if  ( RBP_SYMM >= 4 )
    const ulong nx1 = nh - 2;   // = 01111110
    const ulong nx2 = n1 - nx1; // = 10111101
#endif //  ( RBP_SYMM >= 4 )
```

```
#endif //  ( RBP_SYMM >= 2 )
```

In what follows we set RBP_SYMM to 4 and omit the corresponding preprocessor statements. The cryptic and powerful main loop is

```
    ulong k = 0,  r = 0;
    while ( k<n/RBP_SYMM  )  // n>=16, n/2>=8, n/4>=4
    {
        // ----- k%4 == 0:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, <nh 11
            idx_swap(n1^k, n1^r);  // >nh, >nh 00
            idx_swap(nx1^k, nx1^r);  // <nh, <nh 11
            idx_swap(nx2^k, nx2^r);  // >nh, >nh 00
        }
        r ^= nh;
        ++k;
        // ----- k%4 == 1:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, >nh 10
            idx_swap(n1^k, n1^r);  // >nh, <nh 01
        }
#ifdef FAST_BIT_SCAN
        r = revbin_tupd(r, k);
#else
        r = revbin_upd(r, nh);
#endif
        ++k;
        // ----- k%4 == 2:
        if ( r>k )
        {
            idx_swap(k, r);  // <nh, <nh 11
            idx_swap(n1^k, n1^r); // >nh, >nh 00
        }
        r ^= nh;
        ++k;
        // ----- k%4 == 3:
        if ( r>k )
        {
            idx_swap(k, r);    // <nh, >nh 10
            idx_swap(nx1^k, nx1^r);   // <nh, >nh 10
        }
#ifdef FAST_BIT_SCAN
        r = revbin_tupd(r, k);
#else
        r = revbin_upd(r, nh);
#endif
        ++k;
    }
}  // end revbin_permute()
```

It turns out that the routine takes, for large n, about six times of the simple `reverse()` operation that swaps elements k with $n-k-k$. Much of the time is spend waiting for memory which suggests that further optimizations would best be attempted with special (machine) instructions to bypass the cache or with non-temporal writes.

The routine [FXT: `revbin_permute0()` in perm/revbinpermute0.h] is a specialized version optimized for zero padded data. Some memory access can be avoided for that case. For example, revbin-pairs with both indices larger than $n/2$ need no processing at all. Therefore the routine is faster than the general version.

If, for complex data, one works with separate arrays for the real and imaginary parts one can remove half of the bookkeeping as follows:

```
procedure revbin_permute(a[], b[], n)
{
    if n<=2  return
    r := 0  // the reversed 0
    for x:=1 to n-1
    {
```

```
        r := revbin_upd(r, n/2)  // inline me
        if r>x then
        {
            swap(a[x], a[r])
            swap(b[x], b[r])
        }
    }
}
```

If both real and imaginary part fit into level-1 cache the method can lead to a speedup. However, for large arrays the routine can be drastically slower than two separate calls of the simple method: typically the real and imaginary element for the same index lie apart in memory by a power of two, leading to one hundred percent cache miss for large array sizes.

2.2 The radix permutation

The *radix permutation* is the generalization of the revbin permutation to arbitrary radices. Pairs of elements are swapped when their indices, written in radix r are reversed. For example, in radix 10 and $n = 1000$ the elements with indices 123 and 321 will be swapped. The radix permutation is self-inverse.

C++ code for the radix r permutation of the array `f[]` is given in [FXT: perm/radixpermute.h]. The routine must be called with n a perfect power of the radix r. Using radix $r = 2$ gives the revbin permutation.

```
extern ulong radix_permute_nt[];  // == 9, 90, 900, ...  for r=10
extern ulong radix_permute_kt[];  // == 1, 10, 100, ...  for r=10
#define  NT  radix_permute_nt
#define  KT  radix_permute_kt

template <typename Type>
void radix_permute(Type *f, ulong n, ulong r)
{
    ulong x = 0;
    NT[0] = r-1;
    KT[0] = 1;
    while ( 1 )
    {
        ulong z = KT[x] * r;
        if ( z>n )  break;
        ++x;
        KT[x] = z;

        NT[x] = NT[x-1] * r;
    }
    // here: n == p**x

    for (ulong i=0, j=0;  i < n-1;  i++)
    {
        if ( i<j )  swap2(f[i], f[j]);

        ulong t = x - 1;
        ulong k = NT[t];  // =^=  k = (r-1) * n / r;

        while ( k<=j )
        {
            j -= k;
            k = NT[--t];  // =^=  k /= r;
        }

        j += KT[t]; // =^=  j += (k/(r-1));
    }
}
```

2.3 In-place matrix transposition

Transposing a matrix is easy when it is not done in-place. The simple routine [FXT: `transpose()` in aux2/transpose.h] does the job:

```
template <typename Type>
void transpose(const Type * restrict f, Type * restrict g, ulong nr, ulong nc)
// Transpose nr x nc matrix f[]  into an nc x nr matrix g[].
{
    for (ulong r=0; r<nr; r++)
    {
        ulong isrc = r * nc;
        ulong idst = r;
        for (ulong c=0; c<nc; c++)
        {
            g[idst] = f[isrc];
            isrc += 1;
            idst += nr;
        }
    }
}
```

Matters get more complicated for the in-place equivalent. We have to find the cycles (see section 2.11 on page 104) of the underlying permutation. To transpose a $n_r \times n_c$- matrix first identify the position i of the entry in row r and column c:

$$i = r \cdot n_c + c \tag{2.3-1}$$

After the transposition the element will be at position i' in the transposed $n'_r \times n'_c$- matrix

$$i' = r' \cdot n'_c + c' \tag{2.3-2}$$

Obviously, $r' = c$, $c' = r$, $n'_r = n_c$ and $n'_c = n_r$, so:

$$i' = c \cdot n_r + r \tag{2.3-3}$$

Multiply the last equation by n_c

$$i' \cdot n_c = c \cdot n_r \cdot n_c + r \cdot n_c \tag{2.3-4}$$

With $n := n_r \cdot n_c$ and $r \cdot n_c = i - c$ we get

$$i' \cdot n_c = c \cdot n + i - c \tag{2.3-5}$$
$$i = i' \cdot n_c - c \cdot (n-1) \tag{2.3-6}$$

Take the equation modulo $n - 1$ to get

$$i \equiv i' \cdot n_c \mod (n-1) \tag{2.3-7}$$

That is, the transposition moves the element $i = i' \cdot n_c$ to position i'. Multiply by n_r to get the inverse:

$$i \cdot n_r \equiv i' \cdot n_c \cdot n_r \tag{2.3-8}$$
$$i \cdot n_r \equiv i' \cdot (n - 1 + 1) \tag{2.3-9}$$
$$i \cdot n_r \equiv i' \tag{2.3-10}$$

That is, element i will be moved to $i' = i \cdot n_r \mod (n-1)$.

The routine [FXT: `transpose()` in aux2/transpose.h] uses the a bit-array to keep track of the elements that have been processed so far:

```
#define  SRC(k)  (((unsigned long long)(k)*nc)%n1)
template <typename Type>
void transpose(Type *f, ulong nr, ulong nc, bitarray *ba=0)
// In-place transposition of an  nr X nc  array
//   that lies in contiguous memory.
{
    if ( 1>=nr )  return;
    if ( 1>=nc )  return;

    if ( nr==nc )  transpose_square(f, nr);
```

```
    else
    {
        const ulong n1 = nr * nc - 1;

        bitarray *tba = 0;
        if ( 0==ba )  tba = new bitarray(n1);
        else          tba = ba;
        tba->clear_all();

        for (ulong k=1;  k<n1;  k=tba->next_clear(++k) )  //  0 and n1 are fixed points
        {
            // do a cycle:
            ulong ks = SRC(k);
            ulong kd = k;
            tba->set(kd);
            Type t = f[kd];
            while ( ks != k )
            {
                f[kd] = f[ks];
                kd = ks;
                tba->set(kd);
                ks = SRC(ks);
            }
            f[kd] = t;
        }

        if ( 0==ba )  delete tba;
    }
}
```

Note that one should take care of possible overflows in the calculation $i \cdot n_c$. For the case that n is a power of two (and so are both n_r and n_c) the multiplications modulo $n-1$ are cyclic shifts. Thus any overflow can be avoided and the computation is also significantly cheaper. A C++ implementation is given in [FXT: aux2/transpose2.h].

2.4 Revbin permutation and matrix transposition *

How would you rotate an (length-n) array by s positions (left or right), *without* using any scratch space. If you do not know the solution then try to find it before reading on.

The trick is to use `reverse()` three times as in the following [FXT: `rotate_left()` in perm/rotate.h]:

```
template <typename Type>
void rotate_left(Type *f, ulong n, ulong s)
// Rotate towards element #0
// Shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }

    reverse(f,   s);
    reverse(f+s, n-s);
    reverse(f,   n);
}
```

The technique is usually called the *triple reversion trick*. For example left-rotating an 8-element array by 3 positions is achieved by the following steps:

```
    [ 1 2 3 4 5 6 7 8 ]
    [ 3 2 1 4 5 6 7 8 ]  reverse first 3 elements
    [ 3 2 1 8 7 6 5 4 ]  reverse last 8-3=5 elements
    [ 4 5 6 7 8 1 2 3 ]  reverse whole array
```

Similarly for the other direction. A right rotation of an n-element array by s positions is identical to a left rotation by $n-s$ positions:

```
    [ 1 2 3 4 5 6 7 8 ]
    [ 5 4 3 2 1 6 7 8 ]  reverse first 8-3=5 elements
    [ 5 4 3 2 1 8 7 6 ]  reverse last 3 elements
```

```
    [ 6 7 8 1 2 3 4 5 ]  reverse whole array
```

```
template <typename Type>
void rotate_right(Type *f, ulong n, ulong s)
// Rotate away from element #0
// Shift is taken modulo n
{
    if ( s==0 )  return;
    if ( s>=n )
    {
        if (n<2) return;
        s %= n;
    }
    reverse(f,     n-s);
    reverse(f+n-s, s);
    reverse(f,     n);
}
```

What this has to do with our subject? When transposing an $n_r \times n_c$ matrix whose size is a power of two (thereby both n_r and n_c are also powers of two) the above mentioned rotation is done with the *indices* (written in base two) of the elements. We know how to do a permutation that reverses the complete indices and reversing a few bits at the least significant end is not any harder:

```
template <typename Type>
void revbin_permute_rows(Type *f, ulong ldn, ulong ldnc)
// Revbin-permute the length 2**ldnc rows  of f[0..2**ldn-1]
// (f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix)
{
    ulong n = 1UL<<ldn;
    ulong nc = 1UL<<ldnc;
    for (ulong k=0; k<n; k+=nc)  revbin_permute(f+k, nc);
}
```

And there we go:

```
template <typename Type>
void transpose_by_rbp(Type *f, ulong ldn, ulong ldnc)
// Transpose f[] considered as an 2**(ldn-ldnc) x 2**ldnc matrix
{
    revbin_permute_rows(f, ldn, ldnc);
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    revbin_permute_rows(f, ldn, ldn-ldnc);  // ... that is, columns
}
```

The triple-reversion trick can also be used to swap two blocks in an array: first reverse the three ranges (first blocks, range between block, last block), then reverse the range that consists of all three. This is the *quadruple reversion trick*. The corresponding code is given in [FXT: perm/swapblocks.h]:

```
template <typename Type>
void swap_blocks(Type *f, ulong x1, ulong n1, ulong x2, ulong n2)
// Swap the blocks starting at indices x1 and x2
// n1 and n2 are the block lengths
{
    if ( x1>x2 ) { swap2(x1,x2); swap2(n1,n2); }
    f += x1;
    x2 -= x1;
    ulong n = x2 + n2;
    reverse(f, n1);
    reverse(f+n1, n-n1-n2);
    reverse(f+x2, n2);
    reverse(f, n);
}
```

The elements before `x1` and after `x2+n2` are not accessed. An example [FXT: perm/swap-blocks-demo.cc]:

```
                v v v v v     v v v v         <--= want to swap these blocks
    [ 0 1 2 3 4  a b c d e 7 8 w x y z  N N ]  orig. data
    [ 0 1 2 3 4  e d c b a 7 8 w x y z  N N ]  reverse first block
    [ 0 1 2 3 4  e d c b a 8 7 w x y z  N N ]  reverse range between blocks
    [ 0 1 2 3 4  e d c b a 8 7 z y x w  N N ]  reverse second block
    [ 0 1 2 3 4  w x y z 7 8 a b c d e  N N ]  reverse whole range
                 ^ ^ ^ ^     ^ ^ ^ ^ ^         <--= the swapped blocks
```

The effect of `swap_blocks(f, x1, n1, x2, n2)` can be undone via `swap_blocks(f, x1, n2, x2+n2-n1, n1)`.

2.5 The zip permutation

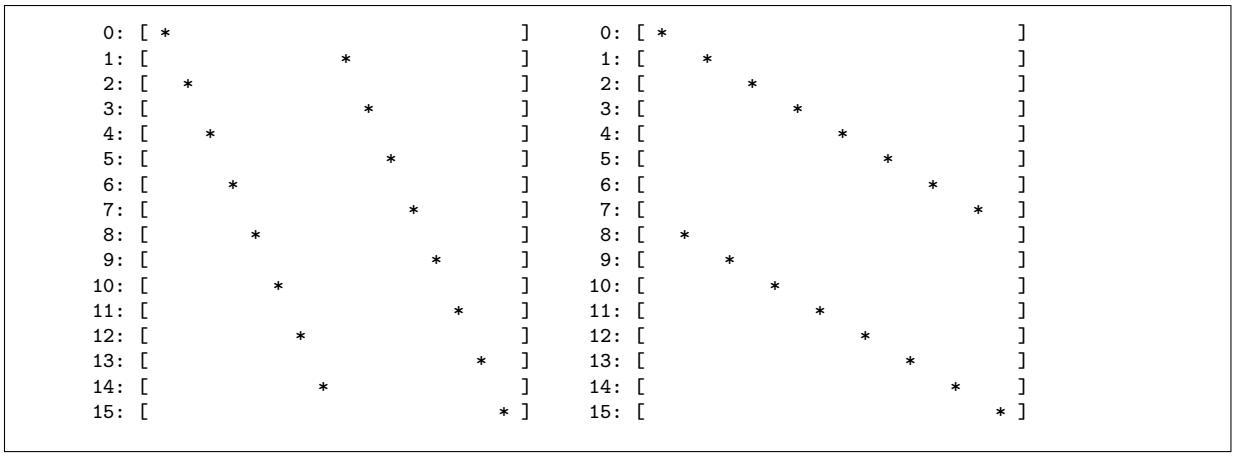


Figure 2.5-A: Permutation matrices of the zip permutation (left) and its inverse, the unzip permutation (right). The zip permutation moves the lower half of the array to the even indices and the upper half to the odd indices.

The *zip permutation* moves the elements from the lower half to the even indices and the elements from the upper half to the odd indices. Symbolically,

```
[ a b c d A B C D ]  |--> [ a A b B c C d D ]
```

The size of the array must be even. A routine for the permutation is

```
template <typename Type>
void zip(const Type * restrict f, Type * restrict g, ulong n)
{
    ulong nh = n/2;
    for (ulong k=0,  k2=0;  k<nh;  ++k, k2+=2)  g[k2] = f[k];
    for (ulong k=nh, k2=1;  k<n;   ++k, k2+=2)  g[k2] = f[k];
}
```

When the array size is a power of two we can use a special case of the 'transposition by revbin permutation' idea to do the operation in-place [FXT: `zip()` in perm/zip.h]:

```
template <typename Type>
void zip(Type *f, ulong n)
{
    ulong nh = n/2;
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
    revbin_permute(f, n);
}
```

If we have a type `Complex` consisting of two `double`s lying contiguous in memory we can optimize the procedure:

```
void zip(double *f, long n)
{
    revbin_permute(f, n);
    revbin_permute((Complex *)f, n/2);
}
```

The inverse of the zip permutation is the *unzip permutation*. We only give the in-place version based on the revbin permutation, the array size must be a power of two:

```
template <typename Type>
void unzip(Type *f, ulong n)
{
```

```
    ulong nh = n/2;
    revbin_permute(f, n);
    revbin_permute(f, nh);  revbin_permute(f+nh, nh);
}
```

The routine can for the type `double` again be optimized as

```
void unzip(double *f, long n)
{
    revbin_permute((Complex *)f, n/2);
    revbin_permute(f, n);
}
```

Connection to matrix transposition

For arrays whose size n is not a power of two the in-place zip permutation can be obtained by transposing the data as a $2 \times n/2$ matrix:

```
transpose(f, 2, n/2);  // =^= zip(f, n)
```

The routines for in-place transposition are given in section 2.3 on page 89. The inverse is clearly obtained by transposing the data as a $n/2 \times 2$ matrix:

```
transpose(f, n/2, 2);  // =^= unzip(f, n)
```

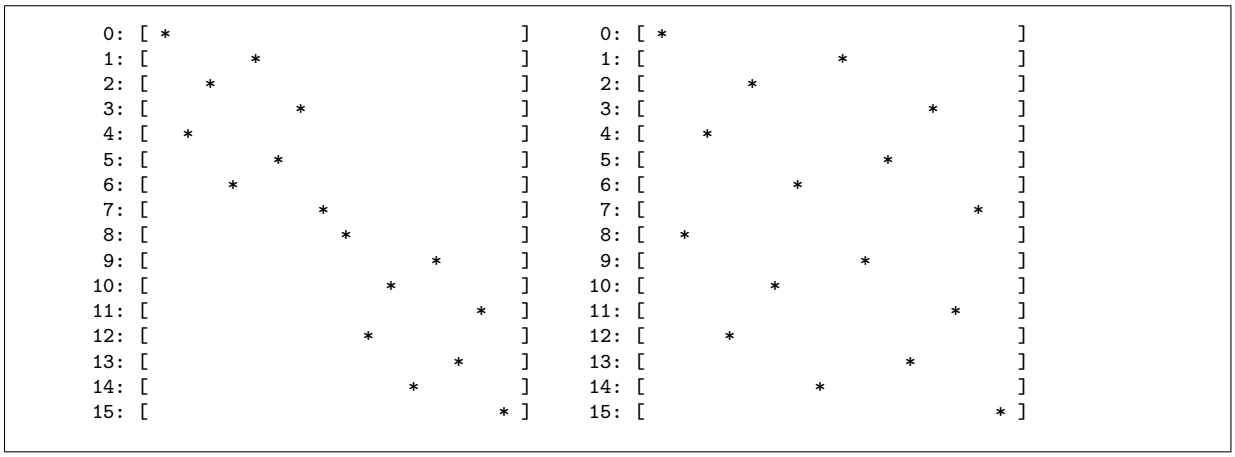


Figure 2.5-B: Revbin permutation matrices that, when multiplied together, give the zip permutation and its inverse. Let L and R be the permutations given on the left and right side, respectively. Then $Z = R\,L$ and $Z^{-1} = L\,R$.

While the above mentioned technique is usually *not* a gain for doing a transposition it may be used to speed up the revbin permutation itself. We operator-ize the idea to see how. Let R be the revbin permutation `revbin_permute()`, $T(n_r, n_c)$ the transposition of the $n_r \times n_c$ matrix and $R(n_c)$ the operation done by `revbin_permute_rows()` (see section 2.4 on page 91). Then

$$T(n_r, n_c) \quad = \quad R(n_r) \cdot R \cdot R(n_c) \tag{2.5-1}$$

The R-operators are their own inverses while T is in general not self inverse[1].

$$R \quad = \quad R(n_r) \cdot T(n_r, n_c) \cdot R(n_c) \tag{2.5-2}$$

There is a degree of freedom in this formula: for fixed $n = n_r \times n_c$ one can choose one of n_r and n_c (only their product is given).

[1] For $n_r = n_c$ it of course is.

2.6 The reversed zip permutation

A permutation closely related to the zip permutation is the *reversed zip permutation*. It moves the lower half of an array to the even indices and the upper half to the odd indices in reversed order. Symbolically,

```
[ a b c d A B C D ]  |--> [ a D b C c B d A ]
```

A C++ routine is [FXT: `zip_rev()` in perm/ziprev.h]:

```
template <typename Type>
void zip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k2] = x[k];
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k2] = x[k];
}
```

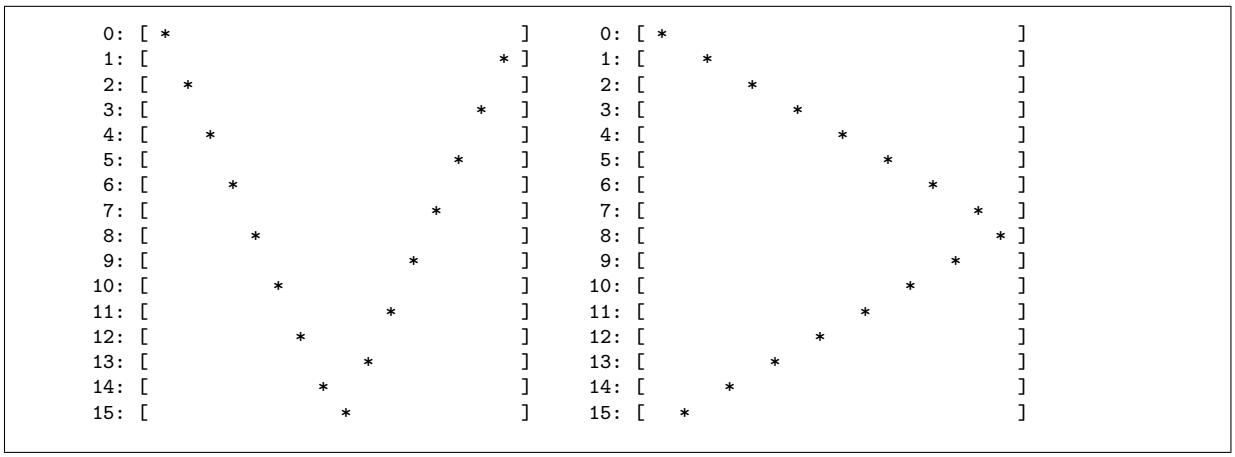


Figure 2.6-A: Permutation matrices of the reversed zip permutation (left) and its inverse (right).

The in-place version can, if the array length is a power of two, be implemented as

```
template <typename Type>
void zip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    reverse(x+nh, nh);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    revbin_permute(x, n);
}
```

The inverse permutation [FXT: `unzip_rev()` in perm/ziprev.h] can be implemented as

```
template <typename Type>
void unzip_rev(const Type * restrict x, Type * restrict y, ulong n)
// n must be even
{
    const ulong nh = n/2;
    for (ulong k=0, k2=0;     k<nh;  k++, k2+=2)  y[k] = x[k2];
    for (ulong k=nh, k2=n-1;  k<n;   k++, k2-=2)  y[k] = x[k2];
}
```

The in-place version is

```
template <typename Type>
void unzip_rev(Type *x, ulong n)
// n must be a power of two
{
    const ulong nh = n/2;
    revbin_permute(x, n);
    revbin_permute(x, nh); revbin_permute(x+nh, nh);
    reverse(x+nh, nh);
}
```

The given permutation is used in an algorithm where the cosine transform is computed using the Hartley transform, see section 24.11 on page 500.

We write Z and Z^{-1} for the zip permutation and its inverse, $\overline{Z}$ and $\overline{Z}^{-1}$ for the reversed zip permutation and its inverse, and R for the revbin permutation. Then the following relations hold:

$$Z = R\,Z^{-1}\,R = \overline{Z}\,Z^{-1}\,\overline{Z} \tag{2.6-1a}$$
$$\overline{Z} = R\,\overline{Z}^{-1}\,R = Z\,\overline{Z}^{-1}\,Z \tag{2.6-1b}$$
$$Z^{-1} = R\,Z\,R = \overline{Z}^{-1}\,Z\,\overline{Z}^{-1} \tag{2.6-1c}$$
$$\overline{Z}^{-1} = R\,\overline{Z}\,R = Z^{-1}\,\overline{Z}\,Z^{-1} \tag{2.6-1d}$$

2.7 The XOR permutation

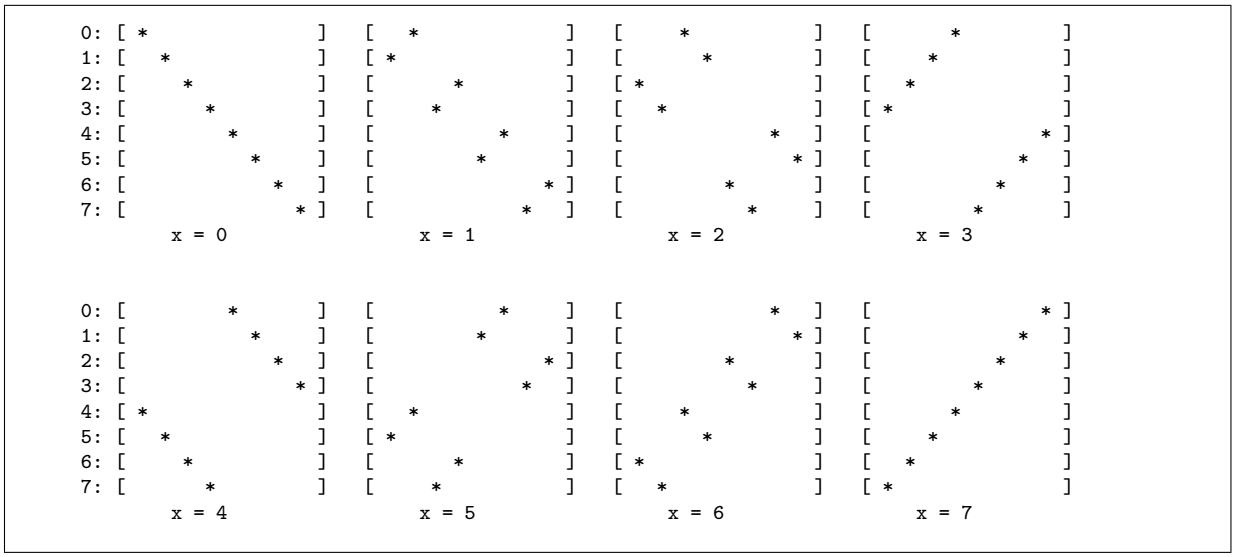


Figure 2.7-A: Permutation matrices of the XOR permutation for length 8 with parameter $x = 0 \ldots 7$. Compare to the table for the dyadic convolution shown in figure 22.7-A on page 446.

The *XOR permutation* may be explained most simply by its trivial implementation: [FXT: `xor_permute()` in perm/xorpermute.h]:

```
template <typename Type>
void xor_permute(Type *f, ulong n, ulong x)
{
    if ( 0==x )  return;
    for (ulong k=0; k<n; ++k)
    {
        ulong r = k^x;
        if ( r>k )  swap2(f[r], f[k]);
    }
}
```

The XOR permutation is evidently self-inverse. The array length n must be divisible by the smallest power of two that is greater than x: for example, n must be even if $x = 1$, n must be divisible by four if $x = 2$ or $x = 3$. With n a power of two and $x < n$ one is on the safe side.

The XOR permutation contains a few other permutations as important special cases (for simplicity assume that the array length n is a power of two): when the third argument x equals $n - 1$ then the permutation is the reversion, with $x = 1$ neighboring even and odd indexed elements are swapped, with $x = n/2$ the upper and the lower half of the array are swapped.

One has

$$X_a\, X_b \;=\; X_b\, X_a \;=\; X_c \quad \text{where} \quad c = a \text{ XOR } b \tag{2.7-1}$$

For the special case $a = b$ the relation expresses the self-inverse property as X_0 is the identity. The XOR permutation often occurs in relations between other permutations where we will use the symbol X_x, the subscript denoting the third argument.

2.8 The Gray code permutation

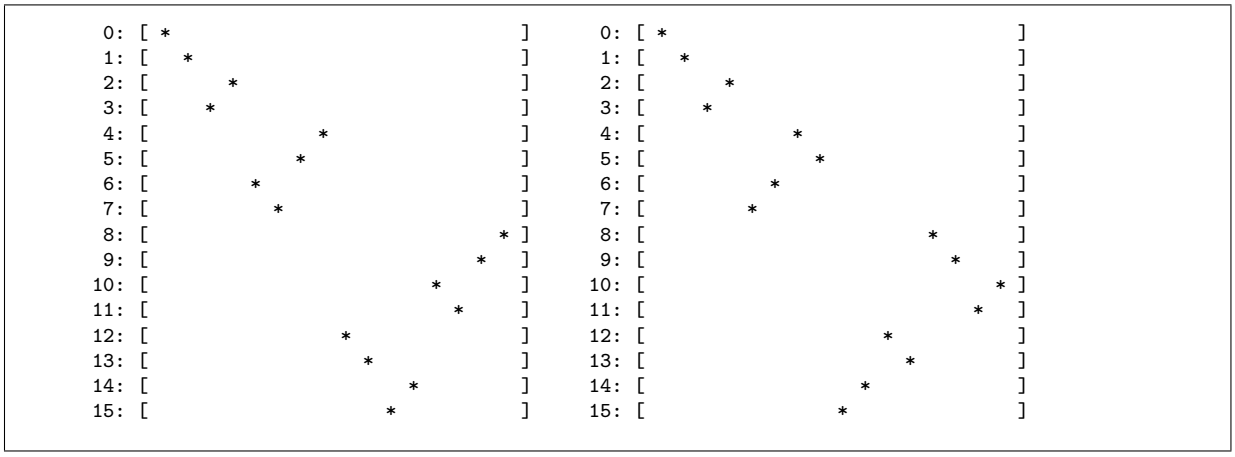


Figure 2.8-A: Permutation matrices of the Gray code permutation (left) and its inverse (right).

The *Gray code permutation* (or simply *Gray permutation*) reorders (length-2^n) arrays according to the binary Gray code described in section 1.15 on page 36. A routine for the permutation is [FXT: perm/graypermute.h]:

```
template <typename Type>
inline void gray_permute(const Type *f, Type * restrict g, ulong n)
// Put Gray permutation of f[] to g[], i.e. g[gray_code(k)] == f[k]
{
    for (ulong k=0; k<n; ++k)  g[gray_code(k)] = f[k];
}
```

Its inverse is

```
template <typename Type>
inline void inverse_gray_permute(const Type *f, Type * restrict g, ulong n)
// Put inverse Gray permutation of f[] to g[], i.e. g[k] == f[gray_code(k)]
// (same as: g[inverse_gray_code(k)] == f[k])
{
    for (ulong k=0; k<n; ++k)  g[k] = f[gray_code(k)];
}
```

We again use calls to `gray_code()` because they are cheaper than the computation of `inverse_gray_code()`.

We now give in-place versions of the above routines that offer very good performance. It is necessary to identify the cycle leaders (see section 2.11 on page 104) of the permutation and find an efficient way to generate them.

2.8.1 Cycles of the permutation

It is instructive to study the complementary masks that occur for cycles (see section 2.11 on page 104) of different lengths. The cycles of the Gray code permutation for length 128 are shown in figure 2.8-B. No structure is immediately visible.

```
     0:  (  2,  3) #=2
     1:  (  4,  7,  5,  6) #=4
     2:  (  8, 15, 10, 12) #=4
     3:  (  9, 14, 11, 13) #=4
     4:  ( 16, 31, 21, 25, 17, 30, 20, 24) #=8
     5:  ( 18, 28, 23, 26, 19, 29, 22, 27) #=8
     6:  ( 32, 63, 42, 51, 34, 60, 40, 48) #=8
     7:  ( 33, 62, 43, 50, 35, 61, 41, 49) #=8
     8:  ( 36, 56, 47, 53, 38, 59, 45, 54) #=8
     9:  ( 37, 57, 46, 52, 39, 58, 44, 55) #=8
    10:  ( 64,127, 85,102, 68,120, 80, 96) #=8
    11:  ( 65,126, 84,103, 69,121, 81, 97) #=8
    12:  ( 66,124, 87,101, 70,123, 82, 99) #=8
    13:  ( 67,125, 86,100, 71,122, 83, 98) #=8
    14:  ( 72,112, 95,106, 76,119, 90,108) #=8
    15:  ( 73,113, 94,107, 77,118, 91,109) #=8
    16:  ( 74,115, 93,105, 78,116, 88,111) #=8
    17:  ( 75,114, 92,104, 79,117, 89,110) #=8
    126 elements in  18 nontrivial cycles.
    cycle lengths:   2 ...   8
   2 fixed points: [0. 1]
```

Figure 2.8-B: Cycles of the Gray code permutation of length 128.

However, one can generate the cycle maxima as follows: for each range $2^k \ldots 2^{k+1}-1$ generate a bit-mask z that is obtained from the $k+1$ leftmost bits of the infinite word that has bits set at positions 0, 1, 2, 4, 8, ..., 2^i, ...:

```
[11101000100000001000000000000000001000 ...]
```

An example: for $k=6$ we have $z=$`[1110100]`. Then take v to be $k+1$ leftmost bits of the complement, $v=$`[0001011]` in our example. Now the set of words $c=z+s$ where s is a subset of v contains exactly one element of each cycle in the range $2^k \ldots 2^{k+1} = 64 \ldots 127$:

```
    117 =  .111.1.1
    118 =  .111.11.
    119 =  .111.111
    124 =  .11111..
    125 =  .11111.1
    126 =  .111111.
    127 =  .1111111
    116 =  .111.1..
      maxima = z XOR subsets(v)  where
      z =  .111.1..
      v =  ....1.11
```

The words obtained are actually the cycle maxima. The list can be generated with the program [FXT: perm/permgray-leaders-demo.cc] which uses [FXT: `class gray_cycle_leaders` in comb/gray-cycle-leaders.h]:

```
class gray_cycle_leaders
// Generate cycle leaders for Gray code permutation
// where highest bit is at position ldn.
{
public:
    bit_subset b_;
    ulong za_;  // mask for cycle maxima
    ulong zi_;  // mask for cycle minima
    ulong len_; // cycle length
    ulong num_; // number of cycles

public:

    gray_cycle_leaders(ulong ldn)  // 0<=ldn<BITS_PER_LONG
        : b_(0)
    {
        init(ldn);
    }

    ~gray_cycle_leaders() {;}

    void init(ulong ldn)
    {
        za_ = 1;
        ulong cz = 0;   // ~z
```

```
        len_ = 1;
        num_ = 1;
        for (ulong ldm=1; ldm<=ldn; ++ldm)
        {
            za_ <<= 1;
            cz <<= 1;
            if ( is_pow_of_2(ldm) )
            {
                ++za_;
                len_ <<= 1;
            }
            else
            {
                ++cz;
                num_ <<= 1;
            }
        }
        zi_ = 1UL << ldn;
        b_.first(cz);
    }
    ulong current_max()  const  { return b_.current() | za_; }
    ulong current_min()  const  { return b_.current() | zi_; }
    bool next()
    {
        return ( 0!=b_.next() );
    }
    ulong num_cycles()  const  { return num_; }
    ulong cycle_length()  const  { return len_; }
};
```

The implementation uses the bit-subset class described in section 1.26 on page 63.

2.8.2 In-place routines

For the in-place versions of the permutation routines are obtained by inlining the generation of the cycle leaders. The forward version is [FXT: perm/graypermute.h]:

```
template <typename Type>
void gray_permute(Type *f, ulong n)
{
    ulong z = 1; // mask for cycle maxima
    ulong v = 0; // ~z
    ulong cl = 1;  // cycle length
    for (ulong ldm=1, m=2;  m<n;  ++ldm, m<<=1)
    {
        z <<= 1;
        v <<= 1;
        if ( is_pow_of_2(ldm) )
        {
            ++z;
            cl <<= 1;
        }
        else  ++v;
        bit_subset b(v);
        do
        {
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                Type tt = f[g];
                f[g] = t;
                t = tt;
                g = gray_code(g);
            }
            f[g] = t;
            // --- end (do cycle) ---
        }
```

```
        while ( b.current() );
    }
}
```

The function `is_pow_of_2()` is described in section 1.7 on page 18. The inverse routine differs only in the block that processes the cycles:

```
template <typename Type>
void inverse_gray_permute(Type *f, ulong n)
{
    [--snip--]
            // --- do cycle: ---
            ulong i = z | b.next();  // start of cycle
            Type t = f[i];           // save start value
            ulong g = gray_code(i);  // next in cycle
            for (ulong k=cl-1; k!=0; --k)
            {
                f[i] = f[g];
                i = g;
                g = gray_code(i);
            }
            f[i] = t;
            // --- end (do cycle) ---
    [--snip--]
}
```

2.8.3 Performance of the routines

How fast is the Gray code permutation? We use the convention that the speed of the trivial (and completely cache-friendly, therefore running at memory bandwidth) `reverse()` is 1.0, our hereby declared time unit for comparison [FXT: perm/reverse.h]. A little benchmark gives, for large (16 MB) arrays:

```
arg 1: 21 == ldn  [Using 2**ldn elements]  default=21
arg 2: 10 == rep  [Number of repetitions]  default=10
Memsize = 16384 kiloByte  ==  2097152 doubles

                reverse(f,n);    dt=  0.0103524    MB/s= 1546   rel=          1
          revbin_permute(f,n);    dt=  0.0674235    MB/s=  237   rel=    6.51282
         revbin_permute0(f,n);    dt=   0.061507    MB/s=  260   rel=    5.94131
            gray_permute(f,n);    dt=  0.0155019    MB/s= 1032   rel=    1.49742
    inverse_gray_permute(f,n);    dt=  0.0150641    MB/s= 1062   rel=    1.45512
                reverse(f,n);    dt=  0.0104008    MB/s= 1538   rel=    1.00467
```

We timed `reverse()` twice to get an impression how much we can trust the observed numbers.

While the revbin permutation takes about 6 units (due to its memory access pattern that is very problematic with respect to cache usage) the Gray code permutation needs only 1.50 units. The difference gets bigger for machines with relatively (to the CPU) slow memory. The Gray code permutation can be used to speed up fast transforms of large lengths a power of two, notably the Walsh transform, see chapter 22 on page 429.

The bandwidth of the `reverse()` is about 1500 MB/sec which should be compared to the output of a memory testing program, revealing that it actually runs at about the bandwidth of copying via a simple loop using pointers to `double`s:

```
    avg:   16777216  [ 0]"memcpy"                   2522.084 MB/s
    avg:   16777216  [ 1]"char *"                    471.873 MB/s
    avg:   16777216  [ 2]"short *"                   711.853 MB/s
    avg:   16777216  [ 3]"int *"                     956.682 MB/s
    avg:   16777216  [ 4]"long *"                   1514.360 MB/s
    avg:   16777216  [ 5]"long * (4x unrolled)"     1330.786 MB/s
    avg:   16777216  [ 6]"int64 *"                  1329.902 MB/s
    avg:   16777216  [ 7]"double *"                 1329.507 MB/s  // <--=
    avg:   16777216  [ 8]"double * (4x unrolled)"   1325.437 MB/s
```

The relative speeds are quite different for small arrays. Using a size of 16 kB (2048 `double`s) we obtain

```
arg 1: 11 == ldn  [Using 2**ldn elements]  default=21
arg 2: 100000 == rep  [Number of repetitions]  default=512
```

```
Memsize = 16 kiloByte  ==  2048 doubles
                reverse(f,n);    dt=1.88726e-06    MB/s= 8279   rel=          1
          revbin_permute(f,n);    dt=3.22166e-06    MB/s= 4850   rel=    1.70706
         revbin_permute0(f,n);    dt=2.69212e-06    MB/s= 5804   rel=    1.42647
            gray_permute(f,n);    dt=4.75155e-06    MB/s= 3288   rel=    2.51769
    inverse_gray_permute(f,n);    dt=3.69237e-06    MB/s= 4232   rel=    1.95647
                reverse(f,n);    dt=1.88833e-06    MB/s= 8275   rel=    1.00057
```

due to the small size, the cache problems are gone.

The memory benchmark gives for that size

```
    avg:      16384  [ 0]"memcpy"                   3290.353 MB/s
    avg:      16384  [ 1]"char *"                    572.922 MB/s
    avg:      16384  [ 2]"short *"                   973.552 MB/s
    avg:      16384  [ 3]"int *"                    1495.920 MB/s
    avg:      16384  [ 4]"long *"                   3560.506 MB/s
    avg:      16384  [ 5]"long * (4x unrolled)"     3220.792 MB/s
    avg:      16384  [ 6]"int64 *"                  2498.137 MB/s
    avg:      16384  [ 7]"double *"                 2498.285 MB/s  // <--=
    avg:      16384  [ 8]"double * (4x unrolled)"   3219.784 MB/s
```

2.9 The reversed Gray code permutation

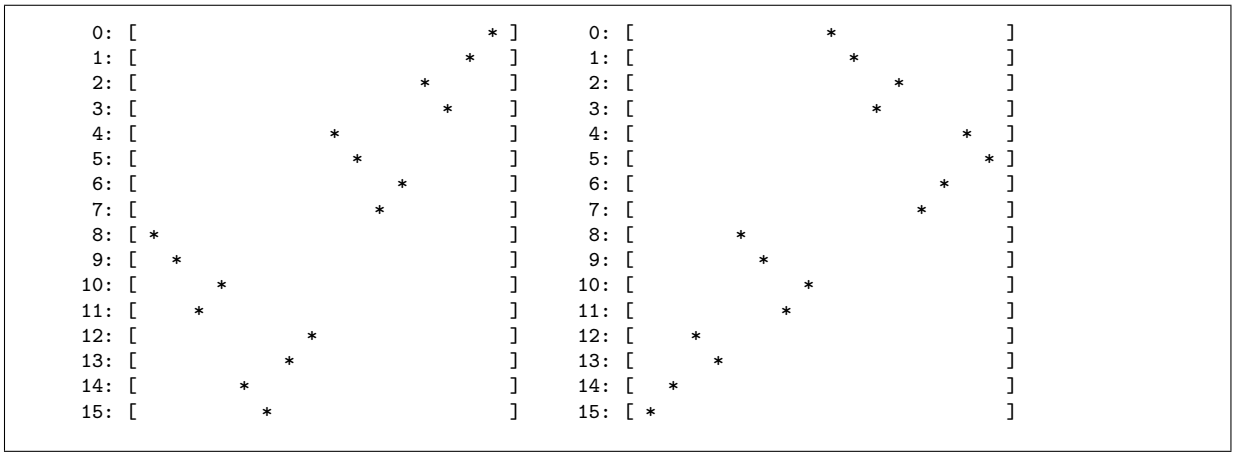


Figure 2.9-A: Permutation matrices of the reversed Gray code permutation (left) and its inverse (right).

If the length-n array is permuted in the way the upper half of the length-$2n$ array would be permuted by `gray_permute()` then all cycles are of the same length. The resulting permutation is equivalent to the *reversed Gray code permutation*:

```
template <typename Type>
inline void gray_rev_permute(const Type *f, Type * restrict g, ulong n)
// gray_rev_permute() =^=
//  { reverse(); gray_permute(); }
{
    for (ulong k=0, m=n-1;  k<n;  ++k, --m)  g[gray_code(m)] = f[k];
}
```

The routine, its inverse and in-place versions are given in [FXT: perm/grayrevpermute.h].

All cycles have the same length, `gray_rev_permute(f, 64)` gives:

```
    0:  (  0, 63, 21, 38,  4, 56, 16, 32) #=8
    1:  (  1, 62, 20, 39,  5, 57, 17, 33) #=8
    2:  (  2, 60, 23, 37,  6, 59, 18, 35) #=8
    3:  (  3, 61, 22, 36,  7, 58, 19, 34) #=8
    4:  (  8, 48, 31, 42, 12, 55, 26, 44) #=8
    5:  (  9, 49, 30, 43, 13, 54, 27, 45) #=8
    6:  ( 10, 51, 29, 41, 14, 52, 24, 47) #=8
```

```
7:  ( 11, 50, 28, 40, 15, 53, 25, 46) #=8
64 elements in   8 nontrivial cycles.
cycle length is == 8
No fixed points.
```

If 64 is added to the indices then the cycles in the upper half of the array as in `gray_permute(f, 128)` are reproduced (by construction).

Let G denote the Gray code permutation, $\overline{G}$ the reversed Gray code permutation. Symbolically one can write

$$G(n) = \{\dots, \overline{G}(n/8), \overline{G}(n/4), \overline{G}(n/2)\} \quad (2.9\text{-}1a)$$
$$G^{-1}(n) = \{\dots, \overline{G}^{-1}(n/8), \overline{G}^{-1}(n/4), \overline{G}^{-1}(n/2)\} \quad (2.9\text{-}1b)$$

Now let r be the reversion and h the permutation that swaps the upper and the lower half of an array. Then

$$\overline{G} = G\,r = h\,G \quad (2.9\text{-}2a)$$
$$\overline{G}^{-1} = r\,G^{-1} \quad (2.9\text{-}2b)$$
$$\overline{G}^{-1}\,G = G^{-1}\,\overline{G} = r = X_{n-1} \quad (2.9\text{-}2c)$$
$$G\,\overline{G}^{-1} = \overline{G}\,G^{-1} = h = X_{n/2} \quad (2.9\text{-}2d)$$

Throughout it is assumed that the array length n is a power of two.

2.10 Decomposing permutations *

In this section we will see some algorithms that use a certain type of decomposition (factorization in term of matrices) of some of the permutations we have studied so far. The resulting algorithms involve proportional $n \cdot \log(n)$ computations for length-n arrays. This might seem to render the schemes worthless as one can always obtain a permutation with work proportional to n. There are, however, situations where one can use the algorithms advantageously. Firstly, with bit manipulations, where the whole binary words are modified in one or a few statements. The corresponding algorithms are therefore only proportional $\log(n)$. Secondly, when the algorithm can be used implicitly in order to integrate the permutation in a fast transform. The work can sometimes be reduced to zero in that case.

Array reversal

The most simple example might be the reversion via

```
template <typename Type>
void perm1(Type *f, ulong n)
// From shorter to longer sub-arrays.
{
    for (ulong k=2; k<=n; k*=2)
    {
        for (ulong j=0; j<n; j+=k)  func(f+j, k);
    }
}
```

where `func()` swaps the upper and lower half of an array:

```
template <typename Type>
void func(Type *f, ulong ldn)  { swap(f, f+n/2, n/2); }
```

This idea has been exploited in section 1.13 on page 30 in order to obtain a bit-reversal routine. The reversal is a self-inverse operation. Therefore one can alternatively execute the steps in reverse order and still get the same permutation:

```
template <typename Type>
void perm2(Type *f, ulong n)
// From longer to shorter sub-arrays.
{
    for (ulong k=n; k>=2; k/=2) // k == n, n/2, n/4, ... , 4, 2
    {
        for (ulong j=0; j<n; j+=k)  func(f+j, k);
    }
}
```

Note that `func()` in turn can be obtained via

```
reverse(f, n);
reverse(f, n/2);  reverse(f+n/2, n/2);
```

or the same statements in reversed order.

Gray code permutation

Let us try a less obvious example, use `perm1()` with `func()` defined to swap the third and fourth quarter:

```
swap(f+n/2, f+n/2+n/4, n/4);  // quarters: [0,1,2,3]-->[0,1,3,2]
```

The resulting permutation is the Gray permutation. The other way round (using `perm2()`) one obtains the inverse Gray permutation.

Using `func()` defined as

```
reverse(f+n/2, n/2);
```

one gets the Gray permutation through `perm2()`, its inverse through `perm1()`. This idea has been used in the core-routine for the sequency-ordered Walsh transform described in section 22.9 on page 448. The work for the Gray permutation has been completely vaporized there.

Note that the routine that swaps the halves of the upper half array could be obtained as either of

```
inverse_gray_permute(f, n);
gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
```

or

```
inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
gray_permute(f, n);
```

Similarly, the routine that reverses the upper half can be obtained as either of

```
gray_permute(f, n/2);     gray_permute(f+n/2, n/2);
inverse_gray_permute(f, n);
```

or

```
gray_permute(f, n);
inverse_gray_permute(f, n/2);  inverse_gray_permute(f+n/2, n/2);
```

The corresponding routines to Gray-permute the bits of a binary word are given in [FXT: bits/bitgraypermute.h].

Zip and revbin permutation

Using `func()` defined to swap the second and third quarter

```
swap(f+n/4, f+n/2, n/4);  // quarters: [0,1,2,3]-->[0,2,1,3]
```

Then with `perm2()` one gets the zip permutation, `perm1()` gives the inverse. This idea has been used for the bit-wise zip shown in section 1.14 on page 35.

Using `func()` to cycle the second, third and fourth quarter:

```
// quarters: [0,1,2,3]-->[0,2,3,1]
```

which was obtained using

```
    zip_rev(f, n);
    unzip_rev(f, n/2);  unzip_rev(f+n/2, n/2);
```

both the reversed zip permutation and its inverse can be computed in the now hopefully obvious way.

The revbin permutation can be generated through the zip permutation or its inverse. However, the zip permutation is the more complicate one, so absorbing the revbin permutation into fast transforms does not seem to be easy. The other way round it makes more sense:

```
    revbin_permute(f, n/2);  revbin_permute(f+n/2, n/2);
    revbin_permute(f, n);
```

Is a convenient (though not the most effective) way to compute the zip permutation.

Clearly, the idea presented here is in analogy with the decomposition of linear transforms. Finally the permutations are (very simple forms of) linear transforms. See also section 1.32 on page 81.

2.11 General permutations and their operations

So far we treated special permutations that occurred as part of other algorithms. It is instructive to study permutations in general with the operations (as composition and inversion) on them.

2.11.1 Basic definitions and operations

A straightforward way to represent a permutation is to consider the array of indices that for the original (unpermuted) data would be the length-n *canonical sequence* 0, 1, 2, ..., $n-1$. The mentioned trivial sequence represents the 'do-nothing' permutation or *identity*. The concept is best described by the routine that *applies* a given permutation x on an array of data f: after the routine has finished the array g will contain the elements of f reordered according to x [FXT: `apply_permutation()` in perm/permapply.h]:

```
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the permutation x[] on f[]
// i.e. set g[k] <-- f[x[k]]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x[k]];
}
```

An example using strings (arrays of characters): the permutation represented by

$x =$`[ 7 6 3 2 5 1 0 4 ]`

and the input data

$f =$`[ A B a d C a f e ]` would produce

$g =$`[ e f d a a B A C ]`

Routines that test various properties of permutations are given in [FXT: perm/permq.h]. To check whether a given permutation is the identity is trivial:

```
bool is_identity(const ulong *f, ulong n)
// Return whether f[] is the identical permutation,
// i.e. whether f[k]==k for all k= 0...n-1
{
    for (ulong k=0; k<n; ++k)  if ( f[k] != k )  return false;
    return true;
}
```

A fixed point of a permutation is an index where the element is not moved:

```
ulong count_fixed_points(const ulong *f, ulong n)
// Return number of fixed points in f[]
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  ++ct;
    return ct;
}
```

A *derangement* is a permutation that has no fixed points. To check whether a permutation is a derangement of identity use:

```
bool is_derangement(const ulong *f, ulong n)
// Return whether f[] is a derangement of identity,
// i.e. whether f[k]!=k for all k
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == k )  return false;
    return true;
}
```

Whether two arrays are mutual derangements can be determined by:

```
bool is_derangement(const ulong *f, const ulong *g, ulong n)
// Return whether f[] is a derangement of g[],
// i.e. whether f[k]!=g[k] for all k
{
    for (ulong k=0; k<n; ++k)  if ( f[k] == g[k] )  return false;
    return true;
}
```

To check whether a given array really describes a valid permutation one has to verify that each index in the valid range appears exactly once. The `bitarray` class described in section 4.6 on page 152 allows us to do the job without modification of the input:

```
bool
is_valid_permutation(const ulong *f, ulong n, bitarray *bp/*=0*/)
// Return whether all values 0...n-1 appear exactly once,
// i.e. whether f represents a permutation of [0,1,...,n-1].
{
    // check whether any element is out of range:
    for (ulong k=0; k<n; ++k)  if ( f[k]>=n )  return false;

    // check whether values are unique:
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong k;
    for (k=0; k<n; ++k)
    {
        if ( tp->test_set(f[k]) )  break;
    }

    if ( 0==bp )  delete tp;

    return  (k==n);
}
```

We note two rather trivial operations for permutation, computing the *complement* [FXT: perm/permcomplement.h]

```
inline void make_complement(const ulong *f, ulong *g, ulong n)
// Set (as permutation) g to the complement of f.
// Can have f==g.
{
    for (ulong k=0; k<n; ++k)  g[k] = n - 1 - f[k];
}
```

and computing the reversal [FXT: perm/reverse.h]

```
template <typename Type>
inline void reverse(Type *f, ulong n)
// Reverse order of array f.
{
    for (ulong k=0, i=n-1;  k<i;  ++k, --i)  swap2(f[k], f[i]);
}
```

2.11.2 Compositions of permutations

One can apply several permutations to an array, one by one. The resulting permutation is called the *composition* of the applied permutations. The routines are given in [FXT: perm/permq.cc]. As an example, the check whether some permutation g is equal to f applied twice, or f *squared*, use:

```
bool is_square(const ulong *f, const ulong *g, ulong n)
// Return whether f * f == g  as a permutation
{
    for (ulong k=0; k<n; ++k)  if ( g[k] != f[f[k]] )  return false;
    return true;
}
```

Note that in general $f \cdot g \neq g \cdot f$ for $f \neq g$, the operation of composition is not commutative.

A permutation f is said to be the *inverse* of another permutation g if it undoes its effect, that is $f \cdot g = \mathrm{id}$:

```
bool is_inverse(const ulong *f, const ulong *g, ulong n)
// Return whether f[] is the inverse of g[]
{
    for (ulong k=0; k<n; ++k)  if ( f[g[k]] != k )  return false;
    return true;
}
```

One has $g \cdot f = f \cdot g = \mathrm{id}$, in a group the left-inverse is equal to the right-inverse and we can simply call g 'the inverse' of f.

A permutation that is its own inverse (like the revbin permutation) is called an *involution*. Checking that is easy:

```
bool is_involution(const ulong *f, ulong n)
// Return whether max cycle length is <= 2,
// i.e. whether f * f = id.
{
    for (ulong k=0; k<n; ++k)  if ( f[f[k]] != k )  return false;
    return true;
}
```

The following routine computes the inverse of a given permutation [FXT: perm/perminvert.cc]:

```
void make_inverse(const ulong *f, ulong * restrict g, ulong n)
// Set (as permutation) g to the inverse of f
{
    for (ulong k=0; k<n; ++k)  g[f[k]] = k;
}
```

2.11.3 Representation as disjoint cycles

If one wants to do the operation in-place a little bit of thought is required. The idea underlying all subsequent routines working in-place is that every permutation entirely consists of disjoint cycles. A *cycle* of a permutation is a subset of the indices that is rotated (by one) by the permutation. The term *disjoint* means that the cycles do not 'cross' each other. While this observation is pretty trivial it allows us to do many operations by following the cycles of the permutation, one by one, and doing the necessary operation on each of them. As an example consider the following permutation of an array originally consisting of the (canonical) sequence 0, 1, ..., 15. Extra spaces are inserted for readability:

```
[ 0, 1, 3, 2,   7, 6, 4, 5,   15, 14, 12, 13,   8, 9, 11, 10 ]
```

There are two fixed points (0 and 1) and these cycles:

```
( 2 <--  3 )
( 4 <--  7 <--  5 <--  6 )
( 8 <-- 15 <-- 10 <-- 12 )
( 9 <-- 14 <-- 11 <-- 13 )
```

The cycles do 'wrap around', for example, the initial 4 of the second cycle goes to position 6, the last element of the second cycle.

Note that the inverse permutation could formally be described by reversing every arrow in each cycle:

```
( 2 -->  3 )
( 4 -->  7 -->  5 -->  6 )
( 8 --> 15 --> 10 --> 12 )
( 9 --> 14 --> 11 --> 13 )
```

Equivalently, one can reverse the order of the elements in each cycle:

```
    ( 3 <--  2 )
    ( 6 <--  5 <--  7 <--  4 )
    (12 <-- 10 <-- 15 <--  8 )
    (13 <-- 11 <-- 14 <--  9 )
```

If we begin each cycle with its smallest element the inverse permutation is written as

```
    ( 2 <--  3 )
    ( 4 <--  6 <--  5 <--  7 )
    ( 8 <-- 12 <-- 10 <-- 15 )
    ( 9 <-- 13 <-- 11 <-- 14 )
```

This form is obtained by reversing all alements except the first in each cycle of the (forward) permutation. The last three sets of cycles all describe the same permutation:

```
    [ 0, 1, 3, 2,   6, 7, 5, 4,   12, 13, 15, 14,   10, 11, 9, 8 ]
```

The cycles above were printed with [FXT: `print_cycles()` in perm/printcycles.cc]

```
ulong print_cycles(const ulong *f, ulong n, bitarray *bp=0)
// Print the cycles of the permutation.
// Return number of fixed points.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    ulong ct = 0;  // # of fixed points
    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // follow a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        if ( g==i )  // fixed point ?
        {
            ++ct;
            continue;
        }

        cout << "(" << setw(3) << i;
        while ( 0==(tp->test_set(g)) )
        {
            cout << " <-- " << setw(3) << g;
            g = f[g];
        }
        cout << " )" << endl;
    }

    if ( 0==bp )  delete tp;

    return  ct;
}
```

The bit-array (see section 4.6 on page 152 for the implementation) is used to keep track of the elements already processed.

A utility class to compute the decomposition of a permutation into cycles is [FXT: `class cycles` in perm/cycles.h]. A program that shows its usage is [FXT: perm/cycles-demo.cc], it prints the cycles

```
Using: gray_permute(y, n)
Computing cycles:
  0:  (  2,  3) #=2
  1:  (  4,  7,  5,  6) #=4
  2:  (  8, 15, 10, 12) #=4
  3:  (  9, 14, 11, 13) #=4
  14 elements in   4 nontrivial cycles.
 cycle lengths:   2 ...   4
number of fixed points = 2
```

and code for a permutation of given size

```
template <typename Type>
inline void foo_perm_16(Type *f)
// unrolled version for length 16
{
```

```
  swap2(f[2], f[3]);
  { Type t=f[4]; f[4]=f[7]; f[7]=f[5]; f[5]=f[6]; f[6]=t; }
  { Type t=f[8]; f[8]=f[15]; f[15]=f[10]; f[10]=f[12]; f[12]=t; }
  { Type t=f[9]; f[9]=f[14]; f[14]=f[11]; f[11]=f[13]; f[13]=t; }
}
```

2.11.4 Cyclic permutations

A permutation consisting of exactly one cycle is called *cyclic*. Whether a given permutation has this property can be tested with [FXT: `is_cyclic()` in perm/permq.cc]:

```
bool
is_cyclic(const ulong *f, ulong n)
// Return whether permutation is exactly one cycle.
{
    if ( n<=1 )  return true;
    ulong k = 0,  e = 0;
    do  { e=f[e]; ++k; }  while ( e!=0 );
    return  (k==n);
}
```

The method used is to follow the cycle starting at position zero and counting how long it is. The permutation is cyclic exactly if the length found equals the array length. There are $(n-1)!$ cyclic permutations of n elements.

2.11.5 Sign and parity of a permutation

Every permutation can be written as a composition of *transpositions* (cycles of length two). This composition is not unique, but its number modulo two is unique. The *sign* of a permutation is defined to be $+1$ the number is even and -1 if the number is odd. The minimal number of transpositions whose composition give a cycle of length l is $l-1$. So the minimal number of transpositions for a permutation consisting of k cycles where the length of the j-th cycle is l_j equals $\sum_{j=1}^{k} l_j - 1 = (\sum_{j=1}^{k} l_j) - k$. The sign corresponds to the homomorphic mapping into the group of the elements $+1$ and -1 with multiplication as group operation. If we count the transpositions modulo two (corresponding to the mapping into the additive group modulo two) we obtain what may be called the *parity* of a permutation.

2.11.6 Inverse and square, in-place

For the computation of the inverse we have to reverse each cycle [FXT: perm/perminvert.cc]:

```
void make_inverse(ulong *f, ulong n, bitarray *bp/*=0*/)
// Set (as permutation) f to its own inverse.
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // invert a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        while ( 0==(tp->test_set(g)) )
        {
            ulong t = f[g];
            f[g] = i;
            i = g;
            g = t;
        }
```

```
        f[g] = i;
    }
    if ( 0==bp )  delete tp;
}
```

The extra array of tag bits can be avoided by using the highest bit of each word as tag bit. The scheme would fail if any word of the permutation array had the highest bit set. However, on byte-addressable machines such an array will not fit into memory at all (for word sizes of 16 or more bits). To keep the code similar to the version using the bit-array we define

```
static const ulong s1 = 1UL << (BITS_PER_LONG - 1);  // highest bit is tag bit
static const ulong s0 = ~s1;  // all bits but tag bit

static inline void SET(ulong *f, ulong k)  { f[k&s0] |= s1; }
static inline void CLEAR(ulong *f, ulong k)  { f[k&s0] &= s0; }
static inline bool TEST(ulong *f, ulong k)  { return (0!=(f[k&s0]&s1)); }
```

Note that we have to mask out the tag-bit when using the value 'k' as index. The routine can then be implemented as

```
void
make_inverse(ulong *f, ulong n)
// Set (as permutation) f to its own inverse.
// In-place version using highest bits of array as tag-bits.
{
    for (ulong k=0; k<n; ++k)
    {
        if ( TEST(f, k) ) { CLEAR(f, k);  continue; }  // already processed
        SET(f, k);

        // invert a cycle:
        ulong i = k;
        ulong g = f[i];  // next index
        while ( 0==TEST(f, g) )
        {
            ulong t = f[g];
            f[g] = i;
            SET(f, g);
            i = g;
            g = t;
        }
        f[g] = i;

        CLEAR(f, k);  // leave no tag bits set
    }
}
```

The extra `CLEAR()` statement at the end removes the tag-bit of the cycle minima. Its effect is that no tag-bits are set after routine has finished. The routine has about the same performance as the bit-array version. For the routine [FXT: perm/permcompose.cc]

```
void make_square(const ulong *f, ulong * restrict g, ulong n)
// Set (as permutation) g = f * f
{
    for (ulong k=0; k<n; ++k)  g[k] = f[f[k]];
}
```

we obtain the following in-place version:

```
void make_square(ulong *f, ulong n, bitarray *bp/*=0*/)
// Set (as permutation) f = f * f
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // square a cycle:
        ulong i = k;
        ulong t = f[i];  // save
        ulong g = f[i];  // next index
```

```
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = f[g];
        }
        f[i] = t;
    }
    if ( 0==bp )  delete tp;
}
```

2.11.7 Powers of a permutation

The e-th power of a permutation f is computed (and returned in g) by a version of the binary exponentiation algorithm described in section 27.6 on page 537 [FXT: perm/permcompose.cc]:

```
void
power(const ulong *f, ulong * restrict g, ulong n, long e,
      ulong * restrict t/*=0*/)
// Set (as permutation) g = f ** e
{
    if ( e==0 )
    {
        for (ulong k=0; k<n; ++k)  g[k] = k;
        return;
    }
    if ( e==1 )
    {
        copy(f, g, n);
        return;
    }
    if ( e==-1 )
    {
        make_inverse(f, g, n);
        return;
    }
    // here:  abs(e) > 1
    ulong x = e>0 ? e : -e;
    if ( is_pow_of_2(x) )  // special case x==2^n
    {
        make_square(f, g, n);
        while ( x>2 )  { make_square(g, n);  x /= 2; }
    }
    else
    {
        ulong *tt = t;
        if ( 0==t )  { tt = new ulong[n]; }
        copy(f, tt, n);
        int firstq = 1;
        while ( 1 )
        {
            if ( x&1 )  // odd
            {
                if ( firstq )  // avoid multiplication by 1
                {
                    copy(tt, g, n);
                    firstq = 0;
                }
                else  compose(tt, g, n);
                if ( x==1 )   goto dort;
            }
            make_square(tt, n);
            x /= 2;
        }
    dort:
        if ( 0==t )  delete [] tt;
    }
    if ( e<0 )  make_inverse(g, n);
```

```
}
```

2.11.8 Applying permutations to data, in-place

The in-place analogue for the routine

```
template <typename Type>
void apply_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the permutation x[] to the array f[]
// i.e. set g[k] <-- f[x[k]]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x[k]];
}
```

is [FXT: perm/permapply.h]:

```
template <typename Type>
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// Apply the permutation x[] to the array f[]
// i.e. set f[k] <-- f[x[k]]  \forall k
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);

        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. inverse_gray_permute()
        {
            f[i] = f[g];
            i = g;
            g = x[i];
        }
        f[i] = t;
        // --- end (do cycle) ---
    }

    if ( 0==bp )  delete tp;
}
```

To apply the inverse of a permutation without actually inverting the permutation itself use

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, const Type *f, Type * restrict g, ulong n)
// Apply the inverse permutation of x[] to the array f[],
// i.e. set g[x[k]] <-- f[k]  \forall k
{
    for (ulong k=0; k<n; ++k)  g[x[k]] = f[k];
}
```

The in-place version is

```
template <typename Type>
void apply_inverse_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
// Apply the inverse permutation of x[] to the array f[]
// i.e. set f[x[k]] <-- f[k]  \forall k
// In-place version.
{
    bitarray *tp = bp;
    if ( 0==bp )  tp = new bitarray(n);  // tags
    tp->clear_all();

    for (ulong k=0; k<n; ++k)
    {
        if ( tp->test_clear(k) )  continue;  // already processed
        tp->set(k);
```

```
        // --- do cycle: ---
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];
        while ( 0==(tp->test_set(g)) )  // cf. gray_permute()
        {
            Type tt = f[g];
            f[g] = t;
            t = tt;
            g = x[g];
        }
        f[g] = t;
        // --- end (do cycle) ---
    }
    if ( 0==bp )  delete tp;
}
```

When a permutation of the set $S := \{0, 1, \ldots, n-1\}$ is given as a function X (where $X(S) = S$) the permutation can be applied to an array f via [FXT: `apply_permutation()` in perm/permapplyfunc.h]:

```
template <typename Type>
void apply_permutation(ulong (*x)(ulong), const Type *f, Type * restrict g, ulong n)
// Set g[k] <-- f[x(k)]  \forall k
// Must have:  0<=x(k)<n \forall k
{
    for (ulong k=0; k<n; ++k)  g[k] = f[x(k)];
}
```

For example, the statement `apply_permutation(inverse_gray_code, f, g, n)` is equivalent to `gray_permute(f, g, n)`. The inverse routine is

```
template <typename Type>
void apply_inverse_permutation(ulong (*x)(ulong), const Type *f, Type * restrict g, ulong n)
// Set g[x(k)] <-- f[k]  \forall k
// Must have:  0<=x(k)<n \forall k
{
    for (ulong k=0; k<n; ++k)  g[x(k)] = f[k];
}
```

The in-place versions of these routines are identical to the routines that apply permutations given as arrays. Only a tiny change must be made in the processing of the cycles. For example, the fragment

```
void apply_permutation(const ulong *x, Type * restrict f, ulong n, bitarray *bp=0)
  [--snip--]
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x[i];  // <--=
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = x[i];  // <--=
        }
        f[i] = t;
  [--snip--]
```

must be changed to (replace '`x[i]`' by '`x(i)`')

```
void apply_permutation(ulong (*x)(ulong), Type *f, ulong n, bitarray *bp=0)
  [--snip--]
        ulong i = k;  // start of cycle
        Type t = f[i];
        ulong g = x(i);  // <--=
        while ( 0==(tp->test_set(g)) )
        {
            f[i] = f[g];
            i = g;
            g = x(i);  // <--=
        }
        f[i] = t;
  [--snip--]
```

2.11.9 Random permutations

Routines for random permutations are given in [FXT: perm/permrand.h]. The following routine randomly permutes an array with arbitrary elements:

```
template <typename Type>
void random_permute(Type *f, ulong n)
{
    for (ulong k=1; k<n; ++k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16;  // avoid using low bits of rand alone
        ulong i = r % (k+1);
        swap2(f[k], f[i]);
    }
}
```

The method is given in [102]. A random permutation can be obtained by applying the function to the canonical sequence:

```
void random_permutation(ulong *f, ulong n)
// Create a random permutation
{
    for (ulong k=0; k<n; ++k)  f[k] = k;
    random_permute(f, n);
}
```

We note that a slight modification of the underlying idea can be used for a routine for random selection from a list with only one linear read. Let L be a list of n items $L_1, \ldots, L_n$.

1. Set $t = L_1$, set $k = 1$.
2. Set $k = k + 1$. If $k > n$ return t.
3. With probability $1/k$ set $t = L_k$.
4. Go to step 2.

A routine to apply a random cyclic permutation (as defined in section 2.11.4 on page 108) to an array is

```
template <typename Type>
void random_permute_cyclic(Type *f, ulong n)
// Permute the elements of f by a random cyclic permutation.
{
    for (ulong k=n-1; k>0; --k)
    {
        ulong r = (ulong)rand();
        r ^= r>>16;  // avoid using low bits of rand alone
        ulong i = r % k;
        swap2(f[k], f[i]);
    }
}
```

Finally, a random cyclic permutation can be obtained by applying a random cyclic permutation to the canonical sequence:

```
inline void
random_cyclic_permutation(ulong *f, ulong n)
// Create a random permutation that is cyclic.
{
    for (ulong k=0; k<n; ++k)  f[k] = k;
    random_permute_cyclic(f, n);
}
```

The cycle representation of a cyclic permutation can be obtained by applying a random permutation to all elements (of the identical permutation) except for the first element.

Chapter 3

Sorting and searching

In this chapter some practical flavors of sorting algorithms are given. These include plain sorting, sorting index arrays, pointer sorting; all optionally with a supplied comparison function. Massive literature exist about the topic so we will not go into the algorithmic details. Very readable text are both [89] and [212], in-depth information can be found in [156]. The sorting algorithms used in this chapter are selection sort, quicksort, counting sort and radix sort.

Some algorithms on sorted arrays like binary searching and determination of unique elements are included. Finally, some functions for scanning unsorted arrays are given.

3.1 Sorting

Selection sort

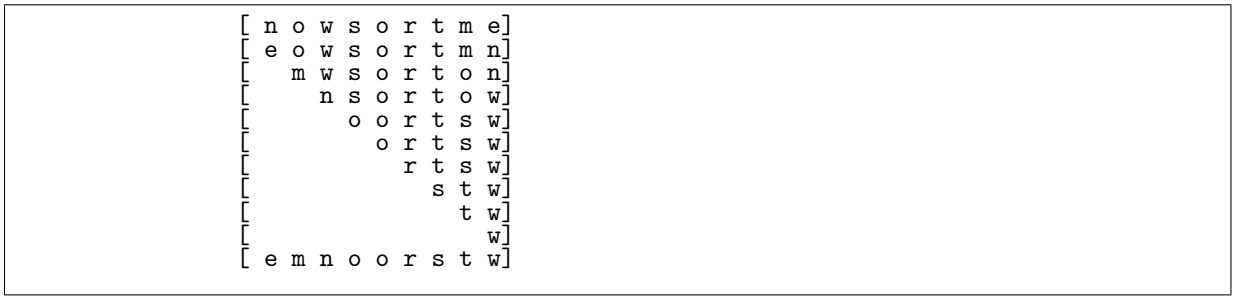


Figure 3.1-A: Sorting the string '`nowsortme`' with the selection sort algorithm.

There are a several algorithms for sorting that scale with $\sim n^2$ where n is the size of the array to be sorted. Here we use *selection sort* whose idea is to find the minimum of the array, swap it with the first element and repeat for all elements but the first. A demonstration of the algorithm is shown in figure 3.1-A, this is the output of [FXT: sort/selection-sort-demo.cc]. The implementation is straightforward [FXT: sort/sort.h]:

```
template <typename Type>
void selection_sort(Type *f, ulong n)
// Sort f[] (ascending order).
// Algorithm is proportional to O(n*n), use for short arrays only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
```

```
        while ( --j > i )  // search (index of) minimum
        {
            if ( f[j]<v )
            {
                m = j;
                v = f[m];
            }
        }
        swap2(f[i], f[m]);
    }
}
```

A verification routine is always handy:

```
template  <typename Type>
bool is_sorted(const Type *f, ulong n)
// Return whether the sequence f[0], f[1], ..., f[n-1]
// is sorted in ascending order.
{
    for (ulong k=1; k<n; ++k)  if ( f[k-1] > f[k] )  return false;
    return  true;
}
```

A test for descending order is

```
template  <typename Type>
bool is_falling(const Type *f, ulong n)
{
    for (ulong k=1; k<n; ++k)  if ( f[k-1] < f[k] )  return false;
    return  true;
}
```

Quicksort

The *quicksort* algorithm scales $\sim n\log(n)$ (in the average case). It does not just obsolete the more simple schemes because for arrays small enough the 'simple' algorithm is usually the fastest method because of its minimal bookkeeping overhead, and it can be used inside the quicksort for lengths below some threshold.

The main ingredient of quicksort is to *partition* the array. The corresponding routine reorders the array and returns an *pivot* index p so that $\max(f_0,\ldots,f_{p-1}) \le \min(f_p,\ldots,f_{n-1})$ [FXT: sort/sort.h]:

```
template <typename Type>
ulong partition(Type *f, ulong n)
{
    // Avoid worst case with already sorted input:
    const Type v = median3(f[0], f[n/2], f[n-1]);
    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do  { ++i; }  while ( f[i]<v );
        do  { --j; }  while ( f[j]>v );
        if ( i<j )  swap2(f[i], f[j]);
        else        return j;
    }
}
```

The function `median3()` is defined in [FXT: sort/minmaxmed23.h]:

```
template <typename Type>
static inline Type median3(const Type &x, const Type &y, const Type &z)
// Return median of the input values
{ return  x<y ? (y<z ? y : (x<z ? z : x)) : (z<y ? y : (z<x ? z : x)); }
```

The function does 2 or 3 comparisons, depending on the input. One could simply use the element `f[0]` as pivot. However, the algorithm will be $\sim n^2$ (that is, quadratic) when the array is already sorted.

Quicksort calls `partition` on the whole array, then on the two parts left and right from the partition index, then for the four, eight, ... parts, until the parts are of length one. Note that the sub-arrays are

usually of different lengths.

```
template <typename Type>
void quick_sort(Type *f, ulong n)
{
    if ( n<=1 )  return;

    ulong p = partition(f, n);
    ulong ln = p + 1;
    ulong rn = n - ln;
    quick_sort(f, ln);  // f[0]  ... f[ln-1]  left
    quick_sort(f+ln, rn);  // f[ln] ... f[n-1]   right
}
```

The actual implementation uses two optimizations: Firstly, when the size of the subproblems is smaller than a certain threshold selection sort is used. Secondly, the recursive calls are made for the smaller of the two sub-arrays, thereby the stack size is bounded by $\lceil \log_2(n) \rceil$.

```
template <typename Type>
void quick_sort(Type *f, ulong n)
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        selection_sort(f, n);
        return;
    }

    ulong p = partition(f, n);
    ulong ln = p + 1;
    ulong rn = n - ln;

    if ( ln>rn )  // recursion for shorter sub-array
    {
        quick_sort(f+ln, rn);  // f[ln] ... f[n-1]   right
        n = ln;
    }
    else
    {
        quick_sort(f, ln);  // f[0]  ... f[ln-1]  left
        n = rn;
        f += ln;
    }

    goto start;
}
```

The quicksort algorithm *will* be quadratic with certain inputs. A clever method to construct such inputs is described in [179]. A *heapsort* algorithm is $\sim n \cdot \log(n)$ also in the worst case, it is described in section 3.10 on page 134. Inputs that lead to quadratic time for the quicksort algorithm with median-of-3 partitioning are described in [185]. There it is suggested to use quicksort but detect problematic behavior during runtime and switch to heapsort if needed. The corresponding algorithm is called *introsort* (for *introspective Sorting*).

3.2 Binary search

The main reason for sorting may be that a fast search has to be performed repeatedly. The *binary search* algorithm works by the obvious subdivision of the data [FXT: `bsearch()` in sort/bsearch.h]:

```
template <typename Type>
ulong bsearch(const Type *f, ulong n, const Type v)
// Return index of first element in f[] that equals v
// Return ~0 if there is no such element.
// f[] must be sorted in ascending order.
// Must have  n!=0
{
    ulong nlo=0, nhi=n-1;
    while ( nlo != nhi )
    {
        ulong t = (nhi+nlo)/2;

        if ( f[t] < v )  nlo = t + 1;
        else             nhi = t;
```

```
    }
    if ( f[nhi]==v )  return nhi;
    else              return ~0UL;
}
```

The algorithm uses $\sim \log_2(n)$ operations. For very large arrays the algorithm can be improved by selecting the new index `t` different from midpoint `(nhi+nlo)/2`, dependent of the value sought and the distribution of the values in the array. As a simple example consider an array of floating point numbers that are equally distributed in the interval $[\min(v), \max(v)]$. If the sought value equals v one would want to use the relation

$$\frac{n - \min(n)}{\max(n) - \min(n)} \quad = \quad \frac{v - \min(v)}{\max(v) - \min(v)} \tag{3.2-1}$$

where n denotes an index, and $\min(n), \max(n)$ denote the minimal and maximal index of the current interval. Solving for n gives the linear interpolation formula

$$n \quad = \quad \min(n) + \frac{\max(n) - \min(n)}{\max(v) - \min(v)} \, (v - \min(v)) \tag{3.2-2}$$

The corresponding *interpolation binary search* algorithm would select the new subdivision index `t` according to the given relation. One could even use quadratic interpolation schemes for the selection of `t`. For the majority of practical applications the midpoint version of the binary search will be good enough.

A simple modification of `bsearch` makes it search the first element greater than or equal to `v`: replace the operator `==` in the above code by `>=` and you have it: [FXT: `bsearch_ge()` in sort/bsearch.h]. Similar for the '`<=`' relation: `bsearch_le()`.

Approximate matches are found by [FXT: `bsearch_approx()` in sort/bsearchapprox.h]:

```
template <typename Type>
ulong bsearch_approx(const Type *f, ulong n, const Type v, Type da)
// Return index of first element x in f[] for which  |(x-v)| <= da
// Return ~0 if there is no such element.
// f[] must be sorted in ascending order.
// da must be positive.
//
// Makes sense only with inexact types (float or double).
// Must have  n!=0
{
    ulong k = bsearch_ge(f, n, v-da);
    if ( k<n )  k = bsearch_le(f+k, n-k, v+da);
    return k;
}
```

3.3 Index sorting

While the 'plain' sorting reorders an array f so that, after it has finished, $f_k \le f_{k+1}$ the following routines sort an array of indices without modifying the actual data. The index-sort routines reorder the indices in an array `x` such that `x` applied to `f` as a permutation (in the sense of section 2.11.8 on page 111) will render `f` a sorted array [FXT: sort/sortidx.h]:

```
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
```

```
        while ( --j > i )  // search (index of) minimum
        {
            if ( f[x[j]]<v )
            {
                m = j;
                v = f[x[m]];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

Apart from the 'read only'-feature the index-sort routines have the nice property to perfectly work on non-contiguous data. The verification code is

```
template  <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x)
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( f[x[n]] < f[x[n-1]] )  break;
    }
    return  !n;
}
```

The transformation of the `partition()` routine is straightforward:

```
template <typename Type>
ulong idx_partition(const Type *f, ulong n, ulong *x)
// rearrange index array, so that for some index p
// max(f[x[0]] ... f[x[p]]) <= min(f[x[p+1]] ... f[x[n-1]])
{
    // Avoid worst case with already sorted input:
    const Type v = median3(*x[0], *x[n/2], *x[n-1], cmp);
    ulong i = 0UL - 1;
    ulong j = n;
    while ( 1 )
    {
        do  ++i;
        while ( f[x[i]]<v );
        do  --j;
        while ( f[x[j]]>v );
        if ( i<j )  swap2(x[i], x[j]);
        else        return j;
    }
}
```

The index-quicksort itself deserves a minute of contemplation comparing it to the plain version:

```
template <typename Type>
void idx_quick_sort(const Type *f, ulong n, ulong *x)
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order.
{
 start:
    if ( n<8 ) // parameter: threshold for nonrecursive algorithm
    {
        idx_selection_sort(f, n, x);
        return;
    }
    ulong p = idx_partition(f, n, x);
    ulong ln = p + 1;
    ulong rn = n - ln;
    if ( ln>rn )  // recursion for shorter sub-array
    {
        idx_quick_sort(f, rn, x+ln);  // f[x[ln]] ... f[x[n-1]]  right
        n = ln;
```

```
    }
    else
    {
        idx_quick_sort(f, ln, x);  // f[x[0]]  ... f[x[ln-1]]  left
        n = rn;
        x += ln;
    }
    goto start;
}
```

The index-analogues of the binary search algorithms are again straightforward, they are given in [FXT: sort/bsearchidx.h].

3.4 Pointer sorting

Pointer sorting is an idea similar to index sorting which is even less restricted than index sort: The data may be unaligned in memory. And overlapping. Or no data at all but port addresses controlling some highly dangerous machinery. Thereby pointer sort is the perfect way to highly cryptic and powerful programs that seg-fault when you least expect it.

Just to make the idea clear, the array of indices is replaced by an array of pointers:

```
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x)
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( *x[j]<v )
            {
                m = j;
                v = *x[m];
            }
        }
        swap2(x[i], x[m]);
    }
}
```

The first argument (`const Type *f`) is not necessary with pointer sorting. It is indicated as comment to make the argument structure clear. The verification routine is

```
template  <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x)
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order.
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( *x[n] < *x[n-1] )  break;
    }
    return  !n;
}
```

Find the pointer sorting code in [FXT: sort/sortptr.h]. The pointer versions of the search routines are given in [FXT: sort/bsearchptr.h].

3.5 Sorting by a supplied comparison function

The routines in [FXT: sort/sortfunc.h] are similar to the C-quicksort `qsort` that is part of the standard library. A comparison function `cmp` has to be supplied by the called so that compound data types can be sorted with respect to some key contained. Citing the manual page for `qsort`:

```
The comparison function must return an integer less than, equal to, or greater than
zero if the first argument is considered to be respectively less than, equal to, or
greater than the second.  If two members compare  as  equal,  their  order  in  the
sorted array is undefined.
```

Note that the numerous calls to `cmp` do have a negative impact on the performance. With C++ you can provide a comparison 'function' for compound data by overloading the operators `<`, `<`, `<=` and `>=` and use the plain version. That is, the comparisons are inlined an we are back in performance land. Isn't C++ nice? As a prototypical example we give the selection sort routine:

```
template <typename Type>
void selection_sort(Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Sort f[] (ascending order)
// with respect to comparison function cmp().
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[i];
        ulong m = i; // position of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(f[j],v) < 0 )
            {
                m = j;
                v = f[m];
            }
        }
        swap2(f[i], f[m]);
    }
}
```

The other routines are rather straightforward translations of the (plain-) sort analogues: replace the comparison operations as follows

```
    (a < b)      cmp(a,b) < 0
    (a > b)      cmp(a,b) > 0
    (a == b)     cmp(a,b) == 0
    (a <= b)     cmp(a,b) <= 0
    (a >= b)     cmp(a,b) >= 0
```

For example, the verification routine is

```
template  <typename Type>
bool is_sorted(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// f[0], f[1], ..., f[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
{
    if ( 0==n )  return 1;
    while ( --n )  // n-1 ... 1
    {
        if ( cmp(f[n], f[n-1]) < 0 )  break;
    }
    return  !n;
}
```

3.5.1 Sorting complex numbers

You want to sort complex numbers? Fine for me, but *don't* tell your local mathematician. To see the mathematical problem we ask whether i is smaller or greater than zero. Assume $i > 0$: follows $i \cdot i > 0$ (we multiplied with a positive value) which is $-1 > 0$ and that is false. So, is $i < 0$? Then $i \cdot i > 0$ (multiplication with a negative value, as assumed). So $-1 > 0$, oops! The lesson is that there is no way to impose an arrangement on the complex numbers that would justify the usage of the symbols '$<$' and '$>$' consistent with the rules to manipulate inequalities.

Nevertheless we can invent a relation that allows us to sort: arranging (sorting) the complex numbers according to their absolute value (modulus) leaves infinitely many numbers in one 'bucket', namely all those that have the same distance from zero. However, one could use the modulus as the *major* ordering parameter, the angle as the *minor*. Or the real part as the major and the imaginary part as the minor. The latter is realized in

```
static inline int
cmp_complex(const Complex &f, const Complex &g)
{
    const double fr = f.real(),  gr = g.real();
    if ( fr!=gr )  return  (fr>gr ? +1 : -1);

    const double fi = f.imag(),  gi = g.imag();
    if ( fi!=gi )  return  (fi>gi ? +1 : -1);

    return  0;
}
```

This routine, when used as comparison with the function-sort, as in

```
void complex_sort(Complex *f, ulong n)
// major order wrt. real part
// minor order wrt. imag part
{
    quick_sort(f, n, cmp_complex);
}
```

can indeed be the practical tool you had in mind.

3.5.2 Index and pointer sorting

The index sorting routines that use a supplied comparison function are given in [FXT: sort/sortidxfunc.h]:

```
template <typename Type>
void idx_selection_sort(const Type *f, ulong n, ulong *x,
                        int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
// with respect to comparison function cmp()
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = f[x[i]];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(f[x[j]], v) < 0 )
            {
                m = j;
                v = f[x[m]];
            }
        }

        swap2(x[i], x[m]);
    }
}
```

The verification routine is:

```
template  <typename Type>
bool is_idx_sorted(const Type *f, ulong n, const ulong *x,
                   int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// f[x[0]], f[x[1]], ... f[x[n-1]]
// is sorted in ascending order
// with respect to comparison function cmp()
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 1
    {
        if ( cmp(f[x[n]], f[x[n-1]]) < 0 )  break;
    }

    return  !n;
}
```

The pointer sorting versions are given in [FXT: sort/sortptrfunc.h]

```
template <typename Type>
void ptr_selection_sort(/*const Type *f,*/ ulong n, const Type **x,
                        int (*cmp)(const Type &, const Type &))
// Sort x[] so that the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
// Algorithm is proportional to O(n*n), use for short array only.
{
    for (ulong i=0; i<n; ++i)
    {
        Type v = *x[i];
        ulong m = i; // position-ptr of minimum
        ulong j = n;
        while ( --j > i )  // search (index of) minimum
        {
            if ( cmp(*x[j],v)<0 )
            {
                m = j;
                v = *x[m];
            }
        }

        swap2(x[i], x[m]);
    }
}
```

The verification routine is:

```
template  <typename Type>
bool is_ptr_sorted(/*const Type *f,*/ ulong n, Type const*const*x,
                   int (*cmp)(const Type &, const Type &))
// Return whether the sequence
// *x[0], *x[1], ..., *x[n-1]
// is sorted in ascending order
// with respect to comparison function cmp().
{
    if ( 0==n )  return 1;

    while ( --n )  // n-1 ... 1
    {
        if ( cmp(*x[n],*x[n-1])<0 )  break;
    }

    return  !n;
}
```

The corresponding versions of the binary search algorithm are given in [FXT: sort/bsearchidxfunc.h] and [FXT: sort/bsearchptrfunc.h].

3.6 Determination of unique elements

We present functions that check whether values in a sorted array are repeated or unique. All routines are taken from [FXT: sort/unique.h]. To test whether all values are unique, use

```
template  <typename Type>
ulong test_unique(const Type *f, ulong n)
// For a sorted array test whether all values are unique
//   (i.e. whether no value is repeated).
// Return 0 if all values are unique else return index of the second
// element in the first pair found.
{
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] == f[k-1] )  return  k;  // k != 0
    }
    return  0;
}

template  <typename Type>
ulong is_unique(const Type *f, ulong n)
// Return true if all values are unique, else return false.
{
    return ( 0==test_unique(f, n) );
}
```

Counting the elements that appear just once:

```
template  <typename Type>
int unique_count(const Type *f, ulong n)
// For a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
//   values is n - unique_count(f, n);
{
    if ( 1>=n )  return n;
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( f[k] != f[k-1] )  ++ct;
    }
    return  ct;
}
```

Removing repeated elements:

```
template  <typename Type>
ulong unique(Type *f, ulong n)
// For a sorted array squeeze all repeated values
// and return the number of unique values.
// Example:  [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long
// as identical elements only appear in contiguous blocks.
// Example: [4, 4, 3, 7, 7] --> [4, 3, 7]
// The order is preserved.
{
    ulong u = unique_count(f, n);
    if ( u==n )  return n;  // nothing to do
    Type v = f[0];
    for (ulong j=1, k=1;  j<u;  ++j)
    {
        while ( f[k]==v )  ++k;  // search next different element
        v = f[j] = f[k];
    }
    return u;
}
```

The inner `while`-loop does never access an element out of bounds as it is executed only as long as there is at least one remaining change of value inside the array.

3.7 Unique elements with inexact types

Determination of unique elements with inexact types (floats) is a bit tricky as one cannot rely that elements that *should* be identical are exactly equal. A solution to the problem is to allow for a maximal (absolute) difference within which two contiguous elements will still be considered equal can be provided as additional parameter. We replace equality conditions with a call to [FXT: sort/uniqueapprox.h]

```
template  <typename Type>
inline bool approx_equal(Type x1, Type x2, Type da)
// Return whether abs(x2-x1) <= da
// Must have da>=0
{
    Type d = x2 - x1;
    if ( d<=0 )  d = -d;
    if ( d <= da )  return true;
    else            return false;
}
```

The verification routine is

```
template  <typename Type>
ulong test_unique_approx(const Type *f, ulong n, Type da)
// For a sorted array test whether all values are
// unique within some tolerance (i.e. whether no value is repeated).
// Return 0 if all values are unique,
// else return index of the second element in the first pair found.
// Makes mostly sense with inexact types (float or double)
{
    if ( da<=0 )  da = -da;  // want positive tolerance

    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) )  return k;  // k != 0
    }

    return  0;
}
```

One subtle point is that the values can slowly 'drift away' unnoticed by this implementation: consider a long array where each difference computed has the same sign and is just smaller than `da`, say it is $d = 0.6 \cdot$`da`. The difference of the first and last value then is $0.6 \cdot (n-1) \cdot d$ which is greater than `da` for $n \geq 3$.

The number of unique elements can be counted as follows:

```
template  <typename Type>
ulong unique_approx_count(const Type *f, ulong n, Type da)
// For a sorted array return the number of unique values
// the number of (not necessarily distinct) repeated
//   values is n - unique_approx_count(f, n, da);
{
    if ( 1>=n )  return n;
    if ( da<=0 )  da = -da;  // Must have positive tolerance
    ulong ct = 1;
    for (ulong k=1; k<n; ++k)
    {
        if ( approx_equal(f[k], f[k-1], da) )  ++ct;
    }
    return  ct;
}
```

The following routine removes duplicates:

```
template  <typename Type>
ulong unique_approx(Type *f, ulong n, Type da)
// For a sorted array squeeze all repeated (within tolerance da) values
// and return the number of unique values.
// Example:  [1, 3, 3, 4, 5, 8, 8] --> [1, 3, 4, 5, 8]
// The routine also works for unsorted arrays as long
// as identical elements only appear in contiguous blocks.
// Example: [4, 4, 3, 7, 7] --> [4, 3, 7]
// The order is preserved.
{
    ulong u = unique_approx_count(f, n, da);
```

```
    if ( u==n )  return n;  // nothing to do
    if ( da<=0 )  da = -da;  // Must have positive tolerance
    Type v = f[0];
    for (ulong j=1, k=1;  j<u;  ++j)
    {
        // search next different element:
        while ( approx_equal(f[k], v, da) )
        {
            v = f[k];  // avoid problem with slowly drifting values
            ++k;
        }
        v = f[j] = f[k];
    }
    return u;
}
```

A useful preprocessing step (before using `test_unique_approx()`) is to quantize the elements of an array [FXT: `quantize()` in sort/quantize.h]:

```
template  <typename Type>
void quantize(Type *f, ulong n, double q)
// In f[] set each element x to q*floor(1/q*(x+q/2))
// E.g.: q=1 ==> round to nearest integer
//       q=1/1000 ==> round to nearest multiple of 1/1000
// For inexact types (float or double).
{
    Type qh = q * 0.5;
    Type q1 = 1.0 / q;
    while ( n-- )
    {
        f[n] = q * floor( q1 * (f[n]+qh) );
    }
}
```

One should use a quantization parameter `q` that is greater than the value used for `da`.

A simple demonstration is given in [FXT: sort/unique-demo.cc]:

```
    Random values:
       0:  0.9727750243
       1:  0.2925167845
       2:  0.7713576982
       3:  0.5267449795
       4:  0.7699138366
       5:  0.4002286223

    Quantization with q=0.01

    Quantized & sorted :
       0:  0.2900000000
       1:  0.4000000000
       2:  0.5300000000
       3:  0.7700000000
       4:  0.7700000000
       5:  0.9700000000
    First REPEATED value at index 4  (and 3)

    Unique'd array:
       0:  0.2900000000
       1:  0.4000000000
       2:  0.5300000000
       3:  0.7700000000
       4:  0.9700000000
```

The routine `quantize()` turns out to be also useful for the conversion of imprecise data to symbols. For example, the array of floating point values on the left corresponds to the symbolic (numbers used as symbols) table on the right:

```
      1.3133     -1.0101     0.79412     -0.71544          9   2   6   3
     0.29064      0.99173   -1.4382       0.79412          5   7   0   6
    -1.1086       1.2521     0.99173     -1.0101           1   8   7   2
    -0.18003     -1.1086     0.29064      1.3133           4   1   5   9
```

In this example values were considered identical when their absolute difference is less than 10^{-3}. The symbolic representation can be helpful to recognize structure in imprecise data. The routine is [FXT: sort/symbolify.h]:

```
template <typename Type>
ulong symbolify_by_size(const Type *f, Type * restrict g, ulong n,
                        Type eps=1e-6, ulong *ix=0)
```

```
// From f[] compute an array of 'symbols' g[] (i.e. numbers)
//   that represent the different values.
// Values are considered identical if their absolute difference
//   is less than eps.
// Symbols are given with respect to sort-order.
// Return number of different values found (after quantize).
// Optionally supply x[] (scratch space for permutations).
{
    ulong *x = ix;
    if ( 0==ix )  x = new ulong[n];

    set_seq(x, n);
    idx_quick_sort(f, n, x);

    apply_permutation(x, f, g, n);
    quantize(g, n, eps);

    eps *= 0.5;  // some val <1.0
    ulong nsym = 1;

    ulong z = 0;
    Type s = 0.0;
    Type el = g[z], lel;
    g[z] = s;
    for(ulong k=z+1; k<n; ++k)
    {
        lel = el;
        el = g[k];
        if ( fabs(el-lel) > eps )
        {
            ++nsym;
            s += 1.0;
        }
        g[k] = s;
    }

    apply_inverse_permutation(x, g, n);

    if ( 0==ix )  delete [] x;

    return  nsym;
}
```

The example shown was created with the program [FXT: sort/symbolify-demo.cc]. The routines `apply_permutation()` and `apply_inverse_permutation()` are given in section 2.11.8 on page 111.

3.8 Determination of equivalence classes

Let S be a set and $C := S \times S$ the set of all ordered pairs (x, y) with $x, y \in S$. A *binary relation* R on S is a subset of C. An *equivalence relation* is a binary relation that has three additional properties:

- *reflexive*: $x \equiv x \; \forall x$.
- *symmetric*: $x \equiv y \iff y \equiv x$.
- *transitive*: $x \equiv y,\ y \equiv z \implies x \equiv z$.

Here we wrote $x \equiv y$ for $(x, y) \in R$ where $x, y \in S$.

We want to determine the *equivalence classes*: an equivalence relation partitions a set into $1 \le q \le n$ subsets E_1, E_2, ..., E_q so that $x \equiv y$ whenever both x and y are in the same subset but $x \not\equiv y$ if x and y are in different subsets.

For example, the usual equality relation is an equivalence relation, with a set of (different) numbers each number is in its own class. With the equivalence relation that $x \equiv y$ whenever $x - y$ is a multiple of some fixed integer m and the set $\mathbb{Z}$ of all natural numbers we obtain m subsets and $x \equiv y$ if and only if $x \equiv y \bmod m$.

Let n be the number of elements in S and Q be a set so that, on termination of the algorithm, $Q_k = j$ if j is the least index so that $S_j \equiv S_k$ (note that we consider the sets to be in a fixed but arbitrary order

here).

We proceed as follows:

1. Put each element in its own equivalence class: $Q_k := k$ for all $0 \le k < n$
2. Set $k := 1$ (index of the second element).
3. Search downwards for an equivalent element: for $j = k-1, \ldots, 0$ test whether $S_k \equiv S_j$. If so (at position j), set $Q_k = Q_j$ and goto step 4.
4. Set $k := k+1$ and if $k < n$ goto step 3, else terminate.

We can terminate the search with the first equivalent element found because, if j is the index of the equivalent, the Q_j is already minimal.

The lower and upper bounds for the computational cost are n and n^2, respectively: the algorithm needs proportional n operations when all elements are in the same equivalence class and n^2 operations when each element lies in its own class.

A C++ implementation is [FXT: `equivalence_classes()` in sort/equivclasses.h]. The equivalence relation must be supplied as a function `equiv_q()` that returns true when its arguments are equivalent.

```
template <typename Type>
void equivalence_classes(const Type *s, ulong n, bool (*equiv_q)(Type,Type), ulong *q)
// Given an equivalence relation '==' (as function equiv_q())
//  and a set s[] with n elements,
// write into q[k] (0<=k<n) the index j of the
//  first element s[j] so that s[k]==s[j].
// For the complexity C:  n<=C<=n*n
//  C=n*n if each element is in its own class
//  C=n   if all elements are in the same class
{
    for (ulong k=0; k<n; ++k)  q[k] = k;  // each in own class
    for (ulong k=1; k<n; ++k)
    {
        ulong j = k;
        while ( j-- )
        {
            if ( equiv_q(s[j], s[k]) )
            {
                q[k] = q[j];
                break;
            }
        }
    }
}
```

3.8.1 Examples for equivalence classes

3.8.1.1 Integers modulo m

Choose an integer $m \ge 2$ and let any two integers a and b be equivalent if $a - b$ is a integer multiple of m. We can choose the numbers $0, 1 \ldots, m-1$ as *representatives* of the m classes obtained. Now we can do computations with those classes via the modular arithmetic as described in section 37.1 on page 731. This is easily the most important example of all equivalence classes.

We note that the concept still make sense with a real (that is, possibly non-integral) modulus m. We still put two numbers a and b into the same class if $a - b$ is a *integer* multiple of m. Finally, the modulus zero leads to the equivalence relation 'equality'.

3.8.1.2 Binary necklaces

The set S of n-bit binary words and the equivalence relation so that two words x and y are equivalent when there is a cyclic shift $h_k(x)$ by $0 \le k < n$ positions so that $h_k(x) = y$. The relation is supplied as the function [FXT: sort/equivclass-necklaces-demo.cc]:

```
static ulong b;  // number of bits
bool n_equiv_q(ulong x, ulong y)  // necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    return  (0==d);
}
```

With $n = 4$ we obtain the following list of equivalence classes:

```
     0:   ....  [#=1]
     1:   1...  .1..  ...1  ..1.  [#=4]
     3:   1..1  11..  ..11  .11.  [#=4]
     5:   .1.1  1.1.  [#=2]
     7:   11.1  111.  1.11  .111  [#=4]
    15:   1111  [#=1]
  # of equivalence classes = 6
```

These correspond to the *binary necklaces* of length 4. One usually chooses the cyclic minima (or maxima) among equivalent words as representatives of the classes.

3.8.1.3 Unlabeled binary necklaces

Same set but the equivalence relation is defined to identify two words x and y when there is a cyclic shift $h_k(x)$ by $0 \le k < b$ positions so that either $h_k(x) = y$ or $h_k(x) = \overline{y}$ where $\overline{y}$ is the complement of y:

```
static ulong mm;  // mask to complement
bool nu_equiv_q(ulong x, ulong y)  // unlabeled necklaces
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d )  d = bit_cyclic_dist(mm^x, y, b);
    return  (0==d);
}
```

With $n = 4$ we obtain:

```
    0:   1111  ....  [#=2]
    1:   111.  11.1  1.11  1...  .111  ...1  ..1.  .1..  [#=8]
    3:   .11.  1..1  11..  ..11  [#=4]
    5:   .1.1  1.1.  [#=2]
  # of equivalence classes = 4
```

These correspond to the *unlabeled binary necklaces* of length 4.

3.8.1.4 Binary bracelets

The *binary bracelets* are obtained by identifying two words that are identical up to rotation and possible reversion. The corresponding comparison function is

```
bool b_equiv_q(ulong x, ulong y)  // bracelets
{
    ulong d = bit_cyclic_dist(x, y, b);
    if ( 0!=d )  d = bit_cyclic_dist(revbin(x,b), y, b);
    return  (0==d);
}
```

There are six binary bracelets of length 4:

```
  0:   ....  [#=1]
  1:   1...  .1..  ...1  ..1.  [#=4]
  3:   1..1  11..  ..11  .11.  [#=4]
  5:   .1.1  1.1.  [#=2]
  7:   11.1  111.  1.11  .111  [#=4]
 15:   1111  [#=1]
```

The *unlabeled binary bracelets* are obtained by additionally allowing for bit-wise complementation:

```
bool bu_equiv_q(ulong x, ulong y)  // unlabeled bracelets
{
    ulong d = bit_cyclic_dist(x, y, b);
    x ^= mm;
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);

    x = revbin(x,b);
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);
    x ^= mm;
    if ( 0!=d )  d = bit_cyclic_dist(x, y, b);

    return  (0==d);
}
```

There are four unlabeled binary bracelets of length 4:

```
  0:   1111  ....  [#=2]
  1:   111.  11.1  1.11  1...  .111  ...1  ..1.  .1..  [#=8]
  3:   .11.  1..1  11..  ..11  [#=4]
  5:   .1.1  1.1.  [#=2]
```

The shown functions are given in [FXT: sort/equivclass-bracelets-demo.cc] which can be used to produce listings of the equivalence classes.

3.8.1.5 The number of necklaces and bracelets

We give the number of binary necklaces '`N`', bracelets '`B`', unlabeled necklaces '`N/U`' and unlabeled bracelets '`B/U`'. The second row gives the sequence number of [214].

n :	N	B	N/U	B/U
[214]#	A000031	A000029	A000013	A000011
1:	2	2	1	1
2:	3	3	2	2
3:	4	4	2	2
4:	6	6	4	4
5:	8	8	4	4
6:	14	13	8	8
7:	20	18	10	9
8:	36	30	20	18
9:	60	46	30	23
10:	108	78	56	44
11:	188	126	94	63
12:	352	224	180	122
13:	632	380	316	190
14:	1182	687	596	362
15:	2192	1224	1096	612

3.8.1.6 Binary words with reversion and complement

The set S of n-bit binary words and the equivalence relation identifying two words x and y whenever they are mutual complements or bit-wise reversals.

For example, the equivalence classes with 3-, 4- and 5-bit words are shown in figure 3.8-A. The sequence of numbers of equivalence classes for word-sizes n is (entry A005418 of [214])

```
n:  1, 2, 3, 4,  5,  6,  7,  8,   9,  10,  11,   12,   13,   14,   15,    16, ...
#:  1, 2, 3, 6, 10, 20, 36, 72, 136, 272, 528, 1056, 2080, 4160, 8256, 16512, ...
```

The equivalence classes can be computed with the program [FXT: sort/equivclass-bitstring-demo.cc].

```
      3 classes with 3-bit words:
        0:   111  ...
        1:   ..1  .11  1..  11.
        2:   1.1  .1.

      6 classes with 4-bit words:
        0:   1111  ....
        1:   111.  1...  .111  ...1
        2:   ..1.  .1..  1.11  11.1
        3:   11..  ..11
        5:   1.1.  .1.1
        6:   .11.  1..1
```

```
10 classes with 5-bit words:
  0:   11111  .....
  1:   1111.  1....  .1111  ....1
  2:   1.111  111.1  .1...  ...1.
  3:   111..  ...11  ..111  11...
  4:   ..1..  11.11
  5:   11.1.  1.1..  ..1.1  .1.11
  6:   ..11.  .11..  11..1  1..11
  9:   .11.1  1.11.  .1..1  1..1.
 10:   .1.1.  1.1.1
 14:   1...1  .111.
```

Figure 3.8-A: Equivalence classes of binary words where words are identified if either their reversals or complements are equal.

We have chosen examples where the resulting equivalence classes can be verified by inspection. For example, we could create the subsets of equivalent necklaces by simply rotating a given word and marking the so far visited words. Such an approach, however, is not possible in general when the equivalence relation does not have an obvious structure.

3.8.2 The number of equivalence relations for a set of n elements

The sequence $B(n)$ of the number of possible partitionings (and thereby equivalence relations) for the set $\{1, 2, \ldots, n\}$ starts as ($n \geq 1$):

```
1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, ...
```

These are the *Bell numbers*, sequence A000110 of [214]. They can be computed easily as indicated in the following table:

```
0:  [ 1]
1:  [ 1,  2]
2:  [ 2,  3,  5]
3:  [ 5,  7, 10,  15]
4:  [15, 20, 27,  37,  52]
5:  [52, 67, 87, 114, 151, 203]
n:  [B(n), ... ]
```

The first element in each row is the last element of the previous row, the remaining elements are the sum of their left and upper left neighbors. As pari/gp code:

```
N=7;  v=w=b=vector(N);  v[1]=1;
{ for(n=1,N-1,
    b[n] = v[1];
    print(n-1, ":  ", v); \\ print row
    w[1] = v[n];
    for(k=2,n+1, w[k]=w[k-1]+v[k-1]);
    v=w;
 ); }
```

An implementation in C++ is given in [FXT: comb/bell-number-demo.cc]. An alternative way to compute the Bell Numbers is shown in section 15.1 on page 320.

3.9 Determination of monotonicity and convexity *

A sequence is called *monotone* if it is either purely ascending or purely descending. This includes the case where subsequent elements are equal. Whether a constant sequence is considered ascending or descending in this context is a matter of convention.

A routine to check for monotonicity is [FXT: sort/monotone.h]:

```
template  <typename Type>
int is_monotone(const Type *f, ulong n)
```

```
// Return
//   +1 for ascending order
//   -1 for descending order
//   else 0
{
    if ( 1>=n )  return  +1;

    ulong k;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }

    if ( k==n )  return +1;  // constant is considered ascending here

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }

    return  s;
}
```

A *strictly monotone* sequence is a monotone sequence that has no identical pairs of elements. The test turns out to be slightly easier:

```
template  <typename Type>
int is_strictly_monotone(const Type *f, ulong n)
// return
//   +1 for strictly ascending order
//   -1 for strictly descending order
//   else 0
{
    if ( 1>=n )  return  +1;

    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;

    int s = ( f[k] > f[k-1] ? +1 : -1 );

    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }

    return  s;
}
```

A sequence is called *convex* if it starts with an ascending part and ends with a descending part. A *concave* sequence starts with a descending and ends with an ascending part. Whether a monotone sequence is considered convex or concave again is a matter of convention (you have the choice to consider the first or the last element as extremum). Lacking a term that contains both convex and concave the following routine is called `is_convex()` [FXT: sort/convex.h]:

```
template <typename Type>
long is_convex(Type *f, ulong n)
// Return
//   +val for convex sequence (first rising then falling)
//   -val for concave sequence (first falling then rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is monotone.
//
// Note: a constant sequence is considered any of rising/falling
```

```
{
    if ( 1>=n )  return +1;
    ulong k = 1;
    for (k=1; k<n; ++k)  // skip constant start
    {
        if ( f[k] != f[k-1] )  break;
    }
    if ( k==n )  return  +n;  // constant is considered convex here
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  break;
        s = -k;
    }
    if ( k==n )  return s;  // sequence is monotone
    // check that the ordering does not change again:
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for strictly ascending pair:
        for (  ; k<n; ++k)  if ( f[k] > f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for strictly descending pair:
        for (  ; k<n; ++k)  if ( f[k] < f[k-1] )  return 0;
    }
    return  s;
}
```

The test for *strictly convex* (or concave) sequences is:

```
template <typename Type>
long is_strictly_convex(Type *f, ulong n)
// Return
//   +val for strictly convex sequence
//        (i.e. first strictly rising then strictly falling)
//   -val for strictly concave sequence
//        (i.e. first strictly falling then strictly rising)
//   else 0
//
// val is the (second) index of the first pair at the point
// where the ordering changes;  val>=n iff seq. is strictly monotone.
{
    if ( 1>=n )  return +1;
    ulong k = 1;
    if ( f[k] == f[k-1] )  return 0;
    int s = ( f[k] > f[k-1] ? +1 : -1 );
    if ( s>0 ) // was: ascending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  break;
        s = +k;
    }
    else // was: descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  break;
        s = -k;
    }
    if ( k==n )  return s;  // sequence is monotone
    else  if ( f[k] == f[k-1] )  return 0;
    // check that the ordering does not change again:
```

```
    if ( s>0 ) // was: ascending --> descending
    {
        // scan for ascending pair:
        for (  ; k<n; ++k)  if ( f[k] >= f[k-1] )  return 0;
    }
    else // was: descending
    {
        // scan for descending pair:
        for (  ; k<n; ++k)  if ( f[k] <= f[k-1] )  return 0;
    }
    return  s;
}
```

3.10 Heapsort

The *heapsort* algorithm uses the heap data structure introduced in section 4.5 on page 148. A heap can be sorted by swapping the first (and biggest) element with the last and 'repairing' the array of size $n-1$ by a call to `heapify1()`. Applying this idea recursively until there is nothing more to sort leads to the routine [FXT: sort/heapsort.h]:

```
template <typename Type>
void heap_sort_ascending(Type *x, ulong n)
// Sort an array that has the heap-property into ascending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        swap2(p[1], p[k]);   // move largest to end of array
        --n;                 // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

that needs time $O(n \log(n))$. That is, a call to

```
template <typename Type>
void heap_sort(Type *x, ulong n)
{
    build_heap(x, n);
    heap_sort_ascending(x, n);
}
```

will sort the array `x[]` into ascending order. Note that sorting into descending order is not any harder:

```
template <typename Type>
void heap_sort_descending(Type *x, ulong n)
// Sort an array that has the heap-property into descending order.
// On return x[] is _not_ a heap anymore.
{
    Type *p = x - 1;
    for (ulong k=n; k>1; --k)
    {
        ++p;  --n;           // remaining array is one element less
        heapify1(p, n, 1);  // restore heap-property
    }
}
```

A program that demonstrates the algorithm is [FXT: sort/heapsort-demo.cc].

3.11 Counting sort and radix sort

Imagine you want to sort an n-element array F of (unsigned) 8-bit values. An sorting algorithm that only uses 2 passes through the data proceeds as follows:

1. Allocate an array C of 256 integers and set all its elements to zero.

2. Count: for $k = 0, 1, \ldots, n-1$ increment $C[F[k]]$.
   Now $C[x]$ contains how many bytes in F have the value x.

3. Set $r = 0$. For $j = 0, 1, \ldots, 255$
   set $k = C[j]$ and write j to the elements $F[r]$, $F[r+1]$, $\ldots$, $F[r+k-1]$ then add k to r.

For large values of n this method is significantly faster than any other sorting algorithm. Note that no comparisons are made between the elements of F. Instead they are counted, the algorithm is the *counting sort* algorithm.

It might seem that the idea applies only to very special cases but with a little care it can be used in more general situations. We modify the method so that we are able to sort also (unsigned) integer variables whose range of values would make the method impractical with respect to a subrange of the bits in each word. We need an array G that has as many elements as F:

1. Choose any consecutive run of b bits, these will be represented by a bit mask m. Allocate an array C of 2^b integers and set all its elements to zero.

2. Let M be a function that maps the (2^b) values of interest (the bits masked out by m) to the range $0, 1, \ldots, 2^b - 1$.

3. Count: for $k = 0, 1, \ldots, n-1$ increment $C[M(F[k])]$.
   Now $C[x]$ contains how many values of $M(F[.])$ equal x.

4. Cumulate: for $j = 1, 2, \ldots, 2^b - 1$ (second to last) add $C[j-1]$ to $C[j]$.
   Now $C[x]$ contains the number of values $M(F[.])$ less or equal to x.

5. Copy: for $k = n-1, \ldots, 2, 1, 0$ (last to first) set $x := M(F[k])$, decrement $C[x]$ then set $i = C[x]$, then set $G[i] := F[x]$.

A crucial property of the algorithm is that it is *stable*: when we use it to sort with respect to a certain bitmask m and there is more than one element being mapped to the same value then the relative order between these elements is preserved.

```
Input                Counting sort wrt. two lowest bits
                       m = .....11
  0:   11111.11<       0:   ..i.i..
  1:   ...i...         1:   .iiii..
  2:   ..i.1.i         2:   .111...
  3:   ..1...1         3:   ..1.1.1
  4:   ..1.1111<       4:   .1..1..1
  5:   .1111..         5:   ..1...1.
  6:   .1..1..1        6:   .1.1.11.
  7:   .1.1.11         7:   11111.11<
  8:   .11..11<        8:   .1.1111<
  9:   .111....        9:   .11...11<
```

Note that relative order of the three words ending with two set bits (marked with '<') is preserved.

A routine that verifies whether an array is sorted with respect to a bit range specified by the variable `b0` and `m` is [FXT: sort/radixsort.cc]:

```
bool
is_counting_sorted(const ulong *f, ulong n, ulong b0, ulong m)
// Whether f[] is sorted wrt. bits b0,...,b0+z-1
//  where z is the number of bits set in m.
// m must contain a single run of bits starting at bit zero.
{
    m <<= b0;
    for (ulong k=1; k<n; ++k)
    {
        ulong xm = (f[k-1] & m ) >> b0;
        ulong xp = (f[k] & m ) >> b0;
        if ( xm>xp )  return false;
    }
    return true;
}
```

The function M is the combination of a mask-out and a shift operation. A routine that sorts according to `b0` and `m` is:

```
void
counting_sort_core(const ulong * restrict f, ulong n, ulong * restrict g, ulong b0, ulong m)
```

```
// Write to g[] the array f[] sorted wrt. bits b0,...,b0+z-1
//  where z is the number of bits set in m.
// m must contain a single run of bits starting at bit zero.
{
    ulong nb = m + 1;
    m <<= b0;
    ALLOCA(ulong, cv, nb);
    for (ulong k=0; k<nb; ++k)  cv[k] = 0;

    // --- count:
    for (ulong k=0; k<n; ++k)
    {
        ulong x = (f[k] & m ) >> b0;
        ++cv[ x ];
    }

    // --- cumulative sums:
    for (ulong k=1; k<nb; ++k)  cv[k] += cv[k-1];

    // --- reorder:
    ulong k = n;
    while ( k-- )  // backwards ==> stable sort
    {
        ulong fk = f[k];
        ulong x = (fk & m) >> b0;
        --cv[x];
        ulong i = cv[x];
        g[i] = fk;
    }
}
```

```
          Input          Stage 1          Stage 2          Stage 3
                       m = ....11       m = ..11..       m = 11....
                             vv               vv               vv
          111.11          ..1...           11....           ..1...
          ..1...          1111..           1...1.           ..1..1
          .1.1.1          11....           1...11           .1.1.1
          1...1.          .1.1.1           .1.1.1           .1.11.
          1.1111          ..1..1           .1.11.           1...1.
          1111..          1...1.           ..1...           1...11
          ..1..1          .1.11.           ..1..1           1.1111
          .1.11.          111.11           111.11           11....
          1...11          1.1111           1111..           111.11
          11....          1...11           1.1111           1111..
```

Figure 3.11-A: Radix sort of 10 six-bit values when using two-bit masks.

Now we can apply counting sort to a set of bit masks that cover the whole range. Figure 3.11-A shows an example with 10 six-bit values and 3 two-bit masks, starting from the least significant bits. This is the output of the program [FXT: sort/radixsort-demo.cc].

The routine [FXT: `radix_sort()` in sort/radixsort.cc] uses 8-bit masks to sort unsigned (long) integers:

```
void
radix_sort(ulong *f, ulong n)
{
    ulong nb = 8;  // Number of bits sorted with each step
    ulong tnb = BITS_PER_LONG; // Total number of bits

    ulong *fi = f;
    ulong *g = new ulong[n];

    ulong m = (1UL<<nb) - 1;
    for (ulong k=1, b0=0;  b0<tnb;  ++k, b0+=nb)
    {
        counting_sort_core(f, n, g, b0, m);
        swap2(f, g);
    }

    if ( f!=fi )  // result is actually in g[]
    {
        swap2(f, g);
        for (ulong k=0; k<n; ++k)  f[k] = g[k];
    }

    delete [] g;
}
```

There is room for optimization. Using two arrays for counting and combining copying with counting for the next pass where possible will reduce the number of passes almost by a factor of two.

A version of radix sort that starts from the most significant bits is given in [212].

3.12 Searching in unsorted arrays

We give an overview of the search functions for unordered arrays.

3.12.1 Minimal and maximal elements in unsorted arrays

Sorting an array only to determine the minimal (and/or maximal element) is obviously a bad idea because these can be found in linear time in the unsorted data.

Corresponding to all flavors of the sorting routines a `min()` and `max()` routine is supplied. We exemplify by giving the versions that determine the minimum.

Finding the minimum in an unsorted array corresponding to the 'plain' sorting routine: [FXT: sort/minmax.h]

```
template <typename Type>
Type inline min(const Type *f, ulong n)
// Return minimum of array.
{
    Type v = f[0];
    while ( n-- )  if ( f[n]<v )  v = f[n];
    return v;
}
```

The index version: [FXT: sort/minmaxidx.h]

```
template <typename Type>
Type idx_min(const Type *f, ulong n, const ulong *x)
// Return minimum (value) of array elements
//   {f[x[0]], f[x[1]], .. , f[x[n-1]]}
{
    Type v = f[x[0]];
    while ( n-- )  { if ( f[x[n]]<v )  v = f[x[n]]; }
    return  v;
}
```

The pointer version: [FXT: sort/minmaxptr.h]

```
template <typename Type>
Type ptr_min(/*const Type *f,*/ ulong n, Type const*const*x)
// Return minimum (value) of array elements
//   {*x[0], *x[1], .. , *x[n-1]}
{
    Type v = *x[0];
    while ( n-- )  { if ( *x[n]<v )  v = *x[n]; }
    return  v;
}
```

The comparison function version: [FXT: sort/minmaxfunc.h]

```
template <typename Type>
Type min(const Type *f, ulong n, int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
// wrt. to comparison function
{
    Type v = f[0];
    while ( n-- )  { if ( cmp(f[n],v) < 0 )  v = f[n]; }
    return  v;
}
```

The comparison function with index version: [FXT: sort/minmaxidxfunc.h]

```
template <typename Type>
Type idx_min(const Type *f, ulong n, const ulong *x,
             int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
//   {f[x[0]], f[x[1]], .. , f[x[n-1]]}
// with respect to comparison function cmp()
{
    Type v = f[x[0]];
    while ( n-- )  { if ( cmp(f[x[n]], v) < 0 )  v = f[x[n]]; }
    return  v;
}
```

The comparison function with pointer version: [FXT: sort/minmaxptrfunc.h]

```
template <typename Type>
Type ptr_min(/*const Type *f,*/ ulong n, Type const*const*x,
             int (*cmp)(const Type &, const Type &))
// Return minimum (value) of array elements
//   {*x[0], *x[1], .. , *x[n-1]}
// with respect to comparison function cmp().
{
    Type v = *x[0];
    while ( n-- )  { if ( cmp(*x[n],v)<0 )  v = *x[n]; }
    return  v;
}
```

3.12.2 Searching for values

To find the first occurrence of a certain value in an unsorted array one can use the routine [FXT: sort/usearch.h]

```
template <typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v
// Return n if all !=v
{
    ulong k = 0;
    while ( (k<n) && (f[k]!=v) )  k++;
    return k;
}
```

The functions `first_ne_idx()`, `first_ge_idx()` and `first_le_idx()` find the first occurrence of an element unequal (to `v`), greater or equal and less or equal, respectively.

If the last bit of speed matters, one could (see [153, p.267]) replace the shown code by

```
template <typename Type>
inline ulong first_eq_idx(const Type *f, ulong n, Type v)
// Return index of first element == v
// Return n if all !=v
{
    Type s = f[n-1];
    (Type *)f[n-1] = v;  // guarantee that the search stops
    ulong k = 0;
    while ( f[k]!=v )  k++;
    (Type *)f[n-1] = s;  // restore value
    if ( (k==n-1) && (v!=s) )  ++k;
    return k;
}
```

The speedup is due to the fact that there is only one branch in the inner loop. The technique is not applicable if the writes to the array '`f[]`' can have any side effects.

Replace the word `first` by last in the names to name the function that detect the last element satisfying the corresponding condition. For example,

```
template <typename Type>
inline ulong last_ge_idx(const Type *f, ulong n, Type v)
// Return index of last element >= v
// Return n if all <v
{
    ulong k = n-1;
```

```
    while ( f[k]<v )
    {
        k--;
        if ( 0==k )  return n;
    }
    return k;
}
```

3.12.3 Counting values

The functions in [FXT: sort/ucount.h] count certain elements in unordered arrays.

```
template <typename Type>
inline ulong eq_count(const Type *f, ulong n, Type v)
// Return number of elements that are ==v
{
    ulong ct = 0;
    while ( n-- )  if ( f[n]==v )  ++ct;
    return ct;
}
```

As above, replace `eq_` in the function name with `ne_`, `ge_` or `le_` to count the number of occurrences of elements that are unequal (to `v`), greater or equal and less or equal, respectively.

3.12.4 Searching matches

The routines in [FXT: sort/usearchfunc.h] generalize the functions from [FXT: sort/usearch.h]: A supplied function implements the condition imposed. For example,

```
template <typename Type>
inline ulong first_idx(const Type *f, ulong n, bool (* func)(Type))
// Return index of first element for which func() returns true.
// Return n if there is no such element.
{
    ulong k = 0;
    while ( (k<n) && (!func(f[k])) )  k++;
    return k;
}
```

and `last_idx()` that scans beginning from the greatest index.

The functions

```
template <typename Type>
inline ulong next_idx(const Type *f, ulong n, bool (* func)(Type), ulong k0)
// Like first_idx() but start from k0.
{
    ulong k = k0;
    while ( (k<n) && (!func(f[k])) )  k++;
    return k;
}
```

and `previous_idx()` determine the next matching element in forward or backward direction.

3.12.5 Selecting matches: grep

The routines from [FXT: sort/grep.h] count or select elements from an unordered array for which a condition implemented by a supplied function is true.

The following function counts the matching elements:

```
template <typename Type>
inline ulong count(const Type *f, ulong n, bool (* func)(Type))
// Return number of elements for which func() returns true.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  ct++;
    return  ct;
```

```
}
```

Discard all non-matches using

```
template <typename Type>
inline ulong grep(Type *f, ulong n, bool (* func)(Type))
// Delete elements for which func() returns false.
// Return number of elements kept.
{
    ulong k, j;
    for (k=0,j=0; j<n; ++k,++j)
    {
        f[k] = f[j];
        if ( func(f[j]) )  --k;
    }
    return k;
}
```

Record the values of the matches using

```
template <typename Type>
inline ulong grep(const Type *f, ulong n, bool (* func)(Type), Type *g)
// Make g[] the sequence of values for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  g[ct++] = f[k];
    return  ct;
}
```

Finally, recording the indices of the matches can be done with

```
template <typename Type>
inline ulong grep_idx(const Type *f, ulong n, bool (* func)(Type), ulong *x)
// Make x[] the sequence of indices for which func() returns true.
// Return number of 'matching' elements found.
{
    ulong ct = 0;
    for (ulong k=0; k<n; ++k)  if ( func(f[k]) )  x[ct++] = k;
    return  ct;
}
```

Chapter 4

Data structures

This chapter presents implementations of the operations on selected data structures like LIFO, FIFO, deque and heap.

4.1 Stack (LIFO)

A *stack* (or LIFO for last-in, first-out) is a data structure that supports the operations: *push* to saves an entry and *pop* to retrieve and remove the entry that was entered last. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it. Similarly, *poke* modifies the last entry. An implementation with the option to let the stack grow when necessary is [FXT: `class stack` in ds/stack.h]:

```
template <typename Type>
class stack
{
public:
    Type  *x_;  // data
    ulong  s_;  // size
    ulong  p_;  // stack pointer (position of next write), top entry @ p-1
    ulong  gq_; // grow gq elements if necessary, 0 for "never grow"
public:
    stack(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        p_ = 0;  // stack is empty
        gq_ = growq;
    }
    ~stack()  { delete [] x_; }
private:
    stack & operator = (const stack &);  // forbidden
public:
    ulong num()  const
    // return number of entries
    {
        return p_;
    }
    ulong push(Type z)
    // return size of stack, zero on stack overflow
    // if gq_ is nonzero the stack grows
    {
        if ( p_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // overflow
            grow();
        }
```

```
        x_[p_] = z;
        ++p_;
        return  s_;
    }

    ulong pop(Type &z)
    // read top entry and remove it
    // return number of entries at function start
    // if empty return zero and z is undefined
    {
        ulong ret = p_;
        if ( 0!=p_ )  { --p_;  z = x_[p_]; }
        return  ret;
    }

    ulong poke(Type z)
    // modify top entry
    // return number of entries
    // if empty return zero and nothing is done
    {
        if ( 0!=p_ )  x_[p_-1] = z;
        return p_;
    }

    ulong peek(Type &z)
    // read top entry (without removing it)
    // return number of entries
    // if empty return zero and z is undefined
    {
        if ( 0!=p_ )  z = x_[p_-1];
        return p_;
    }
   [--snip--]
```

The growth routine is implemented as

```
   [--snip--]
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        x_ = ReAlloc<Type>(x_, ns, s_);
        s_ = ns;
    }
};
```

here we used the function that imports the C function `realloc()`.

```
% man realloc
    #include <stdlib.h>

    void *realloc(void *ptr, size_t size);

    realloc() changes the size of the memory block pointed to  by  ptr  to  size
    bytes.   The  contents  will  be unchanged to the minimum of the old and new
    sizes; newly allocated memory will be uninitialized.  If ptr  is  NULL,  the
    call  is  equivalent  to malloc(size); if size is equal to zero, the call is
    equivalent to free(ptr).  Unless ptr is NULL, it must have been returned  by
    an earlier call to malloc(), calloc() or realloc().
```

Usage of the C function `realloc()` can be disabled globally in [FXT: realloc.h]:

```
#define USE_C_REALLOC  // comment out to disable use of realloc()

#ifdef USE_C_REALLOC
#include <cstdlib>  // realloc()
#endif

#ifdef USE_C_REALLOC
template <typename Type>
inline Type *ReAlloc(Type *p, ulong n, ulong /*nold*/)
{
    return (Type *)realloc((void *)p, n*sizeof(Type));
}

#else

template <typename Type>
inline Type *ReAlloc(Type *p, ulong n, ulong nold)
```

```
{
    Type *np = new Type[n];
    ulong nc = (nold < n ? nold : n);
    for (ulong k=0; k<nc; ++k)  np[k] = p[k];
    delete [] p;
    return np;
}
#endif
```

```
        push( 1)  1  -  -  -                 #=1
        push( 2)  1  2  -  -                 #=2
        push( 3)  1  2  3  -                 #=3
        push( 4)  1  2  3  4                 #=4
        push( 5)  1  2  3  4  5  -  -  -     #=5
        push( 6)  1  2  3  4  5  6  -  -     #=6
        push( 7)  1  2  3  4  5  6  7  -     #=7
        pop== 7   1  2  3  4  5  6  -  -     #=6
        pop== 6   1  2  3  4  5  -  -  -     #=5
        push( 8)  1  2  3  4  5  8  -  -     #=6
        pop== 8   1  2  3  4  5  -  -  -     #=5
        pop== 5   1  2  3  4  -  -  -  -     #=4
        push( 9)  1  2  3  4  9  -  -  -     #=5
        pop== 9   1  2  3  4  -  -  -  -     #=4
        pop== 4   1  2  3  -  -  -  -  -     #=3
        push(10)  1  2  3 10  -  -  -  -     #=4
        pop==10   1  2  3  -  -  -  -  -     #=3
        pop== 3   1  2  -  -  -  -  -  -     #=2
        push(11)  1  2 11  -  -  -  -  -     #=3
        pop==11   1  2  -  -  -  -  -  -     #=2
        pop== 2   1  -  -  -  -  -  -  -     #=1
        push(12)  1 12  -  -  -  -  -  -     #=2
        pop==12   1  -  -  -  -  -  -  -     #=1
        pop== 1   -  -  -  -  -  -  -  -     #=0
        push(13) 13  -  -  -  -  -  -  -     #=1
        pop==13   -  -  -  -  -  -  -  -     #=0
        pop== 0   -  -  -  -  -  -  -  -     #=0
          (stack was empty)
        push(14) 14  -  -  -  -  -  -  -     #=1
        pop==14   -  -  -  -  -  -  -  -     #=0
        pop== 0   -  -  -  -  -  -  -  -     #=0
          (stack was empty)
        push(15) 15  -  -  -  -  -  -  -     #=1
```

Figure 4.1-A: Inserting and retrieving elements with a stack.

A program that shows the working of the stack is [FXT: ds/stack-demo.cc]. An example output where the initial size is 4 and the growths-feature enabled (in steps of 4 elements) is shown in figure 4.1-A.

4.2 Ring buffer

A *ring buffer* is a array plus read- and write operations that wrap around. That is, if the last position of the array is reached writing continues at the begin of the array, thereby erasing the oldest entries. The read operation should start at the oldest entry in the array.

The implementation is [FXT: `class ringbuffer` in ds/ringbuffer.h]:

```
template <typename Type>
class ringbuffer
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong fpos_;  // first position to read in buffer
public:
```

```
    explicit ringbuffer(ulong n)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        wpos_ = 0;
        fpos_ = 0;
    }

    ~ringbuffer()  { delete [] x_; }

    ulong num()  const  { return n_; }

    void insert(const Type &z)
    {
        x_[wpos_] = z;
        if ( ++wpos_>=s_ )  wpos_ = 0;
        if ( n_ < s_ )  ++n_;
        else  fpos_ =  wpos_;
    }

    ulong read(ulong n, Type &z)  const
    // read entry n (that is, [(fpos_ + n)%s_])
    // return 0 if entry #n is last in buffer
    // else return n+1
    {
        if ( n>=n_ )  return 0;
        ulong j = fpos_ + n;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  n + 1;
    }
};
```

Reading from the ring buffer goes like:

```
    ulong k = 0;
    Type z;  // type of entry
    while ( (k = f.read(k, z)) )
    {
        // do something with z
    }
```

A program demonstrating the ring buffer is [FXT: ds/ringbuffer-demo.cc], its output is

```
    insert( 1)  1           #=1   r=1   w=0
    insert( 2)  1  2        #=2   r=2   w=0
    insert( 3)  1  2  3     #=3   r=3   w=0
    insert( 4)  1  2  3  4  #=4   r=0   w=0
    insert( 5)  2  3  4  5  #=4   r=1   w=1
    insert( 6)  3  4  5  6  #=4   r=2   w=2
    insert( 7)  4  5  6  7  #=4   r=3   w=3
    insert( 8)  5  6  7  8  #=4   r=0   w=0
    insert( 9)  6  7  8  9  #=4   r=1   w=1
```

Ring buffers can be useful for storing a constant amount of history-data such as for logging purposes. For that purpose one would enhance the `ringbuffer` class so that it uses an additional array of (fixed width) strings. The message to log would be copied into the array and the pointer set accordingly. A read should then just return the pointer to the string.

4.3 Queue (FIFO)

A *queue* (or FIFO for first-in, first-out) is a data structure that supports two operations: *push* saves an entry and *pop* retrieves (and removes) the entry that was entered the longest time ago. The occasionally useful operation *peek* retrieves the same element as pop but does not remove it.

A utility class with the optional feature of growing if necessary is [FXT: `class queue` in ds/queue.h]:

```
template <typename Type>
class queue
{
public:
```

```
    Type *x_;   // pointer to data
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong wpos_;  // next position to write in buffer
    ulong rpos_;  // next position to read in buffer
    ulong gq_;  // grow gq elements if necessary, 0 for "never grow"
public:
    explicit queue(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        wpos_ = 0;
        rpos_ = 0;
        gq_ = growq;
    }

    ~queue()  { delete [] x_; }

    ulong num()  const  { return n_; }

    ulong push(const Type &z)
    // Return number of entries.
    // Zero is returned on failure
    //   (i.e. space exhausted and 0==gq_)
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return 0;  // growing disabled
            grow();
        }

        x_[wpos_] = z;
        ++wpos_;
        if ( wpos_>=s_ )  wpos_ = 0;

        ++n_;
        return n_;
    }

    ulong peek(Type &z)
    // Return number of entries.
    // if zero is returned the value of z is undefined.
    {
        z = x_[rpos_];
        return n_;
    }

    ulong pop(Type &z)
    // Return number of entries before pop
    // i.e. zero is returned if queue was empty.
    // If zero is returned the value of z is undefined.
    {
        ulong ret = n_;
        if ( 0!=n_ )
        {
            z = x_[rpos_];
            ++rpos_;
            if ( rpos_ >= s_ )  rpos_ = 0;
            --n_;
        }

        return ret;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        // move read-position to zero:
        rotate_left(x_, s_, rpos_);
        x_ = ReAlloc<Type>(x_, ns, s_);
        wpos_ = s_;
        rpos_ = 0;
        s_ = ns;
    }
};
```

Its working is demonstrated by the program [FXT: ds/queue-demo.cc]. An example output where the

```
       push( 1)  1  -  -  -                 #=1   r=0   w=1
       push( 2)  1  2  -  -                 #=2   r=0   w=2
       push( 3)  1  2  3  -                 #=3   r=0   w=3
       push( 4)  1  2  3  4                 #=4   r=0   w=0
       push( 5)  1  2  3  4  5  -  -  -     #=5   r=0   w=5
       push( 6)  1  2  3  4  5  6  -  -     #=6   r=0   w=6
       push( 7)  1  2  3  4  5  6  7  -     #=7   r=0   w=7
       pop== 1   -  2  3  4  5  6  7  -     #=6   r=1   w=7
       pop== 2   -  -  3  4  5  6  7  -     #=5   r=2   w=7
       push( 8)  -  -  3  4  5  6  7  8     #=6   r=2   w=0
       pop== 3   -  -  -  4  5  6  7  8     #=5   r=3   w=0
       pop== 4   -  -  -  -  5  6  7  8     #=4   r=4   w=0
       push( 9)  9  -  -  -  5  6  7  8     #=5   r=4   w=1
       pop== 5   9  -  -  -  -  6  7  8     #=4   r=5   w=1
       pop== 6   9  -  -  -  -  -  7  8     #=3   r=6   w=1
       push(10)  9 10  -  -  -  -  7  8     #=4   r=6   w=2
       pop== 7   9 10  -  -  -  -  -  8     #=3   r=7   w=2
       pop== 8   9 10  -  -  -  -  -  -     #=2   r=0   w=2
       push(11)  9 10 11  -  -  -  -  -     #=3   r=0   w=3
       pop== 9   - 10 11  -  -  -  -  -     #=2   r=1   w=3
       pop==10   -  - 11  -  -  -  -  -     #=1   r=2   w=3
       push(12)  -  - 11 12  -  -  -  -     #=2   r=2   w=4
       pop==11   -  -  - 12  -  -  -  -     #=1   r=3   w=4
       pop==12   -  -  -  -  -  -  -  -     #=0   r=4   w=4
       push(13)  -  -  -  - 13  -  -  -     #=1   r=4   w=5
       pop==13   -  -  -  -  -  -  -  -     #=0   r=5   w=5
       pop== 0   -  -  -  -  -  -  -  -     #=0   r=5   w=5
         (queue was empty)
       push(14)  -  -  -  -  - 14  -  -     #=1   r=5   w=6
       pop==14   -  -  -  -  -  -  -  -     #=0   r=6   w=6
       pop== 0   -  -  -  -  -  -  -  -     #=0   r=6   w=6
         (queue was empty)
       push(15)  -  -  -  -  -  - 15  -     #=1   r=6   w=7
```

Figure 4.3-A: Inserting and retrieving elements with a queue.

initial size is 4 and the growths-feature enabled (in steps of 4 elements) is shown in figure 4.3-A. Compare to the corresponding figure for the stack, figure 4.1-A on page 143.

4.4 Deque (double-ended queue)

A *deque* (for *double-ended queue*) combines the data structures stack and queue: insertion and deletion is possible both at the first- and the last position, all in time-$O(1)$. An implementation with the option to let the stack grow when necessary is [FXT: `class deque` in ds/deque.h]

```
template <typename Type>
class deque
{
public:
    Type *x_;   // data (ring buffer)
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of entries in buffer
    ulong fpos_;  // position of first element in buffer
    // insert_first() will write to (fpos-1)%n
    ulong lpos_;  // position of last element in buffer plus one
    // insert_last() will write to  lpos,  n==(lpos-fpos) (mod s)
    // entries are at [fpos, ... , lpos-1]  (range may be empty)

    ulong  gq_; // grow gq elements if necessary, 0 for "never grow"
public:
    explicit deque(ulong n, ulong growq=0)
    {
        s_ = n;
        x_ = new Type[s_];
        n_ = 0;
        fpos_ = 0;
        lpos_ = 0;
```

```
    gq_ = growq;
}

~deque()  { delete [] x_; }

ulong num()  const  { return n_; }

ulong insert_first(const Type &z)
// returns number of entries after insertion
// zero is returned on failure
//   (i.e. space exhausted and 0==gq_)
{
    if ( n_ >= s_ )
    {
        if ( 0==gq_ )  return 0;  // growing disabled
        grow();
    }

    --fpos_;
    if ( fpos_ == -1UL )  fpos_ = s_ - 1;
    x_[fpos_] = z;
    ++n_;
    return  n_;
}

ulong insert_last(const Type &z)
// returns number of entries after insertion
// zero is returned on failure
//   (i.e. space exhausted and 0==gq_)
{
    if ( n_ >= s_ )
    {
        if ( 0==gq_ )  return 0;  // growing disabled
        grow();
    }

    x_[lpos_] = z;
    ++lpos_;
    if ( lpos_>=s_ )  lpos_ = 0;
    ++n_;
    return  n_;
}

ulong extract_first(Type & z)
// return number of elements before extract
// return 0 if extract on empty deque was attempted
{
    if ( 0==n_ )  return 0;
    z = x_[fpos_];
    ++fpos_;
    if ( fpos_ >= s_ )  fpos_ = 0;
    --n_;
    return  n_ + 1;
}

ulong extract_last(Type & z)
// return number of elements before extract
// return 0 if extract on empty deque was attempted
{
    if ( 0==n_ )  return 0;
    --lpos_;
    if ( lpos_ == -1UL )  lpos_ = s_ - 1;
    z = x_[lpos_];
    --n_;
    return  n_ + 1;
}

ulong read_first(Type & z)  const
// read (but don't remove) first entry
// return number of elements (i.e. on error return zero)
{
    if ( 0==n_ )  return 0;
    z = x_[fpos_];
    return n_;
}

ulong read_last(Type & z)  const
// read (but don't remove) last entry
```

```
    // return number of elements (i.e. on error return zero)
    {
        return  read(n_-1, z);  // ok for n_==0
    }
    ulong read(ulong k, Type & z)  const
    // read entry k (that is, [(fpos_ + k)%s_])
    // return 0 if k>=n_
    // else return k+1
    {
        if ( k>=n_ )  return 0;
        ulong j = fpos_ + k;
        if ( j>=s_ )  j -= s_;
        z = x_[j];
        return  k + 1;
    }
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        // Move read-position to zero:
        rotate_left(x_, s_, fpos_);
        x_ = ReAlloc<Type>(x_, ns, s_);
        fpos_ = 0;
        lpos_ = n_;
        s_ = ns;
    }
};
```

```
        insert_first( 1)        1
        insert_last(51)         1 51
        insert_first( 2)        2  1 51
        insert_last(52)         2  1 51 52
        insert_first( 3)        3  2  1 51 52
        insert_last(53)         3  2  1 51 52 53
        extract_first()= 3      2  1 51 52 53
        extract_last()= 53      2  1 51 52
        insert_first( 4)        4  2  1 51 52
        insert_last(54)         4  2  1 51 52 54
        extract_first()= 4      2  1 51 52 54
        extract_last()= 54      2  1 51 52
        extract_first()= 2      1 51 52
        extract_last()= 52      1 51
        extract_first()= 1     51
        extract_last()= 51
        insert_first( 5)        5
        insert_last(55)         5 55
        extract_first()= 5     55
        extract_last()= 55
        extract_first()= (deque is empty)
        extract_last()=  (deque is empty)
        insert_first( 7)        7
        insert_last(57)         7 57
```

Figure 4.4-A: Inserting and retrieving elements with a queue.

Its working is shown in figure 4.4-A which was created with the program [FXT: ds/deque-demo.cc].

4.5 Heap and priority queue

A *binary heap* is a binary tree where the left and right children are smaller than or equal to their parent node. The following function determines whether a given array is a heap [FXT: ds/heap.h]:

```
template <typename Type>
ulong test_heap(const Type *x, ulong n)
// return 0 if x[] has heap property
// else index of node found to be bigger than its parent
{
    const Type *p = x - 1;
    for (ulong k=n; k>1; --k)
```

```
    {
        ulong t = (k>>1); // parent(k)
        if ( p[t]<p[k] )  return k;
    }
    return 0;  // has heap property
}
```

It turns out that a unordered array of size n can be reordered to a heap in $O(n)$ time, see [89, p.145]. The routine

```
template <typename Type>
void build_heap(Type *x, ulong n)
// reorder data to a heap
{
    for (ulong j=(n>>1); j>0; --j)  heapify1(x-1, n, j);
}
```

does the trick. It uses the following subroutine that runs in time $O(\log n)$:

```
template <typename Type>
void heapify1(Type *z, ulong n, ulong k)
// subject to the condition that the trees below the children of node
// k are heaps, move the element z[k] (down) until the tree below node
// k is a heap.
//
// data expected in z[1..n] (!)
{
    ulong m = k;
    ulong l = (k<<1); // left(k);
    if ( (l <= n) && (z[l] > z[k]) )  m = l;
    ulong r = (k<<1) + 1; // right(k);
    if ( (r <= n) && (z[r] > z[m]) )  m = r;
    if ( m != k )
    {
        swap2(z[k], z[m]);
        heapify1(z, n, m);
    }
}
```

Heaps are useful to build a so-called *priority queue*. This is a data structure that supports insertion of an element and extraction of the maximal element it contains both in an efficient manner. A priority queue can be used to 'schedule' a certain event for a given point in time and return the next pending event.

A new element can be inserted into a heap in $O(\log(n))$ time by appending it and moving it towards the root (that is, the first element) as necessary:

```
bool heap_insert(Type *x, ulong n, ulong s, Type t)
// with x[] a heap of current size n
// and max size s (i.e. space for s elements allocated)
// insert t and restore heap-property.
//
// Return true if successful
// else (i.e. space exhausted) false
//
// Takes time \log(n)
{
    if ( n > s )  return false;
    ++n;
    Type *x1 = x - 1;
    ulong j = n;
    while ( j > 1 )
    {
        ulong k = (j>>1);  // k==parent(j)
        if ( x1[k] >= t )  break;
        x1[j] = x1[k];
        j = k;
    }
    x1[j] = t;
    return true;
}
```

Similarly, the maximal element can be removed in time $O(\log(n))$:

```
template <typename Type>
```

```
Type heap_extract_max(Type *x, ulong n)
// Return maximal element of heap and
// restore heap structure.
// Return value is undefined for 0==n
{
    Type m = x[0];
    if ( 0 != n )
    {
        Type *x1 = x - 1;
        x1[1] = x1[n];
        --n;
        heapify1(x1, n, 1);
    }
    // else error
    return m;
}
```

Two modifications seem appropriate: firstly, replacement of `extract_max()` by a `extract_next()`, leaving it as an (compile time) option whether to extract the minimal or the maximal element. This is achieved by changing the comparison operators at a few strategic places so that the heap is built either as described above or with its minimum as first element:

```
#if  1
// next() is the one with the smallest key
// i.e.  extract_next()  is  extract_min()
#define _CMP_ <
#define _CMPEQ_ <=
#else
// next() is the one with the biggest key
// i.e.  extract_next()  is  extract_max()
#define _CMP_ >
#define _CMPEQ_ >=
#endif
```

Secondly, augmenting the elements used by a event description that can be freely defined. [FXT: `class priority_queue` in ds/priorityqueue.h]:

```
template <typename Type1, typename Type2>
class priority_queue
{
public:
    Type1 *t1_;  // time:   t1[1..s]  one-based array!
    Type2 *e1_;  // events: e1[1..s]  one-based array!
    ulong s_;   // allocated size (# of elements)
    ulong n_;   // current number of events
    ulong gq_;  // grow gq elements if necessary, 0 for "never grow"

public:
    priority_queue(ulong n, ulong growq=0)
    {
        s_ = n;
        t1_ = new Type1[s_] - 1;
        e1_ = new Type2[s_] - 1;
        n_ = 0;
        gq_ = growq;
    }
    ~priority_queue()
    {
        delete [] (t1_+1);
        delete [] (e1_+1);
    }
    ulong num()  const  { return n_; }
    Type1 get_next_t()  const  { return t1_[1]; }
    Type2 get_next_e()  const  { return e1_[1]; }
    Type1 get_next(Type2 &e)  const
    // No check if empty
    {
        e = e1_[1];
        return t1_[1];
    }
    Type1 extract_next(Type2 &e)
    {
        Type1 m = get_next(e);
```

```
        if ( 0 != n_ )
        {
            t1_[1] = t1_[n_];  e1_[1] = e1_[n_];
            --n_;
            heapify1(1);
        }
        // else error
        return m;
    }

    bool insert(const Type1 &t, const Type2 &e)
    // Insert event e at time t
    // Return true if successful
    // else (i.e. space exhausted and 0==gq_) false
    {
        if ( n_ >= s_ )
        {
            if ( 0==gq_ )  return false;  // growing disabled
            grow();
        }
        ++n_;
        ulong j = n_;
        while ( j > 1 )
        {
            ulong k = (j>>1);  // k==parent(j)
            if ( t1_[k] _CMPEQ_ t )  break;
            t1_[j] = t1_[k];  e1_[j] = e1_[k];
            j = k;
        }
        t1_[j] = t;
        e1_[j] = e;
        return true;
    }
```

The member function `reschedule_next()` is more efficient than the sequence `extract_next(); insert();`, as it calls `heapify1()` only once:

```
    void reschedule_next(Type1 t)
    {
        t1_[1] = t;
        heapify1(1);
    }
```

The `heapify1()` function is tail recursive, so we make it iterative:

```
private:
    void heapify1(ulong k)
    {
        ulong m = k;
    hstart:
        ulong l = (k<<1); // left(k);
        ulong r = l + 1; // right(k);
        if ( (l <= n_) && (t1_[l] _CMP_ t1_[k]) )  m = l;
        if ( (r <= n_) && (t1_[r] _CMP_ t1_[m]) )  m = r;
        if ( m != k )
        {
            swap2(t1_[k], t1_[m]);  swap2(e1_[k], e1_[m]);
//            heapify1(m);
            k = m;
            goto hstart;  // tail recursion
        }
    }
   [--snip--]
```

The second argument of the constructor determines the number of elements added in case of growth, it is disabled (equals zero) by default.

```
   [--snip--]
private:
    void grow()
    {
        ulong ns = s_ + gq_;  // new size
        t1_ = ReAlloc<Type1>(t1_+1, ns, s_) - 1;
        e1_ = ReAlloc<Type2>(e1_+1, ns, s_) - 1;
        s_ = ns;
```

```
    }
};
```

The `ReAlloc()` routine is described in section 4.1 on page 141.

```
      Inserting into piority_queue:          Extracting from piority_queue:
       # :  event  @          time             # :  event  @          time
        0:      A  @      0.840188              9:      F  @      0.197551
        1:      B  @      0.394383              8:      I  @      0.277775
        2:      C  @      0.783099              7:      G  @      0.335223
        3:      D  @       0.79844              6:      B  @      0.394383
        4:      E  @      0.911647              5:      J  @       0.55397
        5:      F  @      0.197551              4:      H  @       0.76823
        6:      G  @      0.335223              3:      C  @      0.783099
        7:      H  @       0.76823              2:      D  @       0.79844
        8:      I  @      0.277775              1:      A  @      0.840188
        9:      J  @       0.55397              0:      E  @      0.911647
```

Figure 4.5-A: Insertion of events labeled '`A`', '`B`', ..., '`J`' scheduled for random times into a priority queue (left) and subsequent extraction (right).

The program [FXT: ds/priorityqueue-demo.cc] inserts events at random times $0 \le t < 1$, then extracts all of them. It gives the output shown in figure 4.5-A. A more typical usage would intermix the insertions and extractions.

4.6 Bit-array

The use of *bit-arrays* should be obvious: an array of tag values (like 'seen' versus 'unseen') where all standard data types would be a waste of space. Besides reading and writing individual bits one should implement a convenient search for the next set (or cleared) bit.

The class [FXT: `class bitarray` in ds/bitarray.h] is used, for example, for lists of small primes [FXT: mod/primes.cc], for in-place transposition routines [FXT: aux2/transpose.h] (see section 2.3 on page 89) and several operations on permutations (see section 2.11.8 on page 111).

```
class bitarray
// Bit-array class mostly for use as memory saving array of boolean values.
// Valid index is 0...nb_-1 (as usual in C arrays).
{
public:
    ulong *f_;   // bit bucket
    ulong n_;    // number of bits
    ulong nfw_;  // number of words where all bits are used, may be zero
    ulong mp_;   // mask for partially used word if there is one, else zero
    // (ones are at the positions of the _unused_ bits)
    bool myfq_;  // whether f[] was allocated by class
 [--snip--]
```

The constructor allocates memory by default. If the second argument is nonzero it must point to an accessible memory range:

```
    bitarray(ulong nbits, ulong *f=0)
    // nbits must be nonzero
    {
        ulong nw = ctor_core(nbits);
        if ( f!=0 )
        {
            f_ = (ulong *)f;
            myfq_ = false;
        }
        else
        {
            f_ = new ulong[nw];
            myfq_ = true;
        }
    }
```

The public methods are

```
// operations on bit n:
    ulong test(ulong n)  const // Test whether n-th bit set
    void set(ulong n)          // Set n-th bit
    void clear(ulong n)        // Clear n-th bit
    void change(ulong n)       // Toggle n-th bit
    ulong test_set(ulong n)    // Test whether n-th bit is set and set it
    ulong test_clear(ulong n)  // Test whether n-th bit is set and clear it
    ulong test_change(ulong n) // Test whether n-th bit is set and toggle it

// Operations on all bits:
    void clear_all()           // Clear all bits
    void set_all()             // Set all bits
    int all_set_q()  const;    // Return whether all bits are set
    int all_clear_q()  const;  // Return whether all bits are clear

// Scanning the array:
    // Note: the given index n is included in the search
    ulong next_set_idx(ulong n)  const   // Return index of next set or value beyond end
    ulong next_clear_idx(ulong n)  const // Return index of next clear or value beyond end
```

Combined operations like 'test-and-set-bit', 'test-and-clear-bit', 'test-and-change-bit' are often needed in applications that use bit-arrays. This is underlined by the fact that modern CPUs typically have instructions implementing these operations.

The class does not supply overloading of the array-index operator `[]` because the writing variant would cause a performance penalty. One might want to add 'sparse'-versions of the scan functions (`next_set_idx()` and `next_clear_idx()`) for large bit-arrays with only few bits set or unset.

On the AMD64 architecture the corresponding CPU instructions are used [FXT: bits/bitasm-amd64.h]:

```
static inline ulong asm_bts(ulong *f, ulong i)
// Bit Test and Set
{
    ulong ret;
    asm ( "btsq %2, %1 \n"
          "sbbq %0, %0"
          : "=r" (ret)
          : "m" (*f), "r" (i) );
    return ret;
}
```

If no specialized CPU instructions are available the following two macros are used:

```
#define  DIVMOD(n, d, bm) \
ulong d = n / BITS_PER_LONG; \
ulong bm = 1UL << (n % BITS_PER_LONG);

#define  DIVMOD_TEST(n, d, bm) \
ulong d = n / BITS_PER_LONG; \
ulong bm = 1UL << (n % BITS_PER_LONG); \
ulong t = bm & f_[d];
```

An example:

```
    ulong test_set(ulong n)  // Test whether n-th bit is set and set it.
    {
#ifdef  BITS_USE_ASM
        return asm_bts(f_, n);
#else
        DIVMOD_TEST(n, d, bm);
        f_[d] |= bm;
        return  t;
#endif
    }
```

Performance is still good in that case as the modulo operation and division by `BITS_PER_LONG` (a power of two) are replaced with cheap (bit-and and shift) operations. On the machine described in appendix A on page 883 both versions give practically identical performance.

How out of bounds are handled can be defined at the beginning of the header file:

```
#define CHECK  0  // define to disable check of out of bounds access
//#define CHECK  1  // define to handle out of bounds access
//#define CHECK  2  // define to fail with out of bounds access
```

4.7 Finite-state machines

A *finite-state machine* (FSM) (or *state-engine*) in its simplest form can be described as a program that has finite set of valid states and for each state a certain action is taken. In C-syntax:

```
void state_engine(int state)
{
    while ( state != end )
    {
        // valid states are: st1 ... stn (and end)
        switch ( state )
        {
        case st1:  state = func1();  break;
        case st2:  state = func2();  break;
        case st3:  state = func3();  break;
        [--snip--]
        case stn:  state = funcn();  break;
        default:   blue_smoke();  // invalid state
        }
    }
}

int main()
{
    // initialize:
    int state = start;
    state_engine( state );
    return 0;
}
```

As an example we show a state engine that transforms a linear coordinate t into the corresponding pair (x and y) of coordinates of Hilbert's space-filling curve.

Apart from two bits of internal state the FSM processes at each step two bits of input. The array `htab[]` serves as lookup table for the next state plus two bits of the result.

The function ([FXT: `lin2hilbert()` in bits/hilbert.cc]) implements a FSM as suggested in [30], item 115:

```
void
lin2hilbert(ulong t, ulong &x, ulong &y)
// Transform linear coordinate t to Hilbert x and y
{
    ulong xv = 0, yv = 0;
    ulong c01 = (0<<2);  // (2<<2) for transposed output (swapped x, y)
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        ulong abi = t >> (BITS_PER_LONG-2);
        t <<= 2;

        ulong st = htab[ (c01<<2) | abi ];
        c01 = st & 3;

        yv <<= 1;
        yv |= ((st>>2) & 1);
        xv <<= 1;
        xv |= (st>>3);
    }
    x = xv;  y = yv;
}
```

The table used is defined (see figure 4.7-A) as

```
static const ulong htab[] = {
#define HT(xi,yi,c0,c1) ((xi<<3)+(yi<<2)+(c0<<1)+(c1))
    // index == HT(c0,c1,ai,bi)
    HT( 0, 0,  1, 0 ),
    HT( 0, 1,  0, 0 ),
    HT( 1, 1,  0, 0 ),
    HT( 1, 0,  0, 1 ),
  [--snip--]
    HT( 0, 0,  1, 1 ),
    HT( 0, 1,  1, 0 )
};
```

As indicated in the code the table maps every four bits `c0,c1,ai,bi` to four bits `xi,yi,c0,c1`. The table for the inverse function (again, see figure 4.7-A) is

OLD				NEW				NEW				OLD			
C	C	A	B	X	Y	C	C	C	C	X	Y	A	B	C	C
0	1	I	I	I	I	0	1	0	1	I	I	I	I	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0
0	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1
0	0	1	1	1	0	0	1	0	0	1	1	1	0	0	0
0	1	0	0	1	1	1	1	0	1	0	0	1	0	0	1
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	1	0	0	0	0	1	0	1	1	0	1	1	0	0
0	1	1	1	1	0	0	0	0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	1	0	1	0	1	0	0	1	1	1	1	1
1	0	1	0	1	1	1	0	1	0	1	0	0	1	1	0
1	0	1	1	0	1	1	1	1	0	1	1	1	0	1	0
1	1	0	0	1	1	0	1	1	1	0	0	1	0	1	1
1	1	0	1	1	0	1	1	1	1	0	1	1	1	1	0
1	1	1	0	0	0	1	1	1	1	1	0	0	1	1	1
1	1	1	1	0	1	1	0	1	1	1	1	0	0	0	1

Figure 4.7-A: The original table from [30] for the finite state machine for the 2-dimensional Hilbert curve (left). All sixteen 4-bit words appear in both the '`OLD`' and the '`NEW`' column. So the algorithm is invertible. Swap the columns and sort numerically to obtain the two columns at the right, the table for the inverse function.

```
static const ulong ihtab[] = {
#define IHT(ai,bi,c0,c1) ((ai<<3)+(bi<<2)+(c0<<1)+(c1))
    // index == HT(c0,c1,xi,yi)
    IHT( 0, 0,  1, 0 ),
    IHT( 0, 1,  0, 0 ),
    IHT( 1, 1,  0, 1 ),
    IHT( 1, 0,  0, 0 ),
  [--snip--]
    IHT( 0, 1,  1, 1 ),
    IHT( 0, 0,  0, 1 )
};
```

The words have to be processed backwards:

```
ulong
hilbert2lin(ulong x, ulong y)
// Transform Hilbert x and y to linear coordinate t
{
    ulong t = 0;
    ulong c01 = 0;
    for (ulong i=0; i<(BITS_PER_LONG/2); ++i)
    {
        t <<= 2;
        ulong xi = x >> (BITS_PER_LONG/2-1);
        xi &= 1;
        ulong yi = y >> (BITS_PER_LONG/2-1);
        yi &= 1;
        ulong xyi = (xi<<1) | yi;
        x <<= 1;
        y <<= 1;

        ulong st = ihtab[ (c01<<2) | xyi ];
        c01 = st & 3;

        t |= (st>>2);
    }
    return t;
}
```

A method to compute the direction (left, right, up or down) at the n-th move of the Hilbert curve is given in section 1.19 on page 51. The computation of a function whose series coefficients are ± 1 and $\pm i$ according to the Hilbert curve is described in section 36.9 on page 712.

4.8 Emulation of coroutines

Sometimes it is possible to find an recursive algorithm for solving some problem that is not easily solved iteratively. However the recursive implementations might produce the results in midst of its calling graph. When a utility class providing the results one by one with some `next` call is required there is an apparent problem (with most programming languages): there is only one stack available for function calls. We do not have 'offline' functions or *coroutines*.

As an example consider the following recursive code (given by Glenn Rhoads)

```
int n = 4;
int a[n+1], q[2*n+1];

int main()
{
    paren(0, 0);
    return 0;
}

void paren(long i, long s)
{
    long k, t;
    if ( i<n )
    {
        for (k=0; k<=i-s;  ++k)
        {
            a[i-1] = k;
            t = s + a[i-1];
            q[t + i] = '(';
            paren(i + 1, t);  // recursion
            q[t + i] = ')';
        }
    }
    else
    {
        a[i-1] = n - s;
        Visit();  // next set of parens available
    }
}
```

It generates following output (the different ways to group four pairs of parenthesis):

```
    ((()))
    (()())
    (())()
    ()(())
    ()()()
```

A reasonable way to create coroutines is to rewrite the function as a state engine and use a class [FXT: `class coroutine` in ds/coroutine.h] that provides two stacks, one for local variables (including arguments) and one for the state of the function:

```
template <typename Type>
class coroutine
{
public:
    ulong *t_;  // sTate stack
    ulong tp_;  // sTate stack Pointer
    Type  *d_;  // Data stack
    ulong dp_;  // Data stack Pointer
    ulong  n_;  // size of stacks

public:
    coroutine(ulong maxdepth)
    {
        n_ = maxdepth;
```

```
        t_ = new ulong[n_];
        d_ = new Type[n_];
        init();
    }

    ~coroutine()
    {
        delete [] d_;
        delete [] t_;
    }

    void init()  { dp_=0; tp_=0; }

    // --- state stack:
    void stpush(ulong x)   { chk_tp();  t_[tp_] = x;  ++tp_; }
    ulong stpeek()  const  { chk_tp(1);  return  t_[tp_-1]; }
    void stpoke(ulong x)   { chk_tp(1);  t_[tp_-1] = x; }
    void stpop(ulong ct=1)  { tp_-=ct; }
    void stnext()  { chk_tp(1);  ++t_[tp_-1]; }
    void stnext(ulong x)  { chk_tp(1);  t_[tp_-1] = x; }
    bool more()  const  { return (tp_!=0); }

    // --- stack for variables and args:
    void push(Type x)  {  chk_dp();  d_[dp_] = x;  ++dp_; }
    Type &peek()  { chk_dp(1); return d_[dp_-1]; }
    void pop(ulong ct=1) { dp_-=ct; }

private:
    coroutine & operator = (const coroutine &);  // forbidden
};
```

The functions `chk_tp()` and`chk_dp()` either check for stack overflow or do nothing, as defined by

```
#define CHECK_STACKS  // define to detect stack overflow
```

Rewriting the function in question (as part of a utility class, given in [FXT: ds/coroutine-paren-demo.cc]) only requires the understanding of the language, not of the algorithm. The process is straightforward but needs a bit of concentration:

```
int
paren::next_recursion()
{
 redo:
    vars &V = cr_->peek();
    int &i=V.i,  &s=V.s; // args
    int &k=V.k,  &t=V.t; // locals

 loop:
    switch ( cr_->stpeek() )
    {
    case 0:
        if ( i>=n )
        {
            x[i-1] = n - s;
            cr_->stnext( ST_RETURN );  return 1;
        }
        cr_->stnext();

    case 1:
        // loop end ?
        if ( k>i-s )  { break; }  // shortcut: nothing to do at end

        cr_->stnext();

    case 2: // start of loop body
        {
            long ii = i - 1;
            x[ii] = k;
            t = s + k;

            str[t+i] = '('; // OPEN_CHAR;

            cr_->stnext();
            CORO_CALL( i+1, t, 0, 0 );
            goto redo;
        }

    case 3:
        str[t+i] = ')'; // CLOSE_CHAR;
        ++k;

        // loop end ?
```

```
        if ( k>i-s )  { break; }  // shortcut: nothing to do at end
        cr_->stpoke(2);  goto loop; // shortcut: back to loop body
    default: ;
    }
    CORO_RETURN();
    if ( cr_->more() )  goto redo;
    return 0;
}
```

The following `#define` was used:

```
//           args=(i, s)  (k, t)=locals
#define CORO_CALL(vi, vs, vk, vt) \
 { vars V_; V_.i=vi; V_.s=vs; V_.k=vk; V_.t=vt; \
   cr_->push(V_);  cr_->stpush(ST_INIT); }
```

The constructor pushes the needed variables and parameters on the data stack and the initial state on the state stack:

```
paren::paren(int nn)
{
    n = (nn>0 ? nn : 1);
    x = new int[n+1];
    ++x;
    str = new char[2*n+1];
    int i = 0;
    for (  ; i<n  ; ++i)  str[i] = '('; // OPEN_CHAR;
    for (  ; i<2*n; ++i)  str[i] = ')'; // CLOSE_CHAR;
    str[2*n] = 0;
    cr_ = new coroutine<vars>(n+1);
    //         i, s, k, t
    CORO_CALL( 0, 0, 0, 0 );
    idx = 0;
    q = next_recursion();
}
```

The method `next()` of the paren class lets the offline function advance until the next result is available:

```
int paren::next()
{
    if ( 0==q )  return 0;
    else
    {
        q = next_recursion();
        return  ( q ? ++idx : 0 );
    }
}
```

Performance-wise the `coroutine`-rewritten functions are close to the original: state engines are fast and the stack operations are cheap.

The shown method can also be applied when the recursive algorithm consists of more than one function by merging the functions into one state engine.

Further, investigating the contents of the data stack can be of help in the search of a iterative solution.

The code of the converted functions is admittedly ugly but without a generic support of coroutines it seems hard to du much better. One could also use threads for the emulation of coroutines. This approach, however, adds portability problems.

An iterative algorithm for the generation of valid pairings or parenthesis is given in chapter 13 on page 299.

Part II

Combinatorial generation

Chapter 5

Conventions and considerations

We give algorithms for the generation of all combinatorial objects of certain types such as for the generation of combinations, compositions, subsets, permutations, integer partitions, set partitions, restricted growth strings and necklaces. Finally, we give some constructions for Hadamard and conference matrices. Several (more esoteric) combinatorial objects that are found via searching in directed graphs are presented in chapter 19.

The routines are useful in situations where an exhaustive search over all configurations of a certain kind is needed. Combinatorial algorithms are also fundamental with many programming problems and finally they can simply be fun!

5.1 About representations and orders

For a set of n elements we will always take either $\{0, 1, \ldots, n-1\}$ or $\{1, 2, \ldots, n\}$. Our convention for the set notation is to always start with the smallest element. Often there is more than one useful way to represent a combinatorial object. For example the subset $\{1, 4, 6\}$ of the set $\{0, 1, 2, 3, 4, 5, 6\}$ can also be written as a *delta set* `[0100101]`. Some sources use the term *bit string*. We often write dots for zeros for readability: `[.1..1.1]`. Note that in the delta set we put the first element to the left side (*array notation*), this is in contrast to the usual way to print binary numbers, where the least significant bit (bit number zero) is shown on the right side.

For most objects we will give an algorithm for generation in *lexicographic* (or simply *lex*) order. In lexicographic order a string $X = [x_0, x_1, \ldots]$ precedes another string $Y = [y_0, y_1, \ldots]$ if for the smallest index k where the strings differ we have $x_k < y_k$. Further, the string X precedes $X.W$ (the concatenation of X with W) for any nonempty string W. The ordering obtained by reversing the lexicographic order is sometimes called *relex* order. The *co-lexicographic* (or simply *colex*) order is obtained by the lex order of the reversed strings. The order sometimes depends on the representation that is used, see figure 8.1-A on page 191.

In a *minimal-change* order the amount of change between successive objects is the least possible. Such an order is also called a (combinatorial) *Gray code*. There is in general more than one such order. Often one can impose even stricter conditions, like that (with permutations) the changes are between neighboring positions. The corresponding order is a *strong minimal-change order*. A very readable survey of Gray codes in given in [235], see also [206].

5.2 Ranking, unranking, and counting

For a particular ordering of combinatorial objects (say, lexicographic order for permutations) one can ask which position in the list a given object has. An algorithm for finding the position is called a *ranking* algorithm. An method to determine the object, given its position, is called an *unranking* algorithm.

Given both ranking and unranking methods one can compute the successor of a given object by computing its rank r and unranking $r + 1$. While this method is usually slow the idea can be used to find more efficient algorithms for computing the successor. In addition the idea often suggests interesting orderings for combinatorial objects.

We sometimes give ranking or unranking methods for numbers in special forms such as factorial representations for permutations. Ranking and unranking methods are implicit in generation algorithms based on mixed radix counting given in section 10.8 on page 244.

A simple but surprisingly powerful way for the discovery of isomorphisms (one-to-one correspondences) between combinatorial objects is counting them. If the sequences of numbers of two kinds of objects are identical chances are good to find a conversion routine between the corresponding objects. For example, there are 2^n permutations of n elements such that no element lies more than one position to the right of its original position. From this observation an algorithm for generating these permutations via binary counting can be found, see section 10.15.2 on page 270.

The representation of combinatorial objects as restricted growth strings (as shown in section 13.2 on page 301) follows from the same idea. The resulting generation methods can be very fast and flexible.

Recursive relations for the number of objects often lead to recursive generation algorithms (see, for example, section 6.3 on page 170 and section 11.2.1 on page 277).

5.3 Characteristics of the algorithms

In almost all cases we produce the combinatorial objects on by one. Let n be the size of the object. The successor (with respect to the specified order) is computed from the object itself and additional data whose size is less than a constant multiple of n.

Let B be the total number of combinatorial objects under consideration. Sometimes the cost of a successor computation is proportional to n. Then the total cost for generating all objects is proportional to $n \cdot B$.

If the successor computation takes a fixed number of operations (independent of the object size) then we say the algorithm is $O(1)$. If so, there can be no loop in the implementation, we say the algorithm is *loopless*. The total cost for all objects then is $c \cdot B$ for some c independent of the object size. A loopless algorithm can only exist is the amount of change between successive object is bounded by a constant that does not depend on the object size. Natural candidates for loopless algorithms are Gray codes.

In many cases the cost of computing all objects is also $c \cdot B$ while the computation of the successor does involve a loop. As an example consider incrementing in binary using arrays: in half of the cases just the lowest bit changes, for half of the remaining cases just two bits change, and so on. The total cost is $B \cdot (1 + \frac{1}{2}(1 + \frac{1}{2}(\cdots))) = 2 \cdot B$, independent of the number of bits used. So the total cost is as in the loopless case while the successor computation can be expensive in some cases. Algorithms with this characteristic are called *constant amortized time* (or CAT). Often CAT algorithms are faster than loopless algorithms, typically when their structure is more simple.

5.4 Optimization techniques

Let `x` be an array of n elements. The loop

```
ulong k = 0;
while ( (k<n) && (x[k]!=0) )  ++k;  // find first zero
```

can be replaced by

```
ulong k = 0;
while ( x[k]!=0 )  ++k;  // find first zero
```

if a single *sentinel* element `x[n]=0` is appended to the end of the array. The latter version will often be faster as less branches occur.

In some cases we replace a pointer as in

```
class foo {
    ulong *x;
    foo(ulong n)  { x = new ulong[n]; /*...*/ }
```

by an array

```
class foo {
    ulong x[32];  // works for n<=32 only
    foo(ulong n)  { /*...*/ }
```

The latter version can be faster as the memory address can often be generated more cheaply. This optimization applies to most algorithms for the generation of permutations.

The test for equality as in

```
ulong k = 0;
while ( k!=n )  { /*...*/  ++k; }
```

is more expensive than the test for equality with zero as in

```
ulong k = n;
while ( --k!=0 )  { /*...*/ }
```

Therefore the latter version should be used when applicable.

To reduce the number of branches, replace two tests

```
if ( (x<0) || (x>m) )  { /*...*/ }
```

by the following single test where unsigned integers are used:

```
if ( x>m )  { /*...*/ }
```

Use a do-while construct instead of a while-do loop whenever possible because the latter also tests the loop condition at entry. Even when the do-while version causes some unnecessary work the gain from the avoided branch may outweigh it. Note that in the C language the for-loop also tests the condition at loop entry.

When computing the next object there may be special cases where the update is trivial. If the percentage of these 'easy cases' is not too small an extra branch in the update routine should be created. The performance gain is very visible in most cases (section 10.4 on page 231) and can be drastic (section 10.5 on page 234).

Recursive routines can very be elegant and versatile. However, expect only about half the speed of a good iterative implementation of the same algorithm. The ideas given in section 4.8 on page 156 can be used to obtain iterative versions from a recursive implementation.

Approximate timings for the routines are given with most implementations. These were obtained with the referenced demo-programs, simply uncomment the line

```
//#define TIMING // uncomment to disable printing
```

near the top of the program file for your own measurements. The parameters (program arguments) used for the timing are given near the end of the file. When a generator routine is used in an application one must do the benchmarking with the application. Further optimization will very likely involve the surrounding code rather than the generator alone. The rate of generation for a certain object is occasionally given as 123 M/s, meaning that 123 million objects are generated per second.

Choosing the optimal ordering and type of representation (for example, delta sets versus sets) for the given task is crucial.

Address generation can be simpler if arrays are used instead of pointers. The speedup gained can be spectacular, see section 7.1 on page 184.

5.5 Remarks about the C++ implementations

The iterative methods are implemented as C++ classes. The first object in the given order is created by the method `first()`. The method to compute the successor is usually `next()`. If a method for the computation of the predecessor is given then it is called `prev()`, and the method `last()` computes the last element in the list.

The combinatorial object can be accessed through the method `data()`. In order to make all data of a class accessible the data is declared `public`. Thereby the need for various `get_something()` methods is avoided. To minimize the danger of accidental modification of class data the variable names end with an underscore. For example, the class for the generation of combinations in lexicographic order starts as

```
class combination_lex
{
public:
    ulong *x_;    // combination: k elements 0<=x[j]<k in increasing order
    ulong n_, k_; // Combination (n choose k)
```

The methods for the user of the class are `public`, the internal methods are declared `private`.

Chapter 6

Combinations

n \ k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0:	1															
1:	1	1														
2:	1	2	1													
3:	1	3	3	1												
4:	1	4	6	4	1											
5:	1	5	10	10	5	1										
6:	1	6	15	20	15	6	1									
7:	1	7	21	35	35	21	7	1								
8:	1	8	28	56	70	56	28	8	1							
9:	1	9	36	84	126	126	84	36	9	1						
10:	1	10	45	120	210	252	210	120	45	10	1					
11:	1	11	55	165	330	462	462	330	165	55	11	1				
12:	1	12	66	220	495	792	924	792	495	220	66	12	1			
13:	1	13	78	286	715	1287	1716	1716	1287	715	286	78	13	1		
14:	1	14	91	364	1001	2002	3003	3432	3003	2002	1001	364	91	14	1	
15:	1	15	105	455	1365	3003	5005	6435	6435	5005	3003	1365	455	105	15	1

Figure 6.0-A: The binomial coefficients $\binom{n}{k}$ for $0 \le n, k \le 15$.

The number of ways to choose k elements from a set of n elements equals the binomial coefficient ('n choose k'):

$$\binom{n}{k} \;=\; \frac{n!}{k!\,(n-k)!} \;=\; \frac{\prod_{j=1}^{k}(n-j+1)}{k!} \tag{6.0-1}$$

Equivalently, a set of n elements has $\binom{n}{k}$ subsets of exactly k elements. These subsets are called the *k-subsets* (where k is fixed) or *k-combinations* of an n-set.

To avoid overflow during the computation of the binomial coefficient, use the form

$$\binom{n}{k} \;=\; \frac{(n-k+1)^{\overline{k}}}{1^{\overline{k}}} \;=\; \frac{n-k+1}{1}\cdot\frac{n-k+2}{2}\cdot\frac{n-k+3}{3}\cdots\frac{n}{k} \tag{6.0-2}$$

An implementation is given in [FXT: aux0/binomial.h]:

```
inline ulong binomial(ulong n, ulong k)
{
    if ( k>n )  return  0;
    if ( (k==0) || (k==n) )  return  1;
    if ( 2*k > n )  k = n-k;  // use symmetry

    ulong b = n - k + 1;
    ulong f = b;
    for (ulong j=2; j<=k; ++j)
    {
        ++f;
        b *= f;
```

```
        b /= j;
    }
    return  b;
}
```

The table of the first binomial coefficients is shown in figure 6.0-A. This table is called *Pascal's triangle*, it was generated with the program [FXT: comb/binomial-demo.cc]. Observe that

$$\binom{n}{k} \;=\; \binom{n-1}{k-1} + \binom{n-1}{k} \tag{6.0-3}$$

That is, each entry is the sum of its upper and left upper neighbor.

6.1 Lexicographic and co-lexicographic order

	lexicographic		co-lexicographic		
	set	delta set	set	delta set	set reversed
1:	{ 0, 1, 2 }	111...	{ 0, 1, 2 }	111...	{ 2, 1, 0 }
2:	{ 0, 1, 3 }	11.1..	{ 0, 1, 3 }	11.1..	{ 3, 1, 0 }
3:	{ 0, 1, 4 }	11..1.	{ 0, 2, 3 }	1.11..	{ 3, 2, 0 }
4:	{ 0, 1, 5 }	11...1	{ 1, 2, 3 }	.111..	{ 3, 2, 1 }
5:	{ 0, 2, 3 }	1.11..	{ 0, 1, 4 }	11..1.	{ 4, 1, 0 }
6:	{ 0, 2, 4 }	1.1.1.	{ 0, 2, 4 }	1.1.1.	{ 4, 2, 0 }
7:	{ 0, 2, 5 }	1.1..1	{ 1, 2, 4 }	.11.1.	{ 4, 2, 1 }
8:	{ 0, 3, 4 }	1..11.	{ 0, 3, 4 }	1..11.	{ 4, 3, 0 }
9:	{ 0, 3, 5 }	1..1.1	{ 1, 3, 4 }	.1.11.	{ 4, 3, 1 }
10:	{ 0, 4, 5 }	1...11	{ 2, 3, 4 }	..111.	{ 4, 3, 2 }
11:	{ 1, 2, 3 }	.111..	{ 0, 1, 5 }	11...1	{ 5, 1, 0 }
12:	{ 1, 2, 4 }	.11.1.	{ 0, 2, 5 }	1.1..1	{ 5, 2, 0 }
13:	{ 1, 2, 5 }	.11..1	{ 1, 2, 5 }	.11..1	{ 5, 2, 1 }
14:	{ 1, 3, 4 }	.1.11.	{ 0, 3, 5 }	1..1.1	{ 5, 3, 0 }
15:	{ 1, 3, 5 }	.1.1.1	{ 1, 3, 5 }	.1.1.1	{ 5, 3, 1 }
16:	{ 1, 4, 5 }	.1..11	{ 2, 3, 5 }	..11.1	{ 5, 3, 2 }
17:	{ 2, 3, 4 }	..111.	{ 0, 4, 5 }	1...11	{ 5, 4, 0 }
18:	{ 2, 3, 5 }	..11.1	{ 1, 4, 5 }	.1..11	{ 5, 4, 1 }
19:	{ 2, 4, 5 }	..1.11	{ 2, 4, 5 }	..1.11	{ 5, 4, 2 }
20:	{ 3, 4, 5 }	...111	{ 3, 4, 5 }	...111	{ 5, 4, 3 }

Figure 6.1-A: All combinations $\binom{6}{3}$ in lexicographic order (left), and co-lexicographic order (right).

The combinations of three elements out of six in *lexicographic* (or simply *lex*) order are shown in figure 6.1-A (left). The sequence is such that the sets are ordered lexicographically. Note that for the delta sets the element zero is printed first whereas with binary words (section 1.25 on page 61) the least significant bit (bit zero) is printed last. The sequence for *co-lexicographic* (or *colex*) order is such that the sets, when written reversed, are ordered lexicographically.

6.1.1 Lexicographic order

The C++ class [FXT: `class combination_lex` in comb/combination-lex.h] generates the sets:

```
class combination_lex
{
public:
    ulong *x_;    // combination: k elements 0<=x[j]<k in increasing order
    ulong n_, k_; // Combination (n choose k)

public:
    combination_lex(ulong n, ulong k)
    {
        n_ = n;  k_ = k;
        x_ = new ulong[k_];
```

```
    first();
}

~combination_lex()  { delete [] x_; }

void first()
{
    for (ulong k=0; k<k_; ++k)  x_[k] = k;
}

void last()
{
    for (ulong i=0; i<k_; ++i)  x_[i] = n_ - k_ + i;
}
```

Computation of the successor and predecessor:

```
ulong next()
// Return smallest position that changed, return k with last combination
{
    if ( x_[0] == n_ - k_ )  // current combination is the last
    { first();  return k_; }

    ulong j = k_ - 1;
    // easy case:  highest element != highest possible value:
    if ( x_[j] < (n_-1) )  { ++x_[j];  return j; }

    // find highest falling edge:
    while ( 1 == (x_[j] - x_[j-1]) )  { --j; }

    // move lowest element of highest block up:
    ulong ret = j - 1;
    ulong z = ++x_[j-1];

    // ... and attach rest of block:
    while ( j < k_ )  { x_[j] = ++z;  ++j; }

    return  ret;
}

ulong prev()
// Return smallest position that changed, return k with last combination
{
    if ( x_[k_-1] == k_-1 )  // current combination is the first
    { last();  return k_; }

    // find highest falling edge:
    ulong j = k_ - 1;
    while ( 1 == (x_[j] - x_[j-1]) )  { --j; }

    ulong ret = j;
    --x_[j];  // move down edge element

    // ... and move rest of block to high end:
    while ( ++j < k_ )  x_[j] = n_ - k_ + j;

    return  ret;
}
```

The listing in figure 6.1-A was created with the program [FXT: comb/combination-lex-demo.cc]. The routine generates the combinations $\binom{32}{20}$ at a rate of about 95 million per second. The combinations $\binom{32}{12}$ are generated at a rate of 160 million per second.

6.1.2 Co-lexicographic order

The combinations of three elements out of six in *co-lexicographic* (or *colex*) order are shown in figure 6.1-A (right). Algorithms to compute the successor and predecessor are implemented in [FXT: `class combination_colex` in comb/combination-colex.h]:

```
class combination_colex
{
public:
    ulong *x_;    // combination: k elements 0<=x[j]<k in increasing order
    ulong n_, k_; // Combination (n choose k)

    combination_colex(ulong n, ulong k)
    {
```

```
        n_ = n;  k_ = k;
        x_ = new ulong[k_+1];
        x_[k_] = n_ + 2;  // sentinel
        first();
    }
  [--snip--]
    ulong next()
    // Return greatest position that changed, return k with last combination
    {
        if ( x_[0] == n_ - k_ )  // current combination is the last
        { first();  return k_; }
        ulong j = 0;
        // until lowest rising edge:  attach block at low end
        while ( 1 == (x_[j+1] - x_[j]) )  { x_[j] = j;  ++j; }  // can touch sentinel
        ++x_[j];  // move edge element up
        return  j;
    }
    ulong prev()
    // Return greatest position that changed, return k with last combination
    {
        if ( x_[k_-1] == k_-1 )  // current combination is the first
        { last();  return k_; }
        // find lowest falling edge:
        ulong j = 0;
        while ( j == x_[j] )  ++j;  // can touch sentinel
        --x_[j];  // move edge element down
        ulong ret = j;
        // attach rest of low block:
        while ( 0!=j-- )  x_[j] = x_[j+1] - 1;
        return  ret;
    }
  [--snip--]
```

The listing in figure 6.1-A was created with the program [FXT: comb/combination-colex-demo.cc]. The combinations are generated $\binom{32}{20}$ at a rate of about 140 million objects per second, the combinations $\binom{32}{12}$ are generated at a rate of 190 million objects per second.

As a toy application of the combinations in co-lexicographic order we compute the products of k of the n smallest primes. We maintain an array of k products shown at the right of figure 6.1-B. When the return value of the method `next()` is j then $j+1$ elements have to be updated from right to left [FXT: comb/kproducts-colex-demo.cc]:

```
combination_colex C(n, k);
const ulong *c = C.data();  // combinations as sets
ulong *tf = new ulong[n];  // table of Factors (primes)
// fill in small primes:
for (ulong j=0,f=2; j<n; ++j)  { tf[j] = f;  f=next_small_prime(f+1); }
ulong *tp = new ulong[k+1];  // table of Products
tp[k] = 1; // one appended (sentinel)
ulong j = k-1;
do
{
    // update products from right:
    ulong x = tp[j+1];
    { ulong i = j;
    do
    {
        ulong f = tf[ c[i] ];
        x *= f;
        tp[i] = x;
    }
    while ( i-- );
    } // here: final product is x == tp[0]
    // visit the product x here
    j = C.next();
```

```
              combination  j     delta-set      products
        1:    { 0, 1, 2 }  2      111....    [   30   15    5    1 ]
        2:    { 0, 1, 3 }  2      11.1...    [   42   21    7    1 ]
        3:    { 0, 2, 3 }  1      1.11...    [   70   35    7    1 ]
        4:    { 1, 2, 3 }  0      .111...    [  105   35    7    1 ]
        5:    { 0, 1, 4 }  2      11..1..    [   66   33   11    1 ]
        6:    { 0, 2, 4 }  1      1.1.1..    [  110   55   11    1 ]
        7:    { 1, 2, 4 }  0      .11.1..    [  165   55   11    1 ]
        8:    { 0, 3, 4 }  1      1..11..    [  154   77   11    1 ]
        9:    { 1, 3, 4 }  0      .1.11..    [  231   77   11    1 ]
       10:    { 2, 3, 4 }  0      ..111..    [  385   77   11    1 ]
       11:    { 0, 1, 5 }  2      11...1.    [   78   39   13    1 ]
       12:    { 0, 2, 5 }  1      1.1..1.    [  130   65   13    1 ]
       13:    { 1, 2, 5 }  0      .11..1.    [  195   65   13    1 ]
       14:    { 0, 3, 5 }  1      1..1.1.    [  182   91   13    1 ]
       15:    { 1, 3, 5 }  0      .1.1.1.    [  273   91   13    1 ]
       16:    { 2, 3, 5 }  0      ..11.1.    [  455   91   13    1 ]
       17:    { 0, 4, 5 }  1      1...11.    [  286  143   13    1 ]
       18:    { 1, 4, 5 }  0      .1..11.    [  429  143   13    1 ]
       19:    { 2, 4, 5 }  0      ..1.11.    [  715  143   13    1 ]
       20:    { 3, 4, 5 }  0      ...111.    [ 1001  143   13    1 ]
       21:    { 0, 1, 6 }  2      11....1    [  102   51   17    1 ]
       22:    { 0, 2, 6 }  1      1.1...1    [  170   85   17    1 ]
       23:    { 1, 2, 6 }  0      .11...1    [  255   85   17    1 ]
       24:    { 0, 3, 6 }  1      1..1..1    [  238  119   17    1 ]
       25:    { 1, 3, 6 }  0      .1.1..1    [  357  119   17    1 ]
       26:    { 2, 3, 6 }  0      ..11..1    [  595  119   17    1 ]
       27:    { 0, 4, 6 }  1      1...1.1    [  374  187   17    1 ]
       28:    { 1, 4, 6 }  0      .1..1.1    [  561  187   17    1 ]
       29:    { 2, 4, 6 }  0      ..1.1.1    [  935  187   17    1 ]
       30:    { 3, 4, 6 }  0      ...11.1    [ 1309  187   17    1 ]
       31:    { 0, 5, 6 }  1      1....11    [  442  221   17    1 ]
       32:    { 1, 5, 6 }  0      .1...11    [  663  221   17    1 ]
       33:    { 2, 5, 6 }  0      ..1..11    [ 1105  221   17    1 ]
       34:    { 3, 5, 6 }  0      ...1.11    [ 1547  221   17    1 ]
       35:    { 4, 5, 6 }  0      ....111    [ 2431  221   17    1 ]
```

Figure 6.1-B: All products of $k = 3$ of the $n = 7$ smallest primes $(2, 3, 5, \ldots, 17)$. The products are the leftmost elements of the array on the right hand side.

```
}
while ( j < k );
```

The leftmost element of this array is the desired product. A sentinel element one at the end of the array is used to keep the code simple. With lexicographic order the update would go from left to right.

6.2 Order by prefix shifts (cool-lex)

```
1:  1....      1:  11...      1:  111..      1:  1111.
2:  .1...      2:  .11..      2:  .111.      2:  .1111
3:  ..1..      3:  1.1..      3:  1.11.      3:  1.111
4:  ...1.      4:  .1.1.      4:  11.1.      4:  11.11
5:  ....1      5:  ..11.      5:  .11.1      5:  111.1
               6:  1..1.      6:  1.1.1
               7:  .1..1      7:  .1.11
               8:  ..1.1      8:  ..111
               9:  ...11      9:  1..11
              10:  1...1     10:  11..1
```

Figure 6.2-A: Combinations $\binom{5}{k}$, for $k = 1, 2, 3, 4$ in an ordering generated by prefix shifts.

An algorithm for generating combinations by prefix shifts which is given in [203]. The ordering is called *cool-lex* in the paper. Figure 6.2-A shows some orders for $\binom{5}{k}$, figure 6.2-B shows the combinations

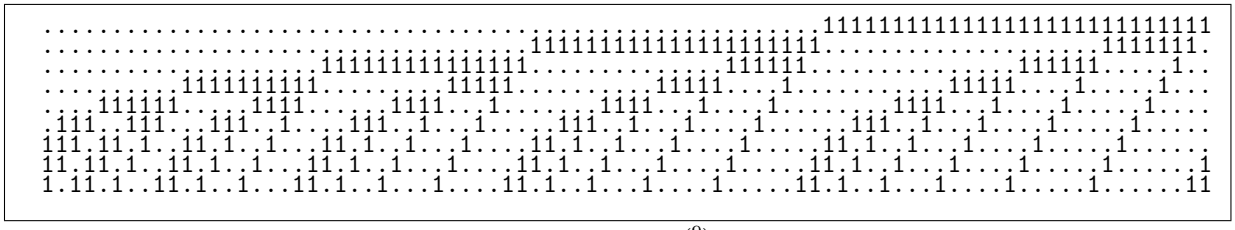


Figure 6.2-B: Combinations $\binom{9}{3}$ via prefix shifts.

$\binom{9}{3}$. The listings were created with the program [FXT: comb/combination-pref-demo.cc] which uses the implementation in [FXT: `class combination_pref` in comb/combination-pref.h]:

```
class combination_pref
{
public:
    ulong *b_;  // data as delta set
    ulong s_, t_, n_;  // combination (n choose k) where n=s+t, k=t.
private:
    ulong x, y;  // aux
public:
    combination_pref(ulong n, ulong k)
    // Must have: n>=2, k>=1  (i.e. s!=0 and t!=0)
    {
        s_ = n - k;
        t_ = k;
        n_ = s_ + t_;
        b_ = new ulong[n_];
        first();
    }
  [--snip--]

    void first()
    {
        for (ulong j=0; j<n_; ++j)  b_[j] = 0;
        for (ulong j=0; j<t_; ++j)  b_[j] = 1;
        x = 0;  y = 0;
    }

    bool next()
    {
        if ( x==0 )  { x=1;  b_[t_]=1;  b_[0]=0;  return true; }
        else
        {
            if ( x>=n_-1 )  return false;
            else
            {
                b_[x] = 0; ++x;  b_[y] = 1; ++y; // X(s,t)
                if ( b_[x]==0 )
                {
                    b_[x] = 1;  b_[0] = 0;       // Y(s,t)
                    if ( y>1 )  x = 1;           // Z(s,t)
                    y = 0;
                }
                return true;
            }
        }
    }
  [--snip--]
```

The combinations are generated $\binom{32}{20}$ at a rate of about 95 million objects per second, the combinations $\binom{32}{12}$ are generated at a rate of 85 million objects per second.

6.3 Minimal-change order

The combinations of three elements out of six in a *minimal-change* order (a *Gray code*) are shown in figure 6.3-A (left). With each transition exactly one element changes its position. We use a recursion for

Gray code			complemented Gray code		
1:	{ 0, 1, 2 }	111...	1:	{ 3, 4, 5 }	...111
2:	{ 0, 2, 3 }	1.11..	2:	{ 1, 4, 5 }	.1..11
3:	{ 1, 2, 3 }	.111..	3:	{ 0, 4, 5 }	1...11
4:	{ 0, 1, 3 }	11.1..	4:	{ 2, 4, 5 }	..1.11
5:	{ 0, 3, 4 }	1..11.	5:	{ 1, 2, 5 }	.11..1
6:	{ 1, 3, 4 }	.1.11.	6:	{ 0, 2, 5 }	1.1..1
7:	{ 2, 3, 4 }	..111.	7:	{ 0, 1, 5 }	11...1
8:	{ 0, 2, 4 }	1.1.1.	8:	{ 1, 3, 5 }	.1.1.1
9:	{ 1, 2, 4 }	.11.1.	9:	{ 0, 3, 5 }	1..1.1
10:	{ 0, 1, 4 }	11..1.	10:	{ 2, 3, 5 }	..11.1
11:	{ 0, 4, 5 }	1...11	11:	{ 1, 2, 3 }	.111..
12:	{ 1, 4, 5 }	.1..11	12:	{ 0, 2, 3 }	1.11..
13:	{ 2, 4, 5 }	..1.11	13:	{ 0, 1, 3 }	11.1..
14:	{ 3, 4, 5 }	...111	14:	{ 0, 1, 2 }	111...
15:	{ 0, 3, 5 }	1..1.1	15:	{ 1, 2, 4 }	.11.1.
16:	{ 1, 3, 5 }	.1.1.1	16:	{ 0, 2, 4 }	1.1.1.
17:	{ 2, 3, 5 }	..11.1	17:	{ 0, 1, 4 }	11..1.
18:	{ 0, 2, 5 }	1.1..1	18:	{ 1, 3, 4 }	.1.11.
19:	{ 1, 2, 5 }	.11..1	19:	{ 0, 3, 4 }	1..11.
20:	{ 0, 1, 5 }	11...1	20:	{ 2, 3, 4 }	..111.

Figure 6.3-A: Combinations $\binom{6}{3}$ in Gray order (left), and complemented Gray order (right).

the list $C(n,k)$ of combinations $\binom{n}{k}$ (symbols as in relation 12.0-1 on page 281):

$$C(n,k) \;=\; \begin{matrix}[C(n-1,\,k) \qquad\qquad\;\;] \\ [(n)\,.\,C^{\mathbf{R}}(n-1,\,k-1)]\end{matrix} \;=\; \begin{matrix}[0\,.\,C(n-1,\,k) \qquad\;\;] \\ [1\,.\,C^{\mathbf{R}}(n-1,\,k-1)]\end{matrix} \tag{6.3-1}$$

The first equality is for the set representation, the second for the delta-set representation. An implementation is given in [FXT: comb/combination-gray-rec-demo.cc]:

```
ulong *x;   // elements in combination at x[1] ... x[k]

void comb_gray(ulong n, ulong k, bool z)
{
    if ( k==n )
    {
        for (ulong j=1; j<=k; ++j)  x[j] = j;
        visit();
        return;
    }

    if ( z )  // forward:
    {
        comb_gray(n-1, k, z);
        if ( k>0 )  { x[k] = n;  comb_gray(n-1, k-1, !z); }
    }
    else     // backward:
    {
        if ( k>0 )  { x[k] = n;  comb_gray(n-1, k-1, !z); }
        comb_gray(n-1, k, z);
    }
}
```

A recursion for the complemented order is

$$C(n,k) \;=\; \begin{matrix}[(n)\,.\,C(n-1,\,k-1)] \\ [C^{\mathbf{R}}(n-1,\,k) \qquad\;\;]\end{matrix} \;=\; \begin{matrix}[1\,.\,C(n-1,\,k-1)] \\ [0\,.\,C^{\mathbf{R}}(n-1,\,k) \;\;]\end{matrix} \tag{6.3-2}$$

```
void comb_gray_compl(ulong n, ulong k, bool z)
{
  [--snip--]
    if ( z )  // forward:
    {
        if ( k>0 )  { x[k] = n;  comb_gray_compl(n-1, k-1, z); }
        comb_gray_compl(n-1, k, !z);
    }
    else     // backward:
```

```
    {
        comb_gray_compl(n-1, k, !z);
        if ( k>0 )  { x[k] = n;  comb_gray_compl(n-1, k-1, z); }
    }
}
```

A very efficient (*revolving door*) algorithm that generates the sets for the Gray code is given in [191]. An implementation following [157] is [FXT: `class combination_revdoor` in comb/combination-revdoor.h]. Usage of the class is shown in [FXT: comb/combination-revdoor-demo.cc]. The routine generates the combinations $\binom{32}{20}$ at a rate of about 115 M/s, the combinations $\binom{32}{12}$ are generated at a rate of 181 M/s. An implementation geared for good performance with small values of k is given in [161], a C++ adaptation is [FXT: comb/combination-lam-demo.cc]. The combinations $\binom{32}{12}$ a generated at a rate of 190 M/s, and 250 M/s for the combinations $\binom{64}{7}$. The routine is limited to values $k \geq 2$.

6.4 The Eades-McKay strong minimal-change order

```
          Eades-McKay                      complemented Eades-McKay

 1:  { 4, 5, 6 }  ....111          1:  { 4, 5, 6 }  ....111
 2:  { 3, 5, 6 }  ...1.11          2:  { 3, 5, 6 }  ...1.11
 3:  { 2, 5, 6 }  ..1..11          3:  { 2, 5, 6 }  ..1..11
 4:  { 1, 5, 6 }  .1...11          4:  { 1, 5, 6 }  .1...11
 5:  { 0, 5, 6 }  1....11          5:  { 0, 5, 6 }  1....11
 6:  { 0, 1, 6 }  11....1          6:  { 0, 4, 6 }  1...1.1
 7:  { 0, 2, 6 }  1.1...1          7:  { 1, 4, 6 }  .1..1.1
 8:  { 1, 2, 6 }  .11...1          8:  { 2, 4, 6 }  ..1.1.1
 9:  { 1, 3, 6 }  .1.1..1          9:  { 3, 4, 6 }  ...11.1
10:  { 0, 3, 6 }  1..1..1         10:  { 2, 3, 6 }  ..11..1
11:  { 2, 3, 6 }  ..11..1         11:  { 1, 3, 6 }  .1.1..1
12:  { 2, 4, 6 }  ..1.1.1         12:  { 0, 3, 6 }  1..1..1
13:  { 1, 4, 6 }  .1..1.1         13:  { 0, 2, 6 }  1.1...1
14:  { 0, 4, 6 }  1...1.1         14:  { 1, 2, 6 }  .11...1
15:  { 3, 4, 6 }  ...11.1         15:  { 0, 1, 6 }  11....1
16:  { 3, 4, 5 }  ...111.         16:  { 0, 1, 5 }  11...1.
17:  { 2, 4, 5 }  ..1.11.         17:  { 0, 2, 5 }  1.1..1.
18:  { 1, 4, 5 }  .1..11.         18:  { 1, 2, 5 }  .11..1.
19:  { 0, 4, 5 }  1...11.         19:  { 2, 3, 5 }  ..11.1.
20:  { 0, 1, 5 }  11...1.         20:  { 1, 3, 5 }  .1.1.1.
21:  { 0, 2, 5 }  1.1..1.         21:  { 0, 3, 5 }  1..1.1.
22:  { 1, 2, 5 }  .11..1.         22:  { 0, 4, 5 }  1...11.
23:  { 1, 3, 5 }  .1.1.1.         23:  { 1, 4, 5 }  .1..11.
24:  { 0, 3, 5 }  1..1.1.         24:  { 2, 4, 5 }  ..1.11.
25:  { 2, 3, 5 }  ..11.1.         25:  { 3, 4, 5 }  ...111.
26:  { 2, 3, 4 }  ..111..         26:  { 2, 3, 4 }  ..111..
27:  { 1, 3, 4 }  .1.11..         27:  { 1, 3, 4 }  .1.11..
28:  { 0, 3, 4 }  1..11..         28:  { 0, 3, 4 }  1..11..
29:  { 0, 1, 4 }  11..1..         29:  { 0, 2, 4 }  1.1.1..
30:  { 0, 2, 4 }  1.1.1..         30:  { 1, 2, 4 }  .11.1..
31:  { 1, 2, 4 }  .11.1..         31:  { 0, 1, 4 }  11..1..
32:  { 1, 2, 3 }  .111...         32:  { 0, 1, 3 }  11.1...
33:  { 0, 2, 3 }  1.11...         33:  { 0, 2, 3 }  1.11...
34:  { 0, 1, 3 }  11.1...         34:  { 1, 2, 3 }  .111...
35:  { 0, 1, 2 }  111....         35:  { 0, 1, 2 }  111....
```

Figure 6.4-A: Combinations $\binom{6}{3}$ in the Eades-McKay order (left), and in the complemented Eades-Mckay order (right).

In any Gray code order for combinations just one element is moved between to successive combinations. When an element is moved across any other then there is more than one change on the set representation. If i elements are crossed then $i + 1$ entries in the set change:

```
      set          delta set
{ 0, 1, 2, 3 }    1111..
{ 1, 2, 3, 4 }    .1111.
```

A *strong minimal-change order* is a Gray code where only one entry in the set representation is changed per step. That is, only zeros in the delta set representation are crossed, and the moves are called *homogeneous*. One such order is the *Eades-McKay* sequence described in [104]. The Eades-McKay sequence for the combinations $\binom{6}{3}$ is shown in figure 6.4-A (left). It can be generated with the program [FXT: comb/combination-emk-rec-demo.cc]:

```
ulong *rv;  // elements in combination at rv[1] ... rv[k]

void
comb_emk(ulong n, ulong k, bool z)
{
    if ( k==n )
    {
        for (ulong j=1; j<=k; ++j)  rv[j] = j;
        visit();
        return;
    }

    if ( z )  // forward:
    {
        if ( (n>=2) && (k>=2) )  { rv[k] = n;  rv[k-1] = n-1;  comb_emk(n-2, k-2, z); }
        if ( (n>=2) && (k>=1) )  { rv[k] = n;  comb_emk(n-2, k-1, !z); }
        if ( (n>=1) )            { comb_emk(n-1, k, z); }
    }
    else     // backward:
    {
        if ( (n>=1) )            { comb_emk(n-1, k, z); }
        if ( (n>=2) && (k>=1) )  { rv[k] = n;  comb_emk(n-2, k-1, !z); }
        if ( (n>=2) && (k>=2) )  { rv[k] = n;  rv[k-1] = n-1;  comb_emk(n-2, k-2, z); }
    }
}
```

The combinations $\binom{32}{20}$ are generated at a rate of about 44 million per second, the combinations $\binom{32}{12}$ at a rate of 34 million per second.

The underlying recursion for the list $E(n,k)$ of combinations $\binom{n}{k}$ is (notations as in relation 12.0-1 on page 281)

$$E(n,k) \;=\; \begin{matrix}[(n)\,.\,(n-1)\,.\,E(n-2,\,k-2)] \\ [(n)\,.\,E^{\mathbf{R}}(n-2,\,k-1) \quad\;\;] \\ [E(n-1,\,k) \quad\quad\quad\quad\;\;]\end{matrix} \;=\; \begin{matrix}[1\,1\,.\,E(n-2,\,k-2)\;\;] \\ [1\,0\,.\,E^{\mathbf{R}}(n-2,\,k-1)] \\ [0\,.\,E(n-1,\,k) \quad\;\;]\end{matrix} \tag{6.4-1}$$

Again, the first equality is for the set representation, the second for the delta-set representation. Counting the elements on both sides gives the relation

$$\binom{n}{k} \;=\; \binom{n-2}{k-2} + \binom{n-2}{k-1} + \binom{n-1}{k} \tag{6.4-2}$$

which is an easy consequence of relation 6.0-3 on page 166. A recursion for the complemented sequence (with respect to the delta sets) is

$$E'(n,k) \;=\; \begin{matrix}[E'(n-1,\,k-1) \quad\quad\quad\;\;] \\ [(n-1)\,.\,E'^{\mathbf{R}}(n-2,\,k-1)] \\ [(n)\,.\;.\,E'(n-2,\,k) \quad\quad\;\;]\end{matrix} \;=\; \begin{matrix}[1\,.\,E'(n-1,\,k-1)\;\;] \\ [0\,1\,.\,E'^{\mathbf{R}}(n-2,\,k-1)] \\ [0\,0\,.\,E'(n-2,\,k) \quad\;\;]\end{matrix} \tag{6.4-3}$$

Counting on both sides gives

$$\binom{n}{k} \;=\; \binom{n-2}{k} + \binom{n-2}{k-1} + \binom{n-1}{k-1} \tag{6.4-4}$$

The condition for the recursion end has to be modified:

```
void
comb_emk_compl(ulong n, ulong k, bool z)
{
    if ( (k==0) || (k==n) )
    {
        for (ulong j=1; j<=k; ++j)  rv[j] = j;
        ++ct;
```

```
        visit();
        return;
    }

    if ( z )  // forward:
    {
        if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_emk_compl(n-1, k-1, z); }     // 1
        if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_emk_compl(n-2, k-1, !z); }  // 01
        if ( (n>=2)  )           { comb_emk_compl(n-2, k-0, z); }                 // 00
    }
    else     // backward:
    {
        if ( (n>=2) )            { comb_emk_compl(n-2, k-0, z); }                 // 00
        if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_emk_compl(n-2, k-1, !z); }  // 01
        if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_emk_compl(n-1, k-1, z); }     // 1
    }
}
```

The complemented sequence is not a strong Gray code.

Iterative generation via modulo moves

An iterative algorithm for the Eades-McKay sequence is given in [FXT: `class combination_emk` in comb/combination-emk.h]:

```
class combination_emk
{
public:
    ulong *x_;  // combination: k elements 0<=x[j]<k in increasing order
    ulong *s_;  // aux: start of range for moves
    ulong *a_;  // aux: actual start position of moves
    ulong n_, k_; // Combination (n choose k)
public:
    combination_emk(ulong n, ulong k)
    {
        n_ = n;
        k_ = k;
        x_ = new ulong[k_+1];  // incl. high sentinel
        s_ = new ulong[k_+1];  // incl. high sentinel
        a_ = new ulong[k_];
        x_[k_] = n_;
        first();
    }
  [--snip--]
    void first()
    {
        for (ulong j=0; j<k_; ++j)  x_[j] = j;
        for (ulong j=0; j<k_; ++j)  s_[j] = j;
        for (ulong j=0; j<k_; ++j)  a_[j] = x_[j];
    }
```

The computation of the successor uses modulo steps:

```
    ulong next()
    // Return position where track changed, return k with last combination
    {
        ulong j = k_;
        while ( j-- )  // loop over tracks
        {
            const ulong sj = s_[j];
            const ulong m = x_[j+1] - sj - 1;

            if ( 0!=m )  // unless range empty
            {
                ulong u = x_[j] - sj;

                // modulo moves:
                if ( 0==(j&1) )
                {
                    ++u;
                    if ( u>m )  u = 0;
                }
                else
                {
```

```
                    --u;
                    if ( u>m )  u = m;
                }
                u += sj;
                if ( u != a_[j] )  // next pos != start position
                {
                    x_[j] = u;
                    s_[j+1] = u+1;
                    return j;
                }
            }
            a_[j] = x_[j];
        }
        return  k_;  // current combination is last
    }
};
```

The combinations $\binom{32}{20}$ are generated at a rate of about 60 million per second, the combinations $\binom{32}{12}$ at a rate of 85 million per second [FXT: comb/combination-emk-demo.cc].

6.5 Two-close orderings via endo/enup moves

6.5.1 The endo and enup orderings for numbers

m	endo sequence	m	enup sequence
1:	1 0	1:	0 1
2:	1 2 0	2:	0 2 1
3:	1 3 2 0	3:	0 2 3 1
4:	1 3 4 2 0	4:	0 2 4 3 1
5:	1 3 5 4 2 0	5:	0 2 4 5 3 1
6:	1 3 5 6 4 2 0	6:	0 2 4 6 5 3 1
7:	1 3 5 7 6 4 2 0	7:	0 2 4 6 7 5 3 1
8:	1 3 5 7 8 6 4 2 0	8:	0 2 4 6 8 7 5 3 1
9:	1 3 5 7 9 8 6 4 2 0	9:	0 2 4 6 8 9 7 5 3 1

Figure 6.5-A: The endo (left), and enup (right) orderings with maximal value m.

The *endo* order of the set $\{0, 1, 2, \ldots, m\}$ is obtained by writing all odd numbers of the set in increasing order followed by all even numbers in decreasing order: $\{1, 3, 5, \ldots, 6, 4, 2, 0\}$. The term endo stands for 'Even Numbers DOwn, odd numbers up'. A routine for generating the successor in endo order with maximal value m is [FXT: comb/endo-enup.h]:

```
inline ulong next_endo(ulong x, ulong m)
// Return next number in endo order
{
    if ( x & 1 )  // x odd
    {
        x += 2;
        if ( x>m )  x = m - (m&1);  // == max even <= m
    }
    else  // x even
    {
        x = ( x==0 ? 1 : x-2 );
    }
    return x;
}
```

The sequences for the first few m are shown in figure 6.5-A. The routine computes one for the input zero.

An ordering starting with the even numbers in increasing order will be called *enup* (for 'Even Numbers UP, odd numbers down'). The computation of the successor can be implemented as

```
static inline ulong next_enup(ulong x, ulong m)
{
    if ( x & 1 )  // x odd
    {
        x = ( x==1 ? 0 : x-2 );
    }
```

```
    else  // x even
    {
        x += 2;
        if ( x>m )  x = m - !(m&1);  // max odd <=m
    }
    return x;
}
```

As the orderings are reversals of each other, we define:

```
static inline ulong prev_endo(ulong x, ulong m)  { return next_enup(x, m); }
static inline ulong prev_enup(ulong x, ulong m)  { return next_endo(x, m); }
```

A function that returns the x-th number in enup order with maximal digit m is

```
static inline ulong enup_num(ulong x, ulong m)
{
    ulong r = 2*x;
    if  ( r>m )  r = 2*m+1 - r;
    return r;
}
```

The function will only work if $x \leq m$. For example, with $m = 5$:

```
    x:   0 1 2 3 4 5
    r:   0 2 4 5 3 1
```

The inverse function is

```
static inline ulong enup_idx(ulong x, ulong m)
{
    const ulong b = x & 1;
    x >>= 1;
    return  ( b ? m-x : x );
}
```

The equivalent function to map into endo order is

```
static inline ulong endo_num(ulong x, ulong m)
{
    // return enup_num(m-x, m);
    x = m - x;
    ulong r = 2*x;
    if  ( r>m )  r = 2*m+1 - r;
    return  r;
}
```

For example,

```
   x:   0 1 2 3 4 5
   r:   1 3 5 4 2 0
```

The inverse is

```
static inline ulong endo_idx(ulong x, ulong m)
{
    const ulong b = x & 1;
    x >>= 1;
    return  ( b ? x : m-x );
}
```

6.5.2 The endo and enup orderings

Two strong minimal-change orderings for combinations can be obtained via moves in enup and endo order. Figure 6.5-B shows an ordering where the moves to the right are on even positions (enup order, left). If the moves to the right are on odd positions (endo order) then Chase's sequence is obtained (right). Both have the property of being *two-close*: an element in the delta set moves by at most two positions (and the move is homogeneous, no other element is crossed). This property is a direct consequence of the fact that all moves with enup and endo order are by at most two positions. An implementation of an iterative algorithm for the computation of the combinations in enup order is [FXT: `class combination_enup` in comb/combination-enup.h].

```
class combination_enup
{
public:
    ulong *x_;  // combination: k elements 0<=x[j]<k in increasing order
```

enup moves			endo moves		
1:	{ 0, 1, 2 }	111.....	1:	{ 0, 1, 2 }	111.....
2:	{ 0, 1, 4 }	11..1...	2:	{ 0, 1, 3 }	11.1....
3:	{ 0, 1, 6 }	11....1.	3:	{ 0, 1, 5 }	11...1..
4:	{ 0, 1, 7 }	11.....1	4:	{ 0, 1, 7 }	11.....1
5:	{ 0, 1, 5 }	11...1..	5:	{ 0, 1, 6 }	11....1.
6:	{ 0, 1, 3 }	11.1....	6:	{ 0, 1, 4 }	11..1...
7:	{ 0, 2, 3 }	1.11....	7:	{ 0, 3, 4 }	1..11...
8:	{ 0, 2, 4 }	1.1.1...	8:	{ 0, 3, 5 }	1..1.1..
9:	{ 0, 2, 6 }	1.1...1.	9:	{ 0, 3, 7 }	1..1...1
10:	{ 0, 2, 7 }	1.1....1	10:	{ 0, 3, 6 }	1..1..1.
11:	{ 0, 2, 5 }	1.1..1..	11:	{ 0, 5, 6 }	1....11.
12:	{ 0, 4, 5 }	1...11..	12:	{ 0, 5, 7 }	1....1.1
13:	{ 0, 4, 6 }	1...1.1.	13:	{ 0, 6, 7 }	1.....11
14:	{ 0, 4, 7 }	1...1..1	14:	{ 0, 4, 7 }	1...1..1
15:	{ 0, 6, 7 }	1.....11	15:	{ 0, 4, 6 }	1...1.1.
16:	{ 0, 5, 7 }	1....1.1	16:	{ 0, 4, 5 }	1...11..
17:	{ 0, 5, 6 }	1....11.	17:	{ 0, 2, 5 }	1.1..1..
18:	{ 0, 3, 6 }	1..1..1.	18:	{ 0, 2, 7 }	1.1....1
19:	{ 0, 3, 7 }	1..1...1	19:	{ 0, 2, 6 }	1.1...1.
20:	{ 0, 3, 5 }	1..1.1..	20:	{ 0, 2, 4 }	1.1.1...
21:	{ 0, 3, 4 }	1..11...	21:	{ 0, 2, 3 }	1.11....
22:	{ 2, 3, 4 }	..111...	22:	{ 1, 2, 3 }	.111....
23:	{ 2, 3, 6 }	..11..1.	23:	{ 1, 2, 5 }	.11..1..
24:	{ 2, 3, 7 }	..11...1	24:	{ 1, 2, 7 }	.11....1
25:	{ 2, 3, 5 }	..11.1..	25:	{ 1, 2, 6 }	.11...1.
26:	{ 2, 4, 5 }	..1.11..	26:	{ 1, 2, 4 }	.11.1...
27:	{ 2, 4, 6 }	..1.1.1.	27:	{ 1, 3, 4 }	.1.11...
28:	{ 2, 4, 7 }	..1.1..1	28:	{ 1, 3, 5 }	.1.1.1..
29:	{ 2, 6, 7 }	..1...11	29:	{ 1, 3, 7 }	.1.1...1
30:	{ 2, 5, 7 }	..1..1.1	30:	{ 1, 3, 6 }	.1.1..1.
31:	{ 2, 5, 6 }	..1..11.	31:	{ 1, 5, 6 }	.1...11.
32:	{ 4, 5, 6 }	....111.	32:	{ 1, 5, 7 }	.1...1.1
33:	{ 4, 5, 7 }	....11.1	33:	{ 1, 6, 7 }	.1....11
34:	{ 4, 6, 7 }	....1.11	34:	{ 1, 4, 7 }	.1..1..1
35:	{ 5, 6, 7 }	111	35:	{ 1, 4, 6 }	.1..1.1.
36:	{ 3, 6, 7 }	...1..11	36:	{ 1, 4, 5 }	.1..11..
37:	{ 3, 5, 7 }	...1.1.1	37:	{ 3, 4, 5 }	...111..
38:	{ 3, 5, 6 }	...1.11.	38:	{ 3, 4, 7 }	...11..1
39:	{ 3, 4, 6 }	...11.1.	39:	{ 3, 4, 6 }	...11.1.
40:	{ 3, 4, 7 }	...11..1	40:	{ 3, 5, 6 }	...1.11.
41:	{ 3, 4, 5 }	...111..	41:	{ 3, 5, 7 }	...1.1.1
42:	{ 1, 4, 5 }	.1..11..	42:	{ 3, 6, 7 }	...1..11
43:	{ 1, 4, 6 }	.1..1.1.	43:	{ 5, 6, 7 }	111
44:	{ 1, 4, 7 }	.1..1..1	44:	{ 4, 6, 7 }	....1.11
45:	{ 1, 6, 7 }	.1....11	45:	{ 4, 5, 7 }	....11.1
46:	{ 1, 5, 7 }	.1...1.1	46:	{ 4, 5, 6 }	....111.
47:	{ 1, 5, 6 }	.1...11.	47:	{ 2, 5, 6 }	..1..11.
48:	{ 1, 3, 6 }	.1.1..1.	48:	{ 2, 5, 7 }	..1..1.1
49:	{ 1, 3, 7 }	.1.1...1	49:	{ 2, 6, 7 }	..1...11
50:	{ 1, 3, 5 }	.1.1.1..	50:	{ 2, 4, 7 }	..1.1..1
51:	{ 1, 3, 4 }	.1.11...	51:	{ 2, 4, 6 }	..1.1.1.
52:	{ 1, 2, 4 }	.11.1...	52:	{ 2, 4, 5 }	..1.11..
53:	{ 1, 2, 6 }	.11...1.	53:	{ 2, 3, 5 }	..11.1..
54:	{ 1, 2, 7 }	.11....1	54:	{ 2, 3, 7 }	..11...1
55:	{ 1, 2, 5 }	.11..1..	55:	{ 2, 3, 6 }	..11..1.
56:	{ 1, 2, 3 }	.111....	56:	{ 2, 3, 4 }	..111...

Figure 6.5-B: Combinations $\binom{8}{3}$ via enup moves (left), and via endo moves (Chase's sequence, right).

```
    ulong *s_;  // aux: start of range for enup moves
    ulong *a_;  // aux: actual start position of enup moves
    ulong n_, k_; // Combination (n choose k)

public:
    combination_enup(ulong n, ulong k)
    {
        n_ = n;
        k_ = k;
        x_ = new ulong[k_+1];  // incl. high sentinel
        s_ = new ulong[k_+1];  // incl. high sentinel
        a_ = new ulong[k_];
        x_[k_] = n_;
        first();
    }
  [--snip--]
    void first()
    {
        for (ulong j=0; j<k_; ++j)  x_[j] = j;
        for (ulong j=0; j<k_; ++j)  s_[j] = j;
        for (ulong j=0; j<k_; ++j)  a_[j] = x_[j];
    }
```

The successor of the current combination is computed by finding the range of possible movements (variable `m`) and, unless the range is empty, move until we are back at the start position:

```
    ulong next()
    // Return position where track changed, return k with last combination
    {
        ulong j = k_;
        while ( j-- )  // loop over tracks
        {
            const ulong sj = s_[j];
            const ulong m = x_[j+1] - sj - 1;

            if ( 0!=m )  // unless range empty
            {
                ulong u = x_[j] - sj;

                // move right on even positions:
                if ( 0==(sj&1) )  u = next_enup(u, m);
                else              u = next_endo(u, m);

                u += sj;

                if ( u != a_[j] )  // next pos != start position
                {
                    x_[j] = u;
                    s_[j+1] = u+1;
                    return j;
                }
            }

            a_[j] = x_[j];
        }

        return  k_;  // current combination is last
    }
};
```

The routine is similar to the one used for the Eades-McKay sequence that is given on page 174. The combinations $\binom{32}{20}$ are generated at a rate of 45 million objects per second, the combinations $\binom{32}{12}$ at a rate of 55 million per second.

The only change in the implementation for computing the endo ordering is (at the obvious place in the code) [FXT: comb/combination-endo.h]:

```
                // move right on odd positions:
                if ( 0==(sj&1) )  u = next_endo(u, m);
                else              u = next_enup(u, m);
```

The ordering obtained by endo moves is called *Chase's sequence*. Figure 6.5-B was created with the programs [FXT: comb/combination-enup-demo.cc] and [FXT: comb/combination-endo-demo.cc].

The underlying recursion for the list $U(n,k)$ of combinations $\binom{n}{k}$ in enup order is

$$U(n,k) \quad = \quad \begin{array}{l} [(n)\,.\,(n-1)\,.\,U(n-2,\,k-2)] \\ [(n)\,.\,U(n-2,\,k-1) \qquad\qquad\;\;] \\ [U^{\mathbf{R}}(n-1,\,k) \qquad\qquad\qquad\;] \end{array} \;=\; \begin{array}{l} [1\,1\,.\,U(n-2,\,k-2)] \\ [1\,0\,.\,U(n-2,\,k-1)] \\ [0\,.\,U^{\mathbf{R}}(n-1,\,k) \quad\;\;] \end{array} \tag{6.5-1}$$

The recursion is very similar to relation 6.4-1 on page 173, therefore a recursive routine is easy to obtain. The crucial part of the routine is [FXT: comb/combination-enup-rec-demo.cc]:

```
void
comb_enup(ulong n, ulong k, bool z)
{
    if ( k==n )  { visit();  return; }

    if ( z )  // forward:
    {
        if ( (n>=2) && (k>=2) )  { rv[k] = n;  rv[k-1] = n-1;  comb_enup(n-2, k-2, z); }
        if ( (n>=2) && (k>=1) )  { rv[k] = n;  comb_enup(n-2, k-1, z); }
        if ( (n>=1) )            { comb_enup(n-1, k, !z); }
    }
    else     // backward:
    {
        if ( (n>=1) )            { comb_enup(n-1, k, !z); }
        if ( (n>=2) && (k>=1) )  { rv[k] = n;  comb_enup(n-2, k-1, z); }
        if ( (n>=2) && (k>=2) )  { rv[k] = n;  rv[k-1] = n-1;  comb_enup(n-2, k-2, z); }
    }
}
```

A recursion for the complement sequence (with respect to the delta sets) is

$$U'(n,k) \quad = \quad \begin{array}{l} [(n)\,.\,{U'}^{\mathbf{R}}(n-1,\,k-1) \quad] \\ [(n-1)\,.\,U'(n-2,\,k-1)] \\ [U'(n-2,\,k) \qquad\qquad\qquad] \end{array} \;=\; \begin{array}{l} [1\,.\,{U'}^{\mathbf{R}}(n-1,\,k-1)] \\ [0\,1\,.\,U'(n-2,\,k-1)] \\ [0\,0\,.\,U'(n-2,\,k) \qquad] \end{array} \tag{6.5-2}$$

The condition for the recursion end has to be modified:

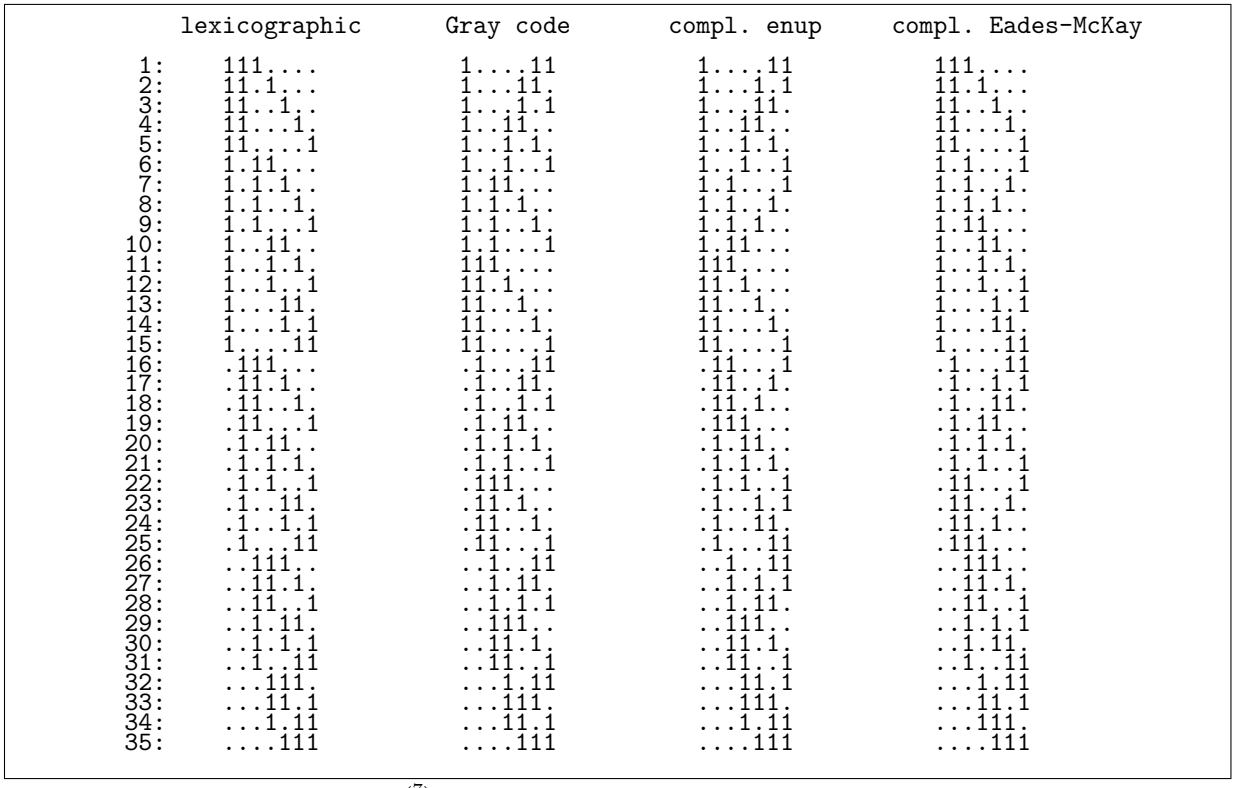
```
void
comb_enup_compl(ulong n, ulong k, bool z)
{
    if ( (k==0) || (k==n) )  { visit();  return; }

    if ( z )  // forward:
    {
        if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_enup_compl(n-1, k-1, !z); }   // 1
        if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_enup_compl(n-2, k-1, z); }  // 01
        if ( (n>=2) )            { comb_enup_compl(n-2, k-0, z); }                // 00
    }
    else     // backward:
    {
        if ( (n>=2) )            { comb_enup_compl(n-2, k-0, z); }                // 00
        if ( (n>=2) && (k>=1) )  { rv[k] = n-1;  comb_enup_compl(n-2, k-1, z); }  // 01
        if ( (n>=1) && (k>=1) )  { rv[k] = n;  comb_enup_compl(n-1, k-1, !z); }   // 1
    }
}
```

The algorithm for Chase's sequence that generates delta sets is described in [157]. An implementation is given in [FXT: `class combination_chase` in comb/combination-chase.h]. The routine generates about 64 million combinations per second both for $\binom{32}{20}$ and $\binom{32}{12}$. See [FXT: comb/combination-chase-demo.cc] for the usage of the class.

6.6 Recursive generation of certain orderings

We give a simple recursive routine to obtain the orders shown in figure 6.6-A. The combinations are generated as sets [FXT: `class comb_rec` in comb/combination-rec.h]:

```
class comb_rec
{
public:
```



	lexicographic	Gray code	compl. enup	compl. Eades-McKay
1:	111....	1....11	1....11	111....
2:	11.1...	1...11.	1...1.1	11.1...
3:	11..1..	1...1.1	1...11.	11..1..
4:	11...1.	1..11..	1..11..	11...1.
5:	11....1	1..1.1.	1..1.1.	11....1
6:	1.11...	1..1..1	1..1..1	1.1...1
7:	1.1.1..	1.11...	1.1...1	1.1..1.
8:	1.1..1.	1.1.1..	1.1..1.	1.1.1..
9:	1.1...1	1.1..1.	1.1.1..	1.11...
10:	1..11..	1.1...1	1.11...	1..11..
11:	1..1.1.	111....	111....	1..1.1.
12:	1..1..1	11.1...	11.1...	1..1..1
13:	1...11.	11..1..	11..1..	1...1.1
14:	1...1.1	11...1.	11...1.	1...11.
15:	1....11	11....1	11....1	1....11
16:	.111...	.1...11	.11...1	.1...11
17:	.11.1..	.1..11.	.11..1.	.1..1.1
18:	.11..1.	.1..1.1	.11.1..	.1..11.
19:	.11...1	.1.11..	.111...	.1.11..
20:	.1.11..	.1.1.1.	.1.11..	.1.1.1.
21:	.1.1.1.	.1.1..1	.1.1.1.	.1.1..1
22:	.1.1..1	.111...	.1.1..1	.11...1
23:	.1..11.	.11.1..	.1..1.1	.11..1.
24:	.1..1.1	.11..1.	.1..11.	.11.1..
25:	.1...11	.11...1	.1...11	.111...
26:	..111..	..1..11	..1..11	..111..
27:	..11.1.	..1.11.	..1.1.1	..11.1.
28:	..11..1	..1.1.1	..1.11.	..11..1
29:	..1.11.	..111..	..111..	..1.1.1
30:	..1.1.1	..11.1.	..11.1.	..1.11.
31:	..1..11	..11..1	..11..1	..1..11
32:	...111.	...1.11	...11.1	...1.11
33:	...11.1	...111.	...111.	...11.1
34:	...1.11	...11.1	...1.11	...111.
35:	111	111	111	111

Figure 6.6-A: All combinations $\binom{7}{3}$ in lexicographic, minimal-change, complemented enup, and complemented Eades-McKay order (from left to right).

```
    ulong n_, k_; // (n choose k)
    ulong *rv_;   //  combination: k elements 0<=x[j]<k in increasing order
    // == Record of Visits in graph
    ulong rq_;   // condition that determines the order:
    // 0 ==> lexicographic order
    // 1 ==> Gray code
    // 2 ==> complemented enup order
    // 3 ==> complemented Eades-McKay sequence
    ulong nq_;   // whether to reverse order
  [--snip--]
    void (*visit_)(const comb_rec &);  // function to call with each combination
  [--snip--]
    void generate(void (*visit)(const comb_rec &), ulong rq, ulong nq=0)
    {
        visit_ = visit;
        rq_ = rq;
        nq_ = nq;
        ct_ = 0;
        rct_ = 0;
        next_rec(0);
    }
```

The recursion function is given in [FXT: comb/combination-rec.cc]:

```
void comb_rec::next_rec(ulong d)
{
    ulong r = k_ - d;  // number of elements remaining
    if ( 0==r )  visit_(*this);
    else
    {
        ulong rv1 = rv_[d-1];  // left neighbor
        bool q;
        switch ( rq_ )
        {
        case 0:  q = 1;  break;          //  0 ==> lexicographic order
```

```
        case 1:  q = !(d&1);  break;     //  1 ==> Gray code
        case 2:  q = rv1&1;  break;      //  2 ==> complemented enup order
        case 3:  q = (d^rv1)&1;  break;  //  3 ==> complemented Eades-McKay sequence
        default: q = 1;
        }
        q ^= nq_;  // reversed order if  nq == true
        if ( q )  // forward:
            for (ulong x=rv1+1; x<=n_-r; ++x)  { rv_[d] = x;  next_rec(d+1); }
        else     // backward:
            for (ulong x=n_-r; (long)x>=(long)rv1+1; --x)  { rv_[d] = x;  next_rec(d+1); }
    }
}
```

Figure 6.6-A was created with the program [FXT: comb/combination-rec-demo.cc]. The routine generates the combinations $\binom{32}{20}$ at a rate of about 32 million objects per second. The combinations $\binom{32}{12}$ are generated at a rate of 45 million objects per second.

Chapter 7

Compositions

The *compositions* of n into (at most) k parts ('k-compositions of n') are the ordered tuples $(x_0, x_1, \ldots, x_{k-1})$ where $x_0 + x_1 + \ldots + x_{k-1} = n$ and $0 \le x_i \le n$. Order matters: one 4-composition of 7 is $(0, 1, 5, 1)$, different ones are $(5, 0, 1, 1)$ and $(0, 5, 1, 1)$. To obtain the compositions of n into exactly k parts (where $k \le n$) generate the compositions of $n - k$ into k parts and add one to each position.

7.1 Co-lexicographic order

The compositions in co-lexicographic (colex) order are shown in figure 7.1-A. The implementation is [FXT: `class composition_colex` in comb/composition-colex.h]:

```
class composition_colex
{
public:
    ulong n_, k_;  // composition of n into k parts
    ulong *x_;     // data (k elements)
  [--snip--]
    void first()
    {
        x_[0] = n_;  // all in first position
        for (ulong k=1; k<k_; ++k)  x_[k] = 0;
    }
    void last()
    {
        for (ulong k=0; k<k_; ++k)  x_[k] = 0;
        x_[k_-1] = n_;  // all in last position
    }
  [--snip--]
```

The methods to compute the successor and predecessor are:

```
    ulong next()
    // Return position of rightmost change, return k with last composition.
    {
        ulong j = 0;
        while ( 0==x_[j] )  ++j;  // find first nonzero
        if ( j==k_-1 )  return k_;  // current composition is last
        ulong v = x_[j];  // value of first nonzero
        x_[j] = 0;        // set to zero
        x_[0] = v - 1;    // value-1 to first position
        ++j;
        ++x_[j];          // increment next position
        return  j;
    }
    ulong prev()
```

```
          composition   chg   combination              composition chg  combination
 1:     [ 3 . . . . ]  4      111....          1:     [ 7 . . ]  2      1111111..
 2:     [ 2 1 . . . ]  1      11.1...          2:     [ 6 1 . ]  1      111111.1.
 3:     [ 1 2 . . . ]  1      1.11...          3:     [ 5 2 . ]  1      11111.11.
 4:     [ . 3 . . . ]  1      .111...          4:     [ 4 3 . ]  1      1111.111.
 5:     [ 2 . 1 . . ]  2      11..1..          5:     [ 3 4 . ]  1      111.1111.
 6:     [ 1 1 1 . . ]  1      1.1.1..          6:     [ 2 5 . ]  1      11.11111.
 7:     [ . 2 1 . . ]  1      .11.1..          7:     [ 1 6 . ]  1      1.111111.
 8:     [ 1 . 2 . . ]  2      1..11..          8:     [ . 7 . ]  1      .1111111.
 9:     [ . 1 2 . . ]  1      .1.11..          9:     [ 6 . 1 ]  2      111111..1
10:     [ . . 3 . . ]  2      ..111..         10:     [ 5 1 1 ]  1      11111.1.1
11:     [ 2 . . 1 . ]  3      11...1.         11:     [ 4 2 1 ]  1      1111.11.1
12:     [ 1 1 . 1 . ]  1      1.1..1.         12:     [ 3 3 1 ]  1      111.111.1
13:     [ . 2 . 1 . ]  1      .11..1.         13:     [ 2 4 1 ]  1      11.1111.1
14:     [ 1 . 1 1 . ]  2      1..1.1.         14:     [ 1 5 1 ]  1      1.11111.1
15:     [ . 1 1 1 . ]  1      .1.1.1.         15:     [ . 6 1 ]  1      .111111.1
16:     [ . . 2 1 . ]  2      ..11.1.         16:     [ 5 . 2 ]  2      11111..11
17:     [ 1 . . 2 . ]  3      1...11.         17:     [ 4 1 2 ]  1      1111.1.11
18:     [ . 1 . 2 . ]  1      .1..11.         18:     [ 3 2 2 ]  1      111.11.11
19:     [ . . 1 2 . ]  2      ..1.11.         19:     [ 2 3 2 ]  1      11.111.11
20:     [ . . . 3 . ]  3      ...111.         20:     [ 1 4 2 ]  1      1.1111.11
21:     [ 2 . . . 1 ]  4      11....1         21:     [ . 5 2 ]  1      .11111.11
22:     [ 1 1 . . 1 ]  1      1.1...1         22:     [ 4 . 3 ]  2      1111..111
23:     [ . 2 . . 1 ]  1      .11...1         23:     [ 3 1 3 ]  1      111.1.111
24:     [ 1 . 1 . 1 ]  2      1..1..1         24:     [ 2 2 3 ]  1      11.11.111
25:     [ . 1 1 . 1 ]  1      .1.1..1         25:     [ 1 3 3 ]  1      1.111.111
26:     [ . . 2 . 1 ]  2      ..11..1         26:     [ . 4 3 ]  1      .1111.111
27:     [ 1 . . 1 1 ]  3      1...1.1         27:     [ 3 . 4 ]  2      111..1111
28:     [ . 1 . 1 1 ]  1      .1..1.1         28:     [ 2 1 4 ]  1      11.1.1111
29:     [ . . 1 1 1 ]  2      ..1.1.1         29:     [ 1 2 4 ]  1      1.11.1111
30:     [ . . . 2 1 ]  3      ...11.1         30:     [ . 3 4 ]  1      .111.1111
31:     [ 1 . . . 2 ]  4      1....11         31:     [ 2 . 5 ]  2      11..11111
32:     [ . 1 . . 2 ]  1      .1...11         32:     [ 1 1 5 ]  1      1.1.11111
33:     [ . . 1 . 2 ]  2      ..1..11         33:     [ . 2 5 ]  1      .11.11111
34:     [ . . . 1 2 ]  3      ...1.11         34:     [ 1 . 6 ]  2      1..111111
35:     [ . . . . 3 ]  4      ....111         35:     [ . 1 6 ]  1      .1.111111
                                              36:     [ . . 7 ]  2      ..1111111
```

Figure 7.1-A: The compositions of 3 into 5 parts in co-lexicographic order and positions of the rightmost change, and delta sets of the corresponding combinations (left); and the corresponding data for compositions of 7 into 3 parts (right). Dots denote zeros.

```
// Return position of rightmost change, return k with last composition.
{
    const ulong v = x_[0];   // value at first position
    if ( n_==v )  return k_; // current composition is first
    x_[0] = 0;          // set first position to zero
    ulong j = 1;
    while ( 0==x_[j] )  ++j;  // find next nonzero
    --x_[j];                  // decrement value
    x_[j-1] = 1 + v;          // set previous position
    return  j;
}
```

With each transition at most 3 entries are changed. The compositions of 10 into 30 parts (sparse case) are generated at a rate of about 110 million per second, the compositions of 30 into 10 parts (dense case) at about 200 million per second [FXT: comb/composition-colex-demo.cc]. With the dense case (corresponding to the right of figure 7.1-A) the computation is faster as the position to change is found earlier.

Optimized implementation

An implementation that is efficient also for the sparse case (that is, k much greater than n) is [FXT: `class composition_colex2` in comb/composition-colex2.h]. One additional variable `p0` records the position of

the first nonzero entry. The method to compute the successor is:

```
class composition_colex2
{
  [--snip--]
    ulong next()
    // Return position of rightmost change, return k with last composition.
    {
        ulong j = p0_;  // position of first nonzero
        if ( j==k_-1 )  return k_;  // current composition is last
        ulong v = x_[j];  // value of first nonzero
        x_[j] = 0;        // set to zero
        --v;
        x_[0] = v;        // value-1 to first position
        ++p0_;            // first nonzero one more right except ...
        if ( 0!=v )  p0_ = 0;   // ... if value v was not one
        ++j;
        ++x_[j];          // increment next position
        return  j;
    }
};
```

About 182 million compositions are generated per second, independent of either n and k [FXT: comb/composition-colex2-demo.cc]. With the line

```
#define COMP_COLEX2_MAX_ARRAY_LEN  128
```

just before the class definition an array is used instead of a pointer. The fixed array length limits the value of k so by default the line is commented out. Using an array gives great speedup, the rate is about 365 million per second (about 6 CPU cycles per update).

7.2 Co-lexicographic order for compositions into exactly k parts

The compositions of n into exactly k parts (where $k \geq n$) can be obtained from the compositions of $n-k$ into at most k parts as shown in figure 7.2-A. The listing was created with the program [FXT: comb/composition-ex-colex-demo.cc]. The compositions can be generated in co-lexicographic order using [FXT: `class composition_ex_colex` in comb/composition-ex-colex.h]:

```
class composition_ex_colex
{
public:
    ulong n_, k_;  // composition of n into exactly k parts
    ulong *x_;     // data (k elements)
    ulong nk1_;    // ==n-k+1
public:
    composition_ex_colex(ulong n, ulong k)
    // Must have n>=k
    {
        n_ = n;
        k_ = k;
        nk1_ = n - k + 1;  // must be >= 1
        if ( (long)nk1_ < 1 )  nk1_ = 1;  // avoid hang with invalid pair n,k
        x_ = new ulong[k_ + 1];
        x_[k] = 0;  // not one
        first();
    }
  [--snip--]
```

The variable `nk1_` is the maximal entry in the compositions:

```
    void first()
    {
        x_[0] = nk1_;  // all in first position
        for (ulong k=1; k<k_; ++k)  x_[k] = 1;
    }
    void last()
    {
```

```
               exact comp.  chg      composition
        1:    [ 4 1 1 1 1 ]  4      [ 3 . . . . ]
        2:    [ 3 2 1 1 1 ]  1      [ 2 1 . . . ]
        3:    [ 2 3 1 1 1 ]  1      [ 1 2 . . . ]
        4:    [ 1 4 1 1 1 ]  1      [ . 3 . . . ]
        5:    [ 3 1 2 1 1 ]  2      [ 2 . 1 . . ]
        6:    [ 2 2 2 1 1 ]  1      [ 1 1 1 . . ]
        7:    [ 1 3 2 1 1 ]  1      [ . 2 1 . . ]
        8:    [ 2 1 3 1 1 ]  2      [ 1 . 2 . . ]
        9:    [ 1 2 3 1 1 ]  1      [ . 1 2 . . ]
       10:    [ 1 1 4 1 1 ]  2      [ . . 3 . . ]
       11:    [ 3 1 1 2 1 ]  3      [ 2 . . 1 . ]
       12:    [ 2 2 1 2 1 ]  1      [ 1 1 . 1 . ]
       13:    [ 1 3 1 2 1 ]  1      [ . 2 . 1 . ]
       14:    [ 2 1 2 2 1 ]  2      [ 1 . 1 1 . ]
       15:    [ 1 2 2 2 1 ]  1      [ . 1 1 1 . ]
       16:    [ 1 1 3 2 1 ]  2      [ . . 2 1 . ]
       17:    [ 2 1 1 3 1 ]  3      [ 1 . . 2 . ]
       18:    [ 1 2 1 3 1 ]  1      [ . 1 . 2 . ]
       19:    [ 1 1 2 3 1 ]  2      [ . . 1 2 . ]
       20:    [ 1 1 1 4 1 ]  3      [ . . . 3 . ]
       21:    [ 3 1 1 1 2 ]  4      [ 2 . . . 1 ]
       22:    [ 2 2 1 1 2 ]  1      [ 1 1 . . 1 ]
       23:    [ 1 3 1 1 2 ]  1      [ . 2 . . 1 ]
       24:    [ 2 1 2 1 2 ]  2      [ 1 . 1 . 1 ]
       25:    [ 1 2 2 1 2 ]  1      [ . 1 1 . 1 ]
       26:    [ 1 1 3 1 2 ]  2      [ . . 2 . 1 ]
       27:    [ 2 1 1 2 2 ]  3      [ 1 . . 1 1 ]
       28:    [ 1 2 1 2 2 ]  1      [ . 1 . 1 1 ]
       29:    [ 1 1 2 2 2 ]  2      [ . . 1 1 1 ]
       30:    [ 1 1 1 3 2 ]  3      [ . . . 2 1 ]
       31:    [ 2 1 1 1 3 ]  4      [ 1 . . . 2 ]
       32:    [ 1 2 1 1 3 ]  1      [ . 1 . . 2 ]
       33:    [ 1 1 2 1 3 ]  2      [ . . 1 . 2 ]
       34:    [ 1 1 1 2 3 ]  3      [ . . . 1 2 ]
       35:    [ 1 1 1 1 4 ]  4      [ . . . . 3 ]
```

Figure 7.2-A: The compositions of $n = 8$ into exactly $k = 5$ parts (left) are obtained from the compositions of $n - k = 3$ into (at most) $k = 5$ parts (right). Co-lexicographic order, dots denote zeros.

```
    for (ulong k=0; k<k_; ++k)  x_[k] = 1;
    x_[k_-1] = nk1_;  // all in last position
}
```

The methods for computing the successor and predecessor are adaptations from the routines from the compositions into at most k parts:

```
ulong next()
// Return position of rightmost change, return k with last composition.
{
    ulong j = 0;
    while ( 1==x_[j] )  ++j;  // find first greater than one

    if ( j==k_ )  return k_;  // current composition is last

    ulong v = x_[j];  // value of first greater one
    x_[j] = 1;        // set to one
    x_[0] = v - 1;    // value-1 to first position
    ++j;
    ++x_[j];          // increment next position

    return  j;
}

ulong prev()
// Return position of rightmost change, return k with last composition.
{
    const ulong v = x_[0];   // value at first position

    if ( nk1_==v )  return k_; // current composition is first

    x_[0] = 1;          // set first position to one
    ulong j = 1;
    while ( 1==x_[j] )  ++j;  // find next greater than one
```

```
        --x_[j];                  // decrement value
        x_[j-1] = 1 + v;          // set previous position
        return  j;
    }
};
```

The routines are as fast as the generation into at most k parts with the corresponding parameters: the compositions of 40 into 10 parts are generated at about 200 million per second.

7.3 Compositions and combinations

```
              combination    delta set       composition
         1:    [ 0 1 2 ]       111...        [ 3 . . . ]
         2:    [ 0 2 3 ]       1.11..        [ 1 2 . . ]
         3:    [ 1 2 3 ]       .111..        [ . 3 . . ]
         4:    [ 0 1 3 ]       11.1..        [ 2 1 . . ]
         5:    [ 0 3 4 ]       1..11.        [ 1 . 2 . ]
         6:    [ 1 3 4 ]       .1.11.        [ . 1 2 . ]
         7:    [ 2 3 4 ]       ..111.        [ . . 3 . ]
         8:    [ 0 2 4 ]       1.1.1.        [ 1 1 1 . ]
         9:    [ 1 2 4 ]       .11.1.        [ . 2 1 . ]
        10:    [ 0 1 4 ]       11..1.        [ 2 . 1 . ]
        11:    [ 0 4 5 ]       1...11        [ 1 . . 2 ]
        12:    [ 1 4 5 ]       .1..11        [ . 1 . 2 ]
        13:    [ 2 4 5 ]       ..1.11        [ . . 1 2 ]
        14:    [ 3 4 5 ]       ...111        [ . . . 3 ]
        15:    [ 0 3 5 ]       1..1.1        [ 1 . 1 1 ]
        16:    [ 1 3 5 ]       .1.1.1        [ . 1 1 1 ]
        17:    [ 2 3 5 ]       ..11.1        [ . . 2 1 ]
        18:    [ 0 2 5 ]       1.1..1        [ 1 1 . 1 ]
        19:    [ 1 2 5 ]       .11..1        [ . 2 . 1 ]
        20:    [ 0 1 5 ]       11...1        [ 2 . . 1 ]
```

Figure 7.3-A: Combinations 6 choose 3 (left) and the corresponding compositions of 3 into 4 parts (right). Note that while the sequence of combinations has a minimal-change property the corresponding sequence of compositions does not. Dots denote zeros.

Figure 7.3-A shows the correspondence between compositions and combinations. The listing was generated using the program [FXT: comb/comb2comp-demo.cc]. Entries in the left column are combinations of 3 parts out of 6. The middle column is the representation of the combinations as delta sets. It also is a binary representation of a composition: A run of r consecutive ones corresponds to an entry r in the composition at the right.

Now write $P(n, k)$ for the compositions of n into (at most) k parts and $C(N, K)$ for the combination $\binom{N}{K}$: A composition of n into at most k parts corresponds to a combination of $K = n$ parts from $N = n+k-1$ elements, symbolically,

$$P(n,\, k) \quad \leftrightarrow \quad C(N,\, K) \;=\; C(n+k-1,\, n) \tag{7.3-1a}$$

A combination of K elements out of N corresponds to a composition of n into at most k parts where $n = K$ and $k = N - K + 1$:

$$C(N,\, K) \quad \leftrightarrow \quad P(n,\, k) \;=\; P(K,\, N-K+1) \tag{7.3-1b}$$

Routines for the conversion between combinations and compositions are given in [FXT: comb/comp2comb.h]. The following routine converts a composition into the corresponding combination:

```
inline void comp2comb(const ulong *p, ulong k, ulong *c)
// Convert composition P(*, k) in p[] to combination in c[]
{
    for (ulong j=0,i=0,z=0; j<k; ++j)
```

```
    {
        ulong pj = p[j];
        for (ulong w=0; w<pj; ++w)   c[i++] = z++;
        ++z;
    }
}
```

Conversion of a combination into the corresponding composition:

```
inline void comb2comp(const ulong *c, ulong N, ulong K, ulong *p)
// Convert combination C(N, K) in c[] to composition P(*,k) in p[]
// Must have: K>0
{
    ulong k = N-K+1;
    for (ulong z=0; z<k; ++z)  p[z] = 0;
    --k;
    ulong c1 = N;
    while ( K-- )
    {
        ulong c0 = c[K];
        ulong d = c1 - c0;
        k -= (d-1);
        ++p[k];
        c1 = c0;
    }
}
```

7.4 Minimal-change orders

A minimal-change order (Gray code) for compositions is such that with each transition one entry is increased by one and another is decreased by one. A recursion for the compositions $P(n,k)$ of n into k parts in lexicographic order is

$$P(n,k) \quad = \quad \begin{matrix} [0\,.\,P(n-0,k-1)] \\ [1\,.\,P(n-1,k-1)] \\ [2\,.\,P(n-2,k-1)] \\ [3\,.\,P(n-3,k-1)] \\ [4\,.\,P(n-4,k-1)] \\ [\quad\vdots\qquad\quad] \\ [n\,.\,P(0,k-1)\quad\;] \end{matrix} \tag{7.4-1}$$

A simple variation gives a Gray code, we change the direction if the element is even:

$$P(n,k) \quad = \quad \begin{matrix} [0\,.\,P^{\mathbf{R}}(n-0,k-1)] \\ [1\,.\,P(n-1,k-1)\;\;] \\ [2\,.\,P^{\mathbf{R}}(n-2,k-1)] \\ [3\,.\,P(n-3,k-1)\;\;] \\ [4\,.\,P^{\mathbf{R}}(n-4,k-1)] \\ [\quad\vdots\qquad\quad] \end{matrix} \tag{7.4-2}$$

The ordering is shown in figure 7.4-A (left), the corresponding combinations are in the (reversed) enup order from section 6.5.2 on page 176. If we change directions at the odd elements

$$P(n,k) \quad = \quad \begin{matrix} [0\,.\,P(n-0,k-1)\;\;] \\ [1\,.\,P^{\mathbf{R}}(n-1,k-1)] \\ [2\,.\,P(n-2,k-1)\;\;] \\ [3\,.\,P^{\mathbf{R}}(n-3,k-1)] \\ [4\,.\,P(n-4,k-1)\;\;] \\ [\quad\vdots\qquad\quad] \end{matrix} \tag{7.4-3}$$

	composition	combination		composition	combination
1:	[. . . 3 .]	...111.	[3 4 5]	[3 . . . .]	111....	[0 1 2]
2:	[. 1 . 2 .]	.1..11.	[1 4 5]	[2 1 . . .]	11.1...	[0 1 3]
3:	[1 . . 2 .]	1...11.	[0 4 5]	[1 2 . . .]	1.11...	[0 2 3]
4:	[. . 1 2 .]	..1.11.	[2 4 5]	[. 3 . . .]	.111...	[1 2 3]
5:	[. . 2 1 .]	..11.1.	[2 3 5]	[. 2 1 . .]	.11.1..	[1 2 4]
6:	[. 1 1 1 .]	.1.1.1.	[1 3 5]	[1 1 1 . .]	1.1.1..	[0 2 4]
7:	[1 . 1 1 .]	1..1.1.	[0 3 5]	[2 . 1 . .]	11..1..	[0 1 4]
8:	[2 . . 1 .]	11...1.	[0 1 5]	[1 . 2 . .]	1..11..	[0 3 4]
9:	[1 1 . 1 .]	1.1..1.	[0 2 5]	[. 1 2 . .]	.1.11..	[1 3 4]
10:	[. 2 . 1 .]	.11..1.	[1 2 5]	[. . 3 . .]	..111..	[2 3 4]
11:	[. 3 . . .]	.111...	[1 2 3]	[. . 2 1 .]	..11.1.	[2 3 5]
12:	[1 2 . . .]	1.11...	[0 2 3]	[1 . 1 1 .]	1..1.1.	[0 3 5]
13:	[2 1 . . .]	11.1...	[0 1 3]	[. 1 1 1 .]	.1.1.1.	[1 3 5]
14:	[3 . . . .]	111....	[0 1 2]	[. 2 . 1 .]	.11..1.	[1 2 5]
15:	[2 . 1 . .]	11..1..	[0 1 4]	[1 1 . 1 .]	1.1..1.	[0 2 5]
16:	[1 1 1 . .]	1.1.1..	[0 2 4]	[2 . . 1 .]	11...1.	[0 1 5]
17:	[. 2 1 . .]	.11.1..	[1 2 4]	[1 . . 2 .]	1...11.	[0 4 5]
18:	[. 1 2 . .]	.1.11..	[1 3 4]	[. 1 . 2 .]	.1..11.	[1 4 5]
19:	[1 . 2 . .]	1..11..	[0 3 4]	[. . 1 2 .]	..1.11.	[2 4 5]
20:	[. . 3 . .]	..111..	[2 3 4]	[. . . 3 .]	...111.	[3 4 5]
21:	[. . 2 . 1]	..11..1	[2 3 6]	[. . . 2 1]	...11.1	[3 4 6]
22:	[. 1 1 . 1]	.1.1..1	[1 3 6]	[1 . . 1 1]	1...1.1	[0 4 6]
23:	[1 . 1 . 1]	1..1..1	[0 3 6]	[. 1 . 1 1]	.1..1.1	[1 4 6]
24:	[2 . . . 1]	11....1	[0 1 6]	[. . 1 1 1]	..1.1.1	[2 4 6]
25:	[1 1 . . 1]	1.1...1	[0 2 6]	[. . 2 . 1]	..11..1	[2 3 6]
26:	[. 2 . . 1]	.11...1	[1 2 6]	[1 . 1 . 1]	1..1..1	[0 3 6]
27:	[. 1 . 1 1]	.1..1.1	[1 4 6]	[. 1 1 . 1]	.1.1..1	[1 3 6]
28:	[1 . . 1 1]	1...1.1	[0 4 6]	[. 2 . . 1]	.11...1	[1 2 6]
29:	[. . 1 1 1]	..1.1.1	[2 4 6]	[1 1 . . 1]	1.1...1	[0 2 6]
30:	[. . . 2 1]	...11.1	[3 4 6]	[2 . . . 1]	11....1	[0 1 6]
31:	[. . . 1 2]	...1.11	[3 5 6]	[1 . . . 2]	1....11	[0 5 6]
32:	[. 1 . . 2]	.1...11	[1 5 6]	[. 1 . . 2]	.1...11	[1 5 6]
33:	[1 . . . 2]	1....11	[0 5 6]	[. . 1 . 2]	..1..11	[2 5 6]
34:	[. . 1 . 2]	..1..11	[2 5 6]	[. . . 1 2]	...1.11	[3 5 6]
35:	[. . . . 3]	111	[4 5 6]	[. . . . 3]	111	[4 5 6]

Figure 7.4-A: Compositions of 3 into 5 parts and the corresponding combinations as delta sets and sets in two minimal-change orders: order with enup moves (left) and order with modulo moves (right). The ordering by enup moves is a two-close Gray code. Dots denote zeros.

then we obtain an ordering (right of figure 7.4-A) corresponding to the combinations are in the (reversed) Eades-McKay order from section 6.4 on page 172. The listings were created with the program [FXT: comb/composition-gray-rec-demo.cc].

Gray codes for combinations correspond to Gray codes for combinations where no element in the delta set crosses another. The standard Gray code for combinations does not lead to a Gray code for compositions as shown in figure 7.3-A on page 187. When the directions in the recursions are always changed then the compositions correspond to combinations that have the complemented delta sets of the standard Gray code in reversed order.

Orderings where the changes involve just one pair of adjacent entries (shown in figure 7.4-B) correspond to the complemented strong Gray codes for combinations. The amount of change is greater than one in general. The listings were created with the program [FXT: comb/combination-rec-demo.cc], see section 6.6 on page 179.

	combination		composition
1:	[0 5 6]	1....11	[1 . . . 2]
2:	[0 4 6]	1...1.1	[1 . . 1 1]
3:	[0 4 5]	1...11.	[1 . . 2 .]
4:	[0 3 4]	1..11..	[1 . 2 . .]
5:	[0 3 5]	1..1.1.	[1 . 1 1 .]
6:	[0 3 6]	1..1..1	[1 . 1 . 1]
7:	[0 2 6]	1.1...1	[1 1 . . 1]
8:	[0 2 5]	1.1..1.	[1 1 . 1 .]
9:	[0 2 4]	1.1.1..	[1 1 1 . .]
10:	[0 2 3]	1.11...	[1 2 . . .]
11:	[0 1 2]	111....	[3 . . . .]
12:	[0 1 3]	11.1...	[2 1 . . .]
13:	[0 1 4]	11..1..	[2 . 1 . .]
14:	[0 1 5]	11...1.	[2 . . 1 .]
15:	[0 1 6]	11....1	[2 . . . 1]
16:	[1 2 6]	.11...1	[. 2 . . 1]
17:	[1 2 5]	.11..1.	[. 2 . 1 .]
18:	[1 2 4]	.11.1..	[. 2 1 . .]
19:	[1 2 3]	.111...	[. 3 . . .]
20:	[1 3 4]	.1.11..	[. 1 2 . .]
21:	[1 3 5]	.1.1.1.	[. 1 1 1 .]
22:	[1 3 6]	.1.1..1	[. 1 1 . 1]
23:	[1 4 6]	.1..1.1	[. 1 . 1 1]
24:	[1 4 5]	.1..11.	[. 1 . 2 .]
25:	[1 5 6]	.1...11	[. 1 . . 2]
26:	[2 5 6]	..1..11	[. . 1 . 2]
27:	[2 4 6]	..1.1.1	[. . 1 1 1]
28:	[2 4 5]	..1.11.	[. . 1 2 .]
29:	[2 3 4]	..111..	[. . 3 . .]
30:	[2 3 5]	..11.1.	[. . 2 1 .]
31:	[2 3 6]	..11..1	[. . 2 . 1]
32:	[3 4 6]	...11.1	[. . . 2 1]
33:	[3 4 5]	...111.	[. . . 3 .]
34:	[3 5 6]	...1.11	[. . . 1 2]
35:	[4 5 6]	111	[. . . . 3]

	combination		composition
1:	[0 1 2]	111....	[3 . . . .]
2:	[0 1 3]	11.1...	[2 1 . . .]
3:	[0 1 4]	11..1..	[2 . 1 . .]
4:	[0 1 5]	11...1.	[2 . . 1 .]
5:	[0 1 6]	11....1	[2 . . . 1]
6:	[0 2 6]	1.1...1	[1 1 . . 1]
7:	[0 2 5]	1.1..1.	[1 1 . 1 .]
8:	[0 2 4]	1.1.1..	[1 1 1 . .]
9:	[0 2 3]	1.11...	[1 2 . . .]
10:	[0 3 4]	1..11..	[1 . 2 . .]
11:	[0 3 5]	1..1.1.	[1 . 1 1 .]
12:	[0 3 6]	1..1..1	[1 . 1 . 1]
13:	[0 4 6]	1...1.1	[1 . . 1 1]
14:	[0 4 5]	1...11.	[1 . . 2 .]
15:	[0 5 6]	1....11	[1 . . . 2]
16:	[1 5 6]	.1...11	[. 1 . . 2]
17:	[1 4 6]	.1..1.1	[. 1 . 1 1]
18:	[1 4 5]	.1..11.	[. 1 . 2 .]
19:	[1 3 4]	.1.11..	[. 1 2 . .]
20:	[1 3 5]	.1.1.1.	[. 1 1 1 .]
21:	[1 3 6]	.1.1..1	[. 1 1 . 1]
22:	[1 2 6]	.11...1	[. 2 . . 1]
23:	[1 2 5]	.11..1.	[. 2 . 1 .]
24:	[1 2 4]	.11.1..	[. 2 1 . .]
25:	[1 2 3]	.111...	[. 3 . . .]
26:	[2 3 4]	..111..	[. . 3 . .]
27:	[2 3 5]	..11.1.	[. . 2 1 .]
28:	[2 3 6]	..11..1	[. . 2 . 1]
29:	[2 4 6]	..1.1.1	[. . 1 1 1]
30:	[2 4 5]	..1.11.	[. . 1 2 .]
31:	[2 5 6]	..1..11	[. . 1 . 2]
32:	[3 5 6]	...1.11	[. . . 1 2]
33:	[3 4 6]	...11.1	[. . . 2 1]
34:	[3 4 5]	...111.	[. . . 3 .]
35:	[4 5 6]	111	[. . . . 3]

Figure 7.4-B: The (reversed) complemented enup ordering (left) and Eades-McKay sequence (right) for combinations correspond to compositions where only two adjacent entries change with each transition, but by more than one in general.

Chapter 8

Subsets

This section gives algorithms to generate all subsets of a set of n elements. There are 2^n subsets, including the empty set.

8.1 Lexicographic order

```
 1:    1....   {0}                  1....   {0}
 2:    11...   {0, 1}               .1...   {1}
 3:    111..   {0, 1, 2}            11...   {0, 1}
 4:    1111.   {0, 1, 2, 3}         ..1..   {2}
 5:    11111   {0, 1, 2, 3, 4}      1.1..   {0, 2}
 6:    111.1   {0, 1, 2, 4}         .11..   {1, 2}
 7:    11.1.   {0, 1, 3}            111..   {0, 1, 2}
 8:    11.11   {0, 1, 3, 4}         ...1.   {3}
 9:    11..1   {0, 1, 4}            1..1.   {0, 3}
10:    1.1..   {0, 2}               .1.1.   {1, 3}
11:    1.11.   {0, 2, 3}            11.1.   {0, 1, 3}
12:    1.111   {0, 2, 3, 4}         ..11.   {2, 3}
13:    1.1.1   {0, 2, 4}            1.11.   {0, 2, 3}
14:    1..1.   {0, 3}               .111.   {1, 2, 3}
15:    1..11   {0, 3, 4}            1111.   {0, 1, 2, 3}
16:    1...1   {0, 4}               ....1   {4}
17:    .1...   {1}                  1...1   {0, 4}
18:    .11..   {1, 2}               .1..1   {1, 4}
19:    .111.   {1, 2, 3}            11..1   {0, 1, 4}
20:    .1111   {1, 2, 3, 4}         ..1.1   {2, 4}
21:    .11.1   {1, 2, 4}            1.1.1   {0, 2, 4}
22:    .1.1.   {1, 3}               .11.1   {1, 2, 4}
23:    .1.11   {1, 3, 4}            111.1   {0, 1, 2, 4}
24:    .1..1   {1, 4}               ...11   {3, 4}
25:    ..1..   {2}                  1..11   {0, 3, 4}
26:    ..11.   {2, 3}               .1.11   {1, 3, 4}
27:    ..111   {2, 3, 4}            11.11   {0, 1, 3, 4}
28:    ..1.1   {2, 4}               ..111   {2, 3, 4}
29:    ...1.   {3}                  1.111   {0, 2, 3, 4}
30:    ...11   {3, 4}               .1111   {1, 2, 3, 4}
31:    ....1   {4}                  11111   {0, 1, 2, 3, 4}
```

Figure 8.1-A: Non-empty subsets of a five element set in lexicographic order for the sets (left), and in lexicographic order for the delta sets (right).

The (nonempty) subsets of a set of five elements in lexicographic order are shown in figure 8.1-A. Note that the lexicographic order is different for the set and the delta set representation.

8.1.1 Generation as delta sets

The listing on the right side of figure 8.1-A is with respect to the delta sets. It was created with the program [FXT: comb/subset-deltalex-demo.cc] which uses [FXT: `class subset_deltalex` in comb/subset-deltalex.h]. The algorithm for the computation of the successor is binary counting:

```
class subset_deltalex
{
public:
    ulong *d_;  // subset as delta set
    ulong n_;   // subsets of the  n-set {0,1,2,...,n-1}

public:
    subset_deltalex(ulong n)
    {
        n_ = n;
        d_ = new ulong[n+1];
        d_[n] = 0;  // sentinel
        first();
    }

    ~subset_deltalex()
    { delete [] d_; }

    void first()
    { for (ulong k=0; k<n_; ++k)  d_[k] = 0; }

    bool next()
    {
        ulong k = 0;
        while ( d_[k]==1 )  { d_[k]=0;  ++k; }

        if ( k==n_ )  return false;
        else
        {
            d_[k] = 1;
            return true;
        }
    }

    const ulong * data()  const { return d_; }
};
```

About 180 million subsets per second are generated. A bit-level algorithm to compute the subsets in lexicographic order is given in section 1.27 on page 64.

8.1.2 Generation as sets

The lexicographic order with respect to the set representation is shown at the left side of figure 8.1-A. The routines in [FXT: `class subset_lex` in comb/subset-lex.h] compute the non-empty sets in the corresponding representation:

```
class subset_lex
{
public:
    ulong *x_;  // subset of {0,1,2,...,n-1}
    ulong n_;   // number of elements in set
    ulong k_;   // index of last element in subset
    // Number of elements in subset == k+1

public:
    subset_lex(ulong n)
    {
        n_ = n;
        x_ = new ulong[n_];
        first();
    }

    ~subset_lex()  { delete [] x_; }

    ulong first()
    {
        k_ = 0;
        x_[0] = 0;
        return  k_ + 1;
    }
```

```
    ulong last()
    {
        k_ = 0;
        x_[0] = n_ - 1;
        return  k_ + 1;
    }
  [--snip--]
```

The methods `next()` and `prev()` compute the successor and predecessor, respectively:

```
    ulong next()
    // Generate next subset
    // Return number of elements in subset
    // Return zero if current == last
    {
        if ( x_[k_] == n_-1 )  // last element is max ?
        {
            if ( k_==0 )  { first();  return 0; }

            --k_;      // remove last element
            x_[k_]++;  // increase last element
        }
        else  // add next element from set:
        {
            ++k_;
            x_[k_] = x_[k_-1] + 1;
        }

        return  k_ + 1;
    }

    ulong prev()
    // Generate previous subset
    // Return number of elements in subset
    // Return zero if current == first
    {
        if ( k_ == 0 )  // only one element ?
        {
            if ( x_[0]==0 )  { last();  return 0; }

            x_[0]--;  // decr first element
            x_[++k_] = n_ - 1;     // add element
        }
        else
        {
            if ( x_[k_] == x_[k_-1]+1 )  --k_;  // remove last element
            else
            {
                x_[k_]--;  // decr last element
                x_[++k_] = n_ - 1;     // add element
            }
        }

        return  k_ + 1;
    }

    const ulong * data()  const { return x_; }
};
```

More than 235 million subsets per second are generated [FXT: comb/subset-lex-demo.cc]. A generalization with mixed radix numbers is described in section 9.3 on page 213.

8.2 Minimal-change order

8.2.1 Generation as delta sets

The subsets of a set with 5 elements in minimal-change order are shown in figure 8.2-A. The implementation [FXT: `class subset_gray_delta` in comb/subset-gray-delta.h] uses the Gray code of binary words and updates the position corresponding to the bit that changes in the Gray code:

```
class subset_gray_delta
// Subsets of the set {0,1,2,...,n-1} in minimal-change (Gray code) order.
{
public:
    ulong *x_;  // current subset as delta-set
```

```
 0:  .....  {}                          0:  11111  { 0, 1, 2, 3, 4 }
 1:  1....  {0}                         1:  .1111  { 1, 2, 3, 4 }
 2:  11...  {0, 1}                      2:  ..111  { 2, 3, 4 }
 3:  .1...  {1}                         3:  1.111  { 0, 2, 3, 4 }
 4:  .11..  {1, 2}                      4:  1..11  { 0, 3, 4 }
 5:  111..  {0, 1, 2}                   5:  ...11  { 3, 4 }
 6:  1.1..  {0, 2}                      6:  .1.11  { 1, 3, 4 }
 7:  ..1..  {2}                         7:  11.11  { 0, 1, 3, 4 }
 8:  ..11.  {2, 3}                      8:  11..1  { 0, 1, 4 }
 9:  1.11.  {0, 2, 3}                   9:  .1..1  { 1, 4 }
10:  1111.  {0, 1, 2, 3}               10:  ....1  { 4 }
11:  .111.  {1, 2, 3}                  11:  1...1  { 0, 4 }
12:  .1.1.  {1, 3}                     12:  1.1.1  { 0, 2, 4 }
13:  11.1.  {0, 1, 3}                  13:  ..1.1  { 2, 4 }
14:  1..1.  {0, 3}                     14:  .11.1  { 1, 2, 4 }
15:  ...1.  {3}                        15:  111.1  { 0, 1, 2, 4 }
16:  ...11  {3, 4}                     16:  111..  { 0, 1, 2 }
17:  1..11  {0, 3, 4}                  17:  .11..  { 1, 2 }
18:  11.11  {0, 1, 3, 4}               18:  ..1..  { 2 }
19:  .1.11  {1, 3, 4}                  19:  1.1..  { 0, 2 }
20:  .1111  {1, 2, 3, 4}               20:  1....  { 0 }
21:  11111  {0, 1, 2, 3, 4}            21:  .....  {  }
22:  1.111  {0, 2, 3, 4}               22:  .1...  { 1 }
23:  ..111  {2, 3, 4}                  23:  11...  { 0, 1 }
24:  ..1.1  {2, 4}                     24:  11.1.  { 0, 1, 3 }
25:  1.1.1  {0, 2, 4}                  25:  .1.1.  { 1, 3 }
26:  111.1  {0, 1, 2, 4}               26:  ...1.  { 3 }
27:  .11.1  {1, 2, 4}                  27:  1..1.  { 0, 3 }
28:  .1..1  {1, 4}                     28:  1.11.  { 0, 2, 3 }
29:  11..1  {0, 1, 4}                  29:  ..11.  { 2, 3 }
30:  1...1  {0, 4}                     30:  .111.  { 1, 2, 3 }
31:  ....1  {4}                        31:  1111.  { 0, 1, 2, 3 }
```

Figure 8.2-A: The subsets of the set {0, 1, 2, 3, 4} in minimal-change order (left), and complemented minimal-change order (right). The changes are on the same places for both orders.

```
    ulong n_;   // number of elements in set <= BITS_PER_LONG
    ulong j_;   // position of last change
    ulong ct_;  // gray_code(ct_) corresponds to the current subset
    ulong mct_; // max value of ct.
public:
    subset_gray_delta(ulong n)
    {
        n_ = (n ? n : 1);  // not zero
        x_ = new ulong[n_];
        mct_ = (1UL<<n) - 1;
        first(0);
    }
    ~subset_gray_delta()  { delete [] x_; }
```

In the initializer one can choose whether the first set is the empty or the full set (left and right of figure 8.2-A):

```
    void first(ulong v=0)
    {
        ct_ = 0;
        j_ = n_ - 1;
        for (ulong j=0; j<n_; ++j)  x_[j] = v;
    }
    const ulong * data()  const  { return x_; }
    ulong pos()  const { return j_; }
    ulong current()  const  { return ct_; }
    ulong next()
    // Return position of change, return n with last subset
    {
        if ( ct_ == mct_ )  { return n_; }
```

```
        ++ct_;
        j_ = lowest_bit_idx( ct_ );
        x_[j_] ^= 1;
        return  j_;
    }
    ulong prev()
    // Return position of change, return n with first subset
    {
        if ( ct_ == 0 )  { return n_; }
        j_ = lowest_bit_idx( ct_ );
        x_[j_] ^= 1;
        --ct_;
        return  j_;
    }
};
```

About 130 million subsets are generated per second [FXT: comb/subset-gray-delta-demo.cc].

8.2.2 Generation as sets

The class [FXT: `class subset_gray` in comb/subset-gray.h] generates the subsets of $\{1, 2, \ldots, n\}$ as sets. The underlying idea is described in section 1.15.3 on page 38.

```
class subset_gray
// Subsets of the set {1,2,...,n} in minimal-change (Gray code) order.
{
public:
    ulong *x_;  // data k-subset of {1,2,...,n} in x[1,...,k]
    ulong n_;   // subsets of n-set
    ulong k_;   // number of elements in subset
public:
    subset_gray(ulong n)
    {
        n_ = n;
        x_ = new ulong[n_+1];
        x_[0] = 0;
        first();
    }
    ~subset_gray()  { delete [] x_; }
    ulong first()  { k_ = 0;  return k_; }
    ulong last()  { x_[1] = 1;  k_ = 1;  return k_; }
    const ulong * data() const { return x_+1; }
    const ulong num() const { return k_; }
private:
    ulong next_even()
    {
        if ( x_[k_]==n_ ) // remove n (from end):
        {
            --k_;
        }
        else  // append n:
        {
            ++k_;
            x_[k_] = n_;
        }
        return  k_;
    }
    ulong next_odd()
    {
        if ( x_[k_]-1==x_[k_-1] ) // remove x[k]-1 (from position k-1):
        {
            x_[k_-1] = x_[k_];
            --k_;
        }
        else // insert x[k]-1 as second last element:
        {
            x_[k_+1] = x_[k_];
```

```
            --x_[k_];
            ++k_;
        }
        return  k_;
    }
public:
    ulong next()
    {
        if ( 0==(k_&1 ) ) return next_even();
        else              return next_odd();
    }

    ulong prev()
    {
        if ( 0==(k_&1 ) )  // k even
        {
            if ( 0==k_ )  return last();
            return next_odd();
        }
        else  return next_even();
    }
};
```

More than 160 million subsets are generated per second [FXT: comb/subset-gray-demo.cc]. The algorithm to generate the successor is given in [141].

8.2.3 Computing just the positions of change

The following algorithm computes only the locations of the changes, it is given in [41]. It can also be obtained as a specialization (for radix 2) of the loopless algorithm for computing a Gray code ordering of mixed radix numbers given section 9.2 on page 210 [FXT: `class ruler_func` in comb/ruler-func.h]:

```
class ruler_func
// Ruler function sequence: 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 ...
{
public:
    ulong *f_;  // focus pointer
    ulong n_;
public:
    ruler_func(ulong n)
    {
        n_ = n;
        f_ = new ulong[n+2];
        first();
    }

    ~ruler_func()  { delete [] f_; }

    void first()  { for (ulong k=0; k<n_+2; ++k)  f_[k] = k; }

    ulong next()
    {
        const ulong j = f_[0];
        //  if ( j==n_ )  { first(); return n_; }  // leave to user
        f_[0] = 0;
        const ulong j1 = j+1;
        f_[j] = f_[j1];
        f_[j1] = j1;
        return j;
    }
};
```

The sequence of positions is generated at a rate of about 290 million per second [FXT: comb/ruler-func-demo.cc].

```
 0:    {0,  ,  ,  ,  }   #=1    {0}
 1:    { , 1,  ,  ,  }   #=1    {1}
 2:    { ,  , 2,  ,  }   #=1    {2}
 3:    { ,  ,  , 3,  }   #=1    {3}
 4:    {0,  ,  ,  , 4}   #=2    {0, 4}
 5:    {0, 1,  ,  ,  }   #=2    {0, 1}
 6:    { , 1, 2,  ,  }   #=2    {1, 2}
 7:    { ,  , 2, 3,  }   #=2    {2, 3}
 8:    {0,  ,  , 3, 4}   #=3    {0, 3, 4}
 9:    { , 1,  ,  , 4}   #=2    {1, 4}
10:    {0,  , 2,  ,  }   #=2    {0, 2}
11:    { , 1,  , 3,  }   #=2    {1, 3}
12:    { ,  , 2,  , 4}   #=2    {2, 4}
13:    {0,  ,  , 3,  }   #=2    {0, 3}
14:    {0, 1,  ,  , 4}   #=3    {0, 1, 4}
15:    {0, 1, 2,  ,  }   #=3    {0, 1, 2}
16:    { , 1, 2, 3,  }   #=3    {1, 2, 3}
17:    {0,  , 2, 3, 4}   #=4    {0, 2, 3, 4}
18:    { , 1,  , 3, 4}   #=3    {1, 3, 4}
19:    {0,  , 2,  , 4}   #=3    {0, 2, 4}
20:    {0, 1,  , 3,  }   #=3    {0, 1, 3}
21:    { , 1, 2,  , 4}   #=3    {1, 2, 4}
22:    {0,  , 2, 3,  }   #=3    {0, 2, 3}
23:    {0, 1,  , 3, 4}   #=4    {0, 1, 3, 4}
24:    {0, 1, 2,  , 4}   #=4    {0, 1, 2, 4}
25:    {0, 1, 2, 3,  }   #=4    {0, 1, 2, 3}
26:    {0, 1, 2, 3, 4}   #=5    {0, 1, 2, 3, 4}
27:    { , 1, 2, 3, 4}   #=4    {1, 2, 3, 4}
28:    { ,  , 2, 3, 4}   #=3    {2, 3, 4}
29:    { ,  ,  , 3, 4}   #=2    {3, 4}
30:    { ,  ,  ,  , 4}   #=1    {4}
31:    { ,  ,  ,  ,  }   #=0    {}
```

Figure 8.3-A: Subsets of a five element set in an order corresponding to a De Bruijn sequence.

8.3 Ordering with De Bruijn sequences

A curious sequence of all subsets of a given set can be generated using a binary *De Bruijn sequence* that is a cyclical sequence of zeros and ones that contains each n-bit word once. In figure 8.3-A the empty places of the subsets are included to make the nice property apparent. The ordering has the *single track* property: each column in this (delta set) representation is a circular shift of the first column. The listing was created with the program [FXT: comb/subset-debruijn-demo.cc]. The underlying De Bruijn sequence is

```
0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1
```

(rotated left in the example so that the empty set appears at the end). Each subset is made from its predecessor by shifting it to the right and inserting the current element from the sequence.

The implementation [FXT: `class subset_debruijn` in comb/subset-debruijn.h] uses [FXT: `class debruijn` in comb/debruijn.h] (which in turn uses [FXT: `class necklace` in comb/necklace.h]). An algorithm for the generation of binary De Bruijn sequences is given in section 39.1 on page 833.

Successive subsets differ in many elements if the sequency (see section 1.16 on page 42) is big. Using the 'sequency-complemented' subsets (see end of section 1.16) we obtain an ordering where more elements change with small sequencies as shown in figure 8.3-B. This ordering corresponds to the complement-shift sequence of section 19.2.3 on page 361.

```
 0:    { ,  , 2,  , 4}   #=2    {2, 4}
 1:    {0, 1, 2,  , 4}   #=4    {0, 1, 2, 4}
 2:    {0,  ,  ,  , 4}   #=2    {0, 4}
 3:    {0,  , 2, 3, 4}   #=4    {0, 2, 3, 4}
 4:    { ,  , 2,  ,  }   #=1    {2}
 5:    { , 1, 2,  , 4}   #=3    {1, 2, 4}
 6:    {0, 1,  ,  , 4}   #=3    {0, 1, 4}
 7:    {0,  ,  , 3, 4}   #=3    {0, 3, 4}
 8:    { ,  , 2, 3,  }   #=2    {2, 3}
 9:    {0, 1, 2,  ,  }   #=3    {0, 1, 2}
10:    { ,  ,  ,  , 4}   #=1    {4}
11:    {0, 1, 2, 3, 4}   #=5    {0, 1, 2, 3, 4}
12:    {0,  ,  ,  ,  }   #=1    {0}
13:    { ,  , 2, 3, 4}   #=3    {2, 3, 4}
14:    { , 1, 2,  ,  }   #=2    {1, 2}
15:    { , 1,  ,  , 4}   #=2    {1, 4}
16:    {0, 1,  , 3, 4}   #=4    {0, 1, 3, 4}
17:    { ,  ,  , 3,  }   #=1    {3}
18:    {0, 1, 2, 3,  }   #=4    {0, 1, 2, 3}
19:    { ,  ,  ,  ,  }   #=0    {}
20:    { , 1, 2, 3, 4}   #=4    {1, 2, 3, 4}
21:    {0, 1,  ,  ,  }   #=2    {0, 1}
22:    { ,  ,  , 3, 4}   #=2    {3, 4}
23:    { , 1, 2, 3,  }   #=3    {1, 2, 3}
24:    { , 1,  ,  ,  }   #=1    {1}
25:    { , 1,  , 3, 4}   #=3    {1, 3, 4}
26:    { , 1,  , 3,  }   #=2    {1, 3}
27:    {0, 1,  , 3,  }   #=3    {0, 1, 3}
28:    {0,  ,  , 3,  }   #=2    {0, 3}
29:    {0,  , 2, 3,  }   #=3    {0, 2, 3}
30:    {0,  , 2,  ,  }   #=2    {0, 2}
31:    {0,  , 2,  , 4}   #=3    {0, 2, 4}
```

Figure 8.3-B: Subsets of a five element set in alternative order corresponding to a De Bruijn sequence. The elements 0, 2, and 4 are present exactly if they are not in figure 8.3-A.

```
 1:  .....1   1    17:  1..111   4    33:  ....11   2    49:  ...111   3
 2:  ....1.   1    18:  ...1.1   2    34:  ...11.   2    50:  ..111.   3
 3:  ...1..   1    19:  ..1.1.   2    35:  ..11..   2    51:  .111..   3
 4:  ..1...   1    20:  .1.1..   2    36:  .11...   2    52:  111...   3
 5:  .1....   1    21:  1.1...   2    37:  11....   2    53:  111..1   4
 6:  1.....   1    22:  1.1..1   3    38:  11...1   3    54:  .111.1   4
 7:  1....1   2    23:  .1.1.1   3    39:  .11..1   3    55:  111.1.   4
 8:  .1...1   2    24:  1.1.1.   3    40:  11..1.   3    56:  111.11   5
 9:  1...1.   2    25:  1.1.11   4    41:  11..11   4    57:  ..1111   4
10:  1...11   3    26:  ..1.11   3    42:  ..11.1   3    58:  .1111.   4
11:  ..1..1   2    27:  .1.11.   3    43:  .11.1.   3    59:  1111..   4
12:  .1..1.   2    28:  1.11..   3    44:  11.1..   3    60:  1111.1   5
13:  1..1..   2    29:  1.11.1   4    45:  11.1.1   4    61:  .11111   5
14:  1..1.1   3    30:  .1.111   4    46:  .11.11   4    62:  11111.   5
15:  .1..11   3    31:  1.111.   4    47:  11.11.   4    63:  111111   6
16:  1..11.   3    32:  1.1111   5    48:  11.111   5
```

Figure 8.4-A: Nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession (shifts-order). All shifts are left shifts.

```
 1:  .....1   1    17:  ..1..1   2    33:  ...111   3    49:  .11.1.   3
 2:  ....1.   1    18:  ..1.11   3    34:  ..111.   3    50:  11.1..   3
 3:  ...1..   1    19:  .1.11.   3    35:  .111..   3    51:  11.1.1   4
 4:  ..1...   1    20:  1.11..   3    36:  111...   3    52:  11.111   5
 5:  .1....   1    21:  1.11.1   4    37:  111..1   4    53:  11.11.   4
 6:  1.....   1    22:  1.1111   5    38:  111.11   5    54:  .11.11   4
 7:  1....1   2    23:  1.111.   4    39:  111.1.   4    55:  .11..1   3
 8:  1...11   3    24:  .1.111   4    40:  .111.1   4    56:  11..1.   3
 9:  1...1.   2    25:  .1.1.1   3    41:  .11111   5    57:  11..11   4
10:  .1...1   2    26:  1.1.1.   3    42:  11111.   5    58:  11...1   3
11:  .1..11   3    27:  1.1.11   4    43:  111111   6    59:  11....   2
12:  1..11.   3    28:  1.1..1   3    44:  1111.1   5    60:  .11...   2
13:  1..111   4    29:  1.1...   2    45:  1111..   4    61:  ..11..   2
14:  1..1.1   3    30:  .1.1..   2    46:  .1111.   4    62:  ...11.   2
15:  1..1..   2    31:  ..1.1.   2    47:  ..1111   4    63:  ....11   2
16:  .1..1.   2    32:  ...1.1   2    48:  ..11.1   3
```

Figure 8.4-B: Nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession and transitions that are not shifts switch just one bit (minimal-change shifts-order).

```
 1:  .......1   1    17:  ..1...1.   2    33:  1..1.1..   3    49:  ..1.1.1.   3
 2:  ......1.   1    18:  .1...1..   2    34:  1..1.1.1   4    50:  .1.1.1..   3
 3:  .....1..   1    19:  1...1...   2    35:  ....1.1.   2    51:  1.1.1...   3
 4:  ....1...   1    20:  1...1..1   3    36:  ....1.1.   2    52:  1.1.1..1   4
 5:  ...1....   1    21:  .1...1.1   3    37:  ...1.1..   2    53:  .1.1.1.1   4
 6:  ..1.....   1    22:  1...1.1.   3    38:  ..1.1...   2    54:  1.1.1.1.   4
 7:  .1......   1    23:  ....1..1   2    39:  .1.1....   2
 8:  1.......   1    24:  ...1..1.   2    40:  1.1.....   2
 9:  1......1   2    25:  ..1..1..   2    41:  1.1....1   3
10:  .1.....1   2    26:  .1..1...   2    42:  .1.1...1   3
11:  1.....1.   2    27:  1..1....   2    43:  1.1...1.   3
12:  ..1....1   2    28:  1..1...1   3    44:  ..1.1..1   3
13:  .1....1.   2    29:  .1..1..1   3    45:  .1.1..1.   3
14:  1....1..   2    30:  1..1..1.   3    46:  1.1..1..   3
15:  1....1.1   3    31:  ..1..1.1   3    47:  1.1..1.1   4
16:  ...1...1   2    32:  .1..1.1.   3    48:  ...1.1.1   3
```

Figure 8.4-C: Nonzero Fibonacci words in an order where all shifts appear in succession.

8.4 Shifts-order for subsets

Figure 8.4-A shows an ordering (*shifts-order*) of the nonempty subsets of a 6-bit binary word where all linear shifts of a word appear in succession. The generation is done by a simple recursion [FXT: comb/shift-subsets-demo.cc]:

```
ulong n;  // number of bits
ulong N;  // 2**n

void A(ulong x)
{
    if ( x>=N )  return;
    visit(x);
    A(2*x);
    A(2*x+1);
}
```

The function `visit()` simply prints the binary expansion of its argument. The initial call is `A(1)`.

The transitions that are not shifts change just one bit if the following pair of functions is used for the recursion (*minimal-change shifts-order* shown in figure 8.4-B):

```
void F(ulong x)
{
    if ( x>=N )  return;
    visit(x);
    F(2*x);
    G(2*x+1);
}

void G(ulong x)
{
    if ( x>=N )  return;
    F(2*x+1);
    G(2*x);
```

```
    visit(x);
}
```

The initial call is `F(1)`, the reversed order can be generated via `G(1)`.

We note that a simple variation can be used to generate the Fibonacci words in a shifts-order shown in figure 8.4-C. With transitions that are not shifts more than one bit is changed in general. The function used is [FXT: comb/shift-subsets-demo.cc]:

```
void B(ulong x)
{
    if ( x>=N )  return;
    visit(x);
    B(2*x);
    B(4*x+1);
}
```

8.5 k-subsets where k lies in a given range

We give algorithms for generating all *k-subsets* (subsets of an n-element set with k elements in each subset) where $k_{min} \le k \le k_{max}$. If $k_{min} = 0$ and $k_{max} = n$ we obtain all subsets, if $k_{min} = k_{max} = k$ we obtain combinations $\binom{n}{k}$.

8.5.1 Recursive algorithm

A recursive routine that generates all k-subsets where lies in a prescribed range is [FXT: `class ksubset_rec` in comb/ksubset-rec.h]. The routine can generate the subsets in 16 different orders. Figure 8.5-A shows the lexicographic orders, figure 8.5-B shows three Gray codes. The order numbers correspond to the second argument of the program [FXT: comb/ksubset-rec-demo.cc]. The constructor has just one argument, the number of elements of the set whose subsets shall be generated:

```
class ksubset_rec
// k-susbsets where kmin<=k<=kmax in various orders.
// Recursive CAT algorithm.
{
public:
    long n_; // subsets of a n-element set
    long kmin_, kmax_;  //  k-subsets where kmin<=k<=kma
    long *rv_;  // record of visits in graph (list of elements in subset)
    ulong ct_;   // count subsets
    ulong rct_;  // count recursions (==work)
    ulong rq_;   // condition that determines the order
    ulong pq_;   // condition that determines the (printing) order
    ulong nq_;   // whether to reverse order
    // function to call with each combination:
    void (*visit_)(const ksubset_rec &, long);
public:
    ksubset_rec(ulong n)
    {
        n_ = n;
        rv_ = new long[n_+1];
        ++rv_;
        rv_[-1] = -1UL;
    }

    ~ksubset_rec()
    {
        --rv_;
        delete [] rv_;
    }
```

One has to supply the interval for k (variables `kmin` and `kmax`) and a function that will be called with each subset. The argument `rq` determines which of the sixteen different orderings is chosen, the order can be reversed with nonzero `nq`.

```
    void generate(void (*visit)(const ksubset_rec &, long),
                  long kmin, long kmax, ulong rq, ulong nq=0)
```

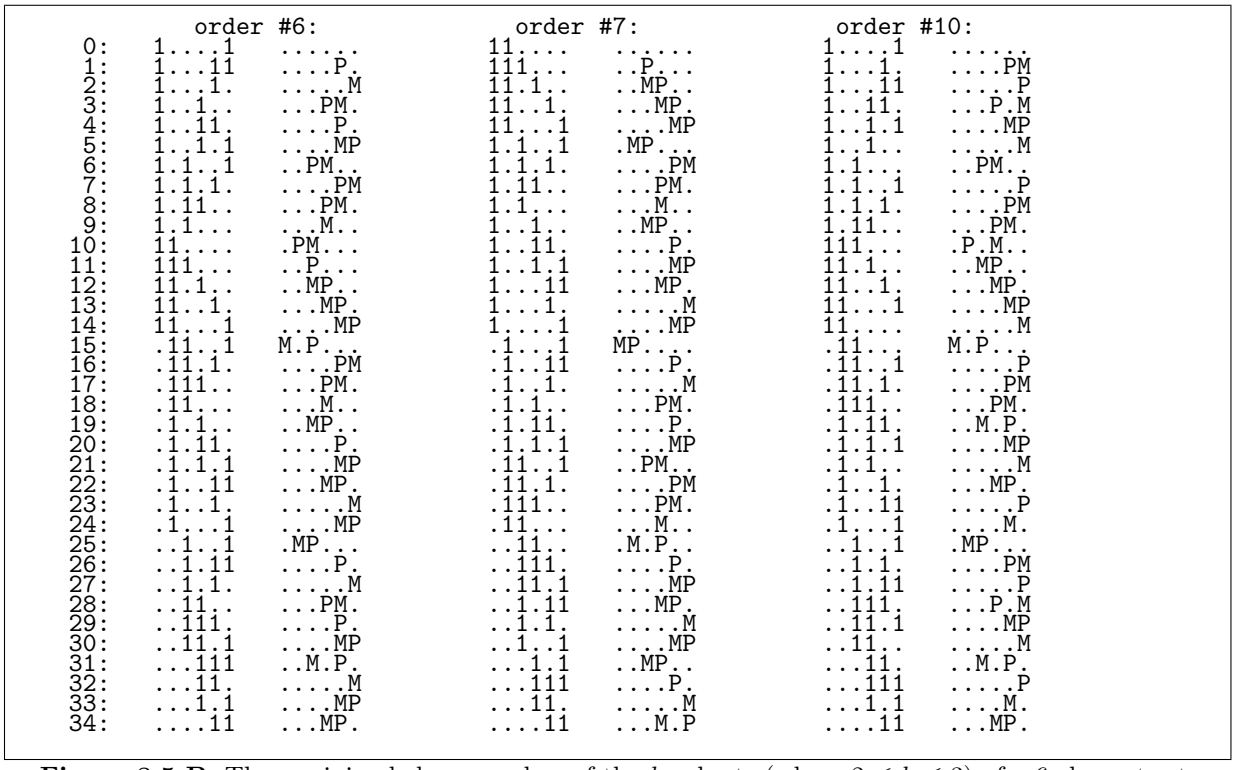
```
          order #0:                                    order #8:
     0:   11....   ......   { 0, 1 }          111...   ......   { 0, 1, 2 }
     1:   111...   ..P...   { 0, 1, 2 }       11.1..   ..MP..   { 0, 1, 3 }
     2:   11.1..   ..MP..   { 0, 1, 3 }       11..1.   ...MP.   { 0, 1, 4 }
     3:   11..1.   ...MP.   { 0, 1, 4 }       11...1   ....MP   { 0, 1, 5 }
     4:   11...1   ....MP   { 0, 1, 5 }       11....   .....M   { 0, 1 }
     5:   1.1...   .MP..M   { 0, 2 }          1.11..   .MPP..   { 0, 2, 3 }
     6:   1.11..   ...P..   { 0, 2, 3 }       1.1.1.   ...MP.   { 0, 2, 4 }
     7:   1.1.1.   ...MP.   { 0, 2, 4 }       1.1..1   ....MP   { 0, 2, 5 }
     8:   1.1..1   ....MP   { 0, 2, 5 }       1.1...   .....M   { 0, 2 }
     9:   1..1..   ..MP.M   { 0, 3 }          1..11.   ..MPP.   { 0, 3, 4 }
    10:   1..11.   ....P.   { 0, 3, 4 }       1..1.1   ....MP   { 0, 3, 5 }
    11:   1..1.1   ....MP   { 0, 3, 5 }       1..1..   .....M   { 0, 3 }
    12:   1...1.   ...MPM   { 0, 4 }          1...11   ...MPP   { 0, 4, 5 }
    13:   1...11   .....P   { 0, 4, 5 }       1...1.   .....M   { 0, 4 }
    14:   1....1   ....M.   { 0, 5 }          1....1   ....MP   { 0, 5 }
    15:   .11...   MPP..M   { 1, 2 }          .111..   MPPP.M   { 1, 2, 3 }
    16:   .111..   ...P..   { 1, 2, 3 }       .11.1.   ...MP.   { 1, 2, 4 }
    17:   .11.1.   ...MP.   { 1, 2, 4 }       .11..1   ....MP   { 1, 2, 5 }
    18:   .11..1   ....MP   { 1, 2, 5 }       .11...   .....M   { 1, 2 }
    19:   .1.1..   ..MP.M   { 1, 3 }          .1.11.   ..MPP.   { 1, 3, 4 }
    20:   .1.11.   ....P.   { 1, 3, 4 }       .1.1.1   ....MP   { 1, 3, 5 }
    21:   .1.1.1   ....MP   { 1, 3, 5 }       .1.1..   .....M   { 1, 3 }
    22:   .1..1.   ...MPM   { 1, 4 }          .1..11   ...MPP   { 1, 4, 5 }
    23:   .1..11   .....P   { 1, 4, 5 }       .1..1.   .....M   { 1, 4 }
    24:   .1...1   ....M.   { 1, 5 }          .1...1   ....MP   { 1, 5 }
    25:   ..11..   .MPP.M   { 2, 3 }          ..111.   .MPPPM   { 2, 3, 4 }
    26:   ..111.   ....P.   { 2, 3, 4 }       ..11.1   ....MP   { 2, 3, 5 }
    27:   ..11.1   ....MP   { 2, 3, 5 }       ..11..   .....M   { 2, 3 }
    28:   ..1.1.   ...MPM   { 2, 4 }          ..1.11   ...MPP   { 2, 4, 5 }
    29:   ..1.11   .....P   { 2, 4, 5 }       ..1.1.   .....M   { 2, 4 }
    30:   ..1..1   ....M.   { 2, 5 }          ..1..1   ....MP   { 2, 5 }
    31:   ...11.   ..MPPM   { 3, 4 }          ...111   ..MPP.   { 3, 4, 5 }
    32:   ...111   .....P   { 3, 4, 5 }       ...11.   .....M   { 3, 4 }
    33:   ...1.1   ....M.   { 3, 5 }          ...1.1   ....MP   { 3, 5 }
    34:   ....11   ...MP.   { 4, 5 }          ....11   ...MP.   { 4, 5 }
```

Figure 8.5-A: The k-subsets (where $2 \le k \le 3$) of a 6-element set. Lexicographic order for sets (left) and reversed lexicographic order for delta sets (right).

```
    {
        ct_ = 0;
        rct_ = 0;

        kmin_ = kmin;
        kmax_ = kmax;
        if ( kmin_ > kmax_ )  swap2(kmin_, kmax_);
        if ( kmax_ > n_ )  kmax_ = n_;
        if ( kmin_ > n_ )  kmin_ = n_;

        visit_ = visit;
        rq_ = rq % 4;
        pq_ = (rq>>2) % 4;
        nq_ = nq;
        next_rec(0);
    }
private:
    void next_rec(long d);
};
```

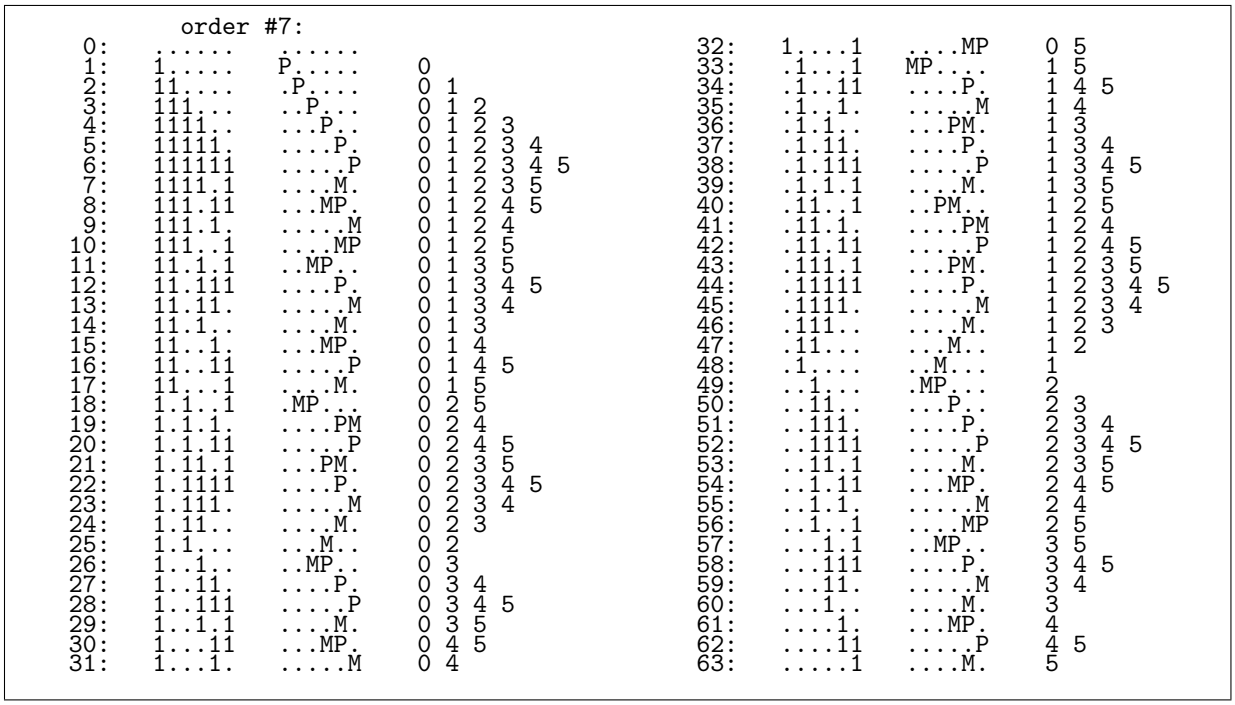
The recursive routine itself is given in [FXT: comb/ksubset-rec.cc]:

```
void
ksubset_rec::next_rec(long d)
{
    if ( d>kmax_ )  return;

    ++rct_;  // measure computational work
    long rv1 = rv_[d-1];  // left neighbor
    bool q;
    switch ( rq_ % 4 )
```



	order #6:		order #7:		order #10:	
0:	1....1	.....	11....	.....	1....1	.....
1:	1...11	....P.	111...	..P...	1...1.	....PM
2:	1...1.	....PM	11.1..	..MP..	1...11	....P.
3:	1..1..	...PM.	11..1.	...MP.	1..11.	...P.M
4:	1..11.	....P.	11...1	....MP	1..1.1	....MP
5:	1..1.1	....MP	1.1..1	.MP...	1..1..	.....M
6:	1.1..1	..PM..	1.1.1.	....PM	1.1...	..PM..
7:	1.1.1.	....PM	1.11..	...PM.	1.1..1	.....P
8:	1.11..	...PM.	1.1...	...M..	1.1.1.	....PM
9:	1.1...	...M..	1..1..	..MP..	1.11..	...PM.
10:	11....	.PM...	1..11.	....P.	111...	.P.M..
11:	111...	..P...	1..1.1	....MP	11.1..	..MP..
12:	11.1..	..MP..	1...11	...MP.	11..1.	...MP.
13:	11..1.	...MP.	1...1.	.....M	11...1	....MP
14:	11...1	....MP	1....1	....MP	11....	.....M
15:	.11..1	M.P...	.1...1	MP....	.11...	M.P...
16:	.11.1.	....PM	.1..11	....P.	.11..1	.....P
17:	.111..	...PM.	.1..1.	.....M	.11.1.	....PM
18:	.11...	...M..	.1.1..	...PM.	.111..	...PM.
19:	.1.1..	..MP..	.1.11.	....P.	.1.11.	..M.P.
20:	.1.11.	....P.	.1.1.1	....MP	.1.1.1	....MP
21:	.1.1.1	....MP	.11..1	..PM..	.1.1..	.....M
22:	.1..11	...MP.	.11.1.	....PM	.1..1.	...MP.
23:	.1..1.	.....M	.111..	...PM.	.1..11	.....P
24:	.1...1	....MP	.11...	...M..	.1...1	....M.
25:	..1..1	.MP...	..11..	.M.P..	..1..1	.MP...
26:	..1.11	....P.	..111.	....P.	..1.1.	....PM
27:	..1.1.	.....M	..11.1	....MP	..1.11	.....P
28:	..11..	...PM.	..1.11	...MP.	..111.	...P.M
29:	..111.	....P.	..1.1.	.....M	..11.1	....MP
30:	..11.1	....MP	..1..1	....MP	..11..	.....M
31:	...111	..M.P.	...1.1	..MP..	...11.	..M.P.
32:	...11.	.....M	...111	....P.	...111	.....P
33:	...1.1	....MP	...11.	.....M	...1.1	....M.
34:	11	...MP.	11	...M.P	11	...MP.

Figure 8.5-B: Three minimal-change orders of the k-subsets (where $2 \leq k \leq 3$) of a 6-element set.



	order #7:								
0:	.....	.....			32:	1....1	....MP	0 5	
1:	1.....	P.....	0		33:	.1...1	MP....	1 5	
2:	11....	.P....	0 1		34:	.1..11	....P.	1 4 5	
3:	111...	..P...	0 1 2		35:	.1..1.	.....M	1 4	
4:	1111..	...P..	0 1 2 3		36:	.1.1..	...PM.	1 3	
5:	11111.	....P.	0 1 2 3 4		37:	.1.11.	....P.	1 3 4	
6:	111111	.....P	0 1 2 3 4 5		38:	.1.111	.....P	1 3 4 5	
7:	1111.1	....M.	0 1 2 3 5		39:	.1.1.1	....M.	1 3 5	
8:	111.11	...MP.	0 1 2 4 5		40:	.11..1	..PM..	1 2 5	
9:	111.1.	.....M	0 1 2 4		41:	.11.1.	....PM	1 2 4	
10:	111..1	....MP	0 1 2 5		42:	.11.11	.....P	1 2 4 5	
11:	11.1.1	..MP..	0 1 3 5		43:	.111.1	...PM.	1 2 3 5	
12:	11.111	....P.	0 1 3 4 5		44:	.11111	....P.	1 2 3 4 5	
13:	11.11.	.....M	0 1 3 4		45:	.1111.	.....M	1 2 3 4	
14:	11.1..	....M.	0 1 3		46:	.111..	....M.	1 2 3	
15:	11..1.	...MP.	0 1 4		47:	.11...	...M..	1 2	
16:	11..11	.....P	0 1 4 5		48:	.1....	..M...	1	
17:	11...1	....M.	0 1 5		49:	..1...	.MP...	2	
18:	1.1..1	.MP...	0 2 5		50:	..11..	...P..	2 3	
19:	1.1.1.	....PM	0 2 4		51:	..111.	....P.	2 3 4	
20:	1.1.11	.....P	0 2 4 5		52:	..1111	.....P	2 3 4 5	
21:	1.11.1	...PM.	0 2 3 5		53:	..11.1	....M.	2 3 5	
22:	1.1111	....P.	0 2 3 4 5		54:	..1.11	...MP.	2 4 5	
23:	1.111.	.....M	0 2 3 4		55:	..1.1.	.....M	2 4	
24:	1.11..	....M.	0 2 3		56:	..1..1	....MP	2 5	
25:	1.1...	...M..	0 2		57:	...1.1	..MP..	3 5	
26:	1..1..	..MP..	0 3		58:	...111	....P.	3 4 5	
27:	1..11.	....P.	0 3 4		59:	...11.	.....M	3 4	
28:	1..111	.....P	0 3 4 5		60:	...1..	....M.	3	
29:	1..1.1	....M.	0 3 5		61:	....1.	...MP.	4	
30:	1...11	...MP.	0 4 5		62:	11	.....P	4 5	
31:	1...1.	.....M	0 4		63:	1	....M.	5	

Figure 8.5-C: With $k_{min} = 0$ and order number seven at each transition either one element is added or removed, or one element moves to an adjacent position.

```
    {
    case 0:  q = 1;  break;
    case 1:  q = !(d&1);  break;
    case 2:  q = rv1&1;  break;
    case 3:  q = (d^rv1)&1;  break;
    }
    if ( nq_ )  q = !q;
    long x0 = rv1 + 1;
    long rx = n_ - (kmin_ - d);
    long x1 = min2( n_-1, rx );
#define PCOND(x) if ( (pq_==x) && (d>=kmin_) )  { visit_(*this, d);  ++ct_; }
    PCOND(0);
    if ( q )  // forward:
    {
        PCOND(1);
        for (long x=x0; x<=x1; ++x)  { rv_[d] = x;  next_rec(d+1); }
        PCOND(2);
    }
    else  // backward:
    {
        PCOND(2);
        for (long x=x1; x>=x0; --x)  { rv_[d] = x;  next_rec(d+1); }
        PCOND(1);
    }
    PCOND(3);
#undef PCOND
}
```

About 50 million subsets per second can be generated.

8.5.2 Iterative algorithm for a minimal-change order

```
             delta set  diff      set
         1:    ...11    .....    { 4, 5 }
         2:    ..11.    ..P.M    { 3, 4 }
         3:    ..111    ....P    { 3, 4, 5 }
         4:    ..1.1    ...M.    { 3, 5 }
         5:    .11..    .P..M    { 2, 3 }
         6:    .11.1    ....P    { 2, 3, 5 }
         7:    .1111    ...P.    { 2, 3, 4, 5 }
         8:    .111.    ....M    { 2, 3, 4 }
         9:    .1.1.    ..M..    { 2, 4 }
        10:    .1.11    ....P    { 2, 4, 5 }
        11:    .1..1    ...M.    { 2, 5 }
        12:    11...    P...M    { 1, 2 }
        13:    11..1    ....P    { 1, 2, 5 }
        14:    11.11    ...P.    { 1, 2, 4, 5 }
        15:    11.1.    ....M    { 1, 2, 4 }
        16:    1111.    ..P..    { 1, 2, 3, 4 }
        17:    111.1    ...MP    { 1, 2, 3, 5 }
        18:    111..    ....M    { 1, 2, 3 }
        19:    1.1..    .M...    { 1, 3 }
        20:    1.1.1    ....P    { 1, 3, 5 }
        21:    1.111    ...P.    { 1, 3, 4, 5 }
        22:    1.11.    ....M    { 1, 3, 4 }
        23:    1..1.    ..M..    { 1, 4 }
        24:    1..11    ....P    { 1, 4, 5 }
        25:    1...1    ...M.    { 1, 5 }
```

Figure 8.5-D: The (25) k-subsets where $2 \le k \le 4$ of a five-element set in a minimal-change order.

The class [FXT: `class ksubset_gray` in comb/ksubset-gray.h] allows the generation of k-subsets of a set where k lies in a prescribed range:

```
class ksubset_gray
{
public:
```

```
    ulong n_;   // k-subsets of {1, 2, ..., n}
    ulong kmin_, kmax_;  // kmin <= k <= kmax
    ulong k_;   // k elements in current set
    ulong *S_;  // set in S[1,2,...,k] with elements \in {1,2,...,n}
    ulong j_;   // aux
public:
    ksubset_gray(ulong n, ulong kmin, ulong kmax)
    {
        n_ = (n>0 ? n : 1);
        // Must have 1<=kmin<=kmax<=n
        kmin_ = kmin;
        kmax_ = kmax;
        if ( kmax_ < kmin_ )  swap2(kmin_, kmax_);
        if ( kmin_==0 )  kmin_ = 1;

        S_ = new ulong[kmax_+1];
        S_[0] = 0;  // sentinel: != 1
        first();
    }

    ~ksubset_gray()  { delete [] S_; }
    const ulong *data()  const  { return S_+1; }
    const ulong num()  const  { return k_; }

    ulong last()
    {
        S_[1] = 1;  k_ = kmin_;
        if ( kmin_==1 )  { j_ = 1; }
        else
        {
            for (ulong i=2; i<=kmin_; ++i)  { S_[i] = n_ - kmin_ + i; }
            j_ = 2;
        }
        return k_;
    }

    ulong first()
    {
        k_ = kmin_;
        for (ulong i=1; i<=kmin_; ++i)  { S_[i] = n_ - kmin_ + i; }
        j_ = 1;
        return k_;
    }

    bool is_first()  const  { return ( S_[1] == n_ - kmin_ + 1 );  }

    bool is_last()  const
    {
        if  ( S_[1] != 1  )   return 0;
        if ( kmin_<=1 )  return (k_==1);
        return  (S_[2]==n_-kmin_+2);
    }
  [--snip--]
```

The routines for computing the next or previous subset are adapted from a routine to compute the successor given in [141]. It is split in two auxiliary functions:

```
private:
    void prev_even()
    {
        ulong &n=n_, &kmin=kmin_, &kmax=kmax_, &j=j_;
        if ( S_[j-1] == S_[j]-1 )  // can touch sentinel
        {
            S_[j-1] = S_[j];
            if ( j > kmin )
            {
                if ( S_[kmin] == n )  { j = j-2; }  else  { j = j-1; }
            }
            else
            {
                S_[j] = n - kmin + j;
                if ( S_[j-1]==S_[j]-1 )  { j = j-2; }
            }
        }
        else
        {
            S_[j] = S_[j] - 1;
```

```
            if ( j < kmax )
            {
                S_[j+1] = S_[j] + 1;
                if ( j >= kmin-1 )  { j = j+1; }  else  { j = j+2; }
            }
        }
    }

    void prev_odd()
    {
        ulong &n=n_, &kmin=kmin_, &kmax=kmax_, &j=j_;
        if ( S_[j] == n )  { j = j-1; }
        else
        {
            if ( j < kmax )
            {
                S_[j+1] = n;
                j = j+1;
            }
            else
            {
                S_[j] = S_[j]+1;
                if ( S_[kmin]==n )  { j = j-1; }
            }
        }
    }
  [--snip--]
```

The `next()` and `prev()` functions use these routines, note that calls cannot not be mixed.

```
    ulong prev()
    {
        if ( is_first() )  { last(); return 0; }

        if ( j_&1 )  prev_odd();
        else         prev_even();

        if ( j_<kmin_ )  { k_ = kmin_; }  else  { k_ = j_; };

        return k_;
    }

    ulong next()
    {
        if ( is_last() )  { first();  return 0; }

        if ( j_&1 )  prev_even();
        else         prev_odd();

        if ( j_<kmin_ )  { k_ = kmin_; }  else  { k_ = j_; };

        return k_;
    }
  [--snip--]
```

Usage of the class is shown in the program [FXT: comb/ksubset-gray-demo.cc], the k-subsets where $2 \le k \le 4$ in the order generated by the algorithm are shown in figure 8.5-D. About 80 million subsets per second can be generated with the routine `next()`, and 85 million with `prev()`.

8.5.3 A two-close order with homogenous moves

Orderings of the k-subsets with k in a given range that are *two-close* are shown in figure 8.5-E: one element is inserted or deleted or moves by at most two positions. The moves by two positions always cross a zero, the changes are *homogenous*. The list was produced with the program [FXT: comb/ksubset-twoclose-demo.cc] which uses [FXT: `class ksubset_twoclose` in comb/ksubset-twoclose.h]:

```
class ksubset_twoclose
// k-subsets (kmin<=k<=kmax) in a two-close order.
// Recursive algorithm.
{
public:
    ulong *rv_;  // record of visits in graph (delta set)
    ulong n_;    // subsets of the n-element set

    // function to call with each combination:
    void (*visit_)(const ksubset_twoclose &);
```

```
         delta set  diff      set                          delta set  diff      set
    1:     .1111    .....    { 1, 2, 3, 4 }        1:     ....11    ......    { 4, 5 }
    2:     ..111    .M...    { 2, 3, 4 }           2:     ...1.1    ...PM.    { 3, 5 }
    3:     1.111    P....    { 0, 2, 3, 4 }        3:     .1...1    .P.M..    { 1, 5 }
    4:     11.11    .PM..    { 0, 1, 3, 4 }        4:     .....1    .M....    { 5 }
    5:     .1.11    M....    { 1, 3, 4 }           5:     1....1    P.....    { 0, 5 }
    6:     ...11    .M...    { 3, 4 }              6:     ..1..1    M.P...    { 2, 5 }
    7:     1..11    P....    { 0, 3, 4 }           7:     ..11..    ...P.M    { 2, 3 }
    8:     11..1    .P.M.    { 0, 1, 4 }           8:     .1.1..    .PM...    { 1, 3 }
    9:     .1..1    M....    { 1, 4 }              9:     ...1..    .M....    { 3 }
   10:     1...1    PM...    { 0, 4 }             10:     1..1..    P.....    { 0, 3 }
   11:     ..1.1    M.P..    { 2, 4 }             11:     11....    .P.M..    { 0, 1 }
   12:     1.1.1    P....    { 0, 2, 4 }          12:     .1....    M.....    { 1 }
   13:     .11.1    MP...    { 1, 2, 4 }          13:     1.....    PM....    { 0 }
   14:     111.1    P....    { 0, 1, 2, 4 }       14:     ..1...    M.P...    { 2 }
   15:     1111.    ...PM    { 0, 1, 2, 3 }       15:     1.1...    P.....    { 0, 2 }
   16:     .111.    M....    { 1, 2, 3 }          16:     .11...    MP....    { 1, 2 }
   17:     ..11.    .M...    { 2, 3 }             17:     .1..1.    ..M.P.    { 1, 4 }
   18:     1.11.    P....    { 0, 2, 3 }          18:     ....1.    .M....    { 4 }
   19:     11.1.    .PM..    { 0, 1, 3 }          19:     1...1.    P.....    { 0, 4 }
   20:     .1.1.    M....    { 1, 3 }             20:     ..1.1.    M.P...    { 2, 4 }
   21:     1..1.    PM...    { 0, 3 }             21:     ...11.    ..MP..    { 3, 4 }
   22:     11...    .P.M.    { 0, 1 }
   23:     1.1..    .MP..    { 0, 2 }
   24:     .11..    MP...    { 1, 2 }
   25:     111..    P....    { 0, 1, 2 }
```

Figure 8.5-E: The k-subsets where $2 \leq k \leq 4$ of 5 elements (left) and the sets where $1 \leq k \leq 2$ of 6 elements (right) in two-close orders.

```
  [--snip--]
    void generate(void (*visit)(const ksubset_twoclose &),
                  ulong kmin, ulong kmax)
    {
        visit_ = visit;
        ulong kmax0 = n_ - kmin;
        next_rec(n_, kmax, kmax0, 0);
    }
```

The recursion is:

```
private:
    void next_rec(ulong d, ulong n1, ulong n0, bool q)
    // d:  remaining depth in recursion
    // n1: remaining ones to fill in
    // n0: remaining zeros to fill in
    // q:  direction in recursion
    {
        if ( 0==d )  { visit_(*this);  return; }
        --d;
        if ( q )
        {
            if ( n0 )  { rv_[d]=0;  next_rec(d, n1-0, n0-1, d&1); }
            if ( n1 )  { rv_[d]=1;  next_rec(d, n1-1, n0-0, q); }
        }
        else
        {
            if ( n1 )  { rv_[d]=1;  next_rec(d, n1-1, n0-0, q); }
            if ( n0 )  { rv_[d]=0;  next_rec(d, n1-0, n0-1, d&1); }
        }
    }
};
```

About 50 million subsets per second can be generated. For $k_{min} = k_{max} =: k$ we obtain the enup order for combinations described in section 6.5.2 on page 176.

Chapter 9

Mixed radix numbers

The *mixed radix* representation $A = [a_0, a_1, a_2, \ldots, a_{n-1}]$ of a number x with respect to a radix vector $M = [m_0, m_1, m_2, \ldots, m_{n-1}]$ is given by the unique expression

$$x = \sum_{k=0}^{n-1} a_k \prod_{j=0}^{k-1} m_j \tag{9.0-1}$$

where $0 \le a_j < m_j$ (and $0 < x < \prod_{j=0}^{n-1} m_j$, so that n digits suffice). For $M = [r, r, r, \ldots, r]$ the relation reduces to the radix-r representation:

$$x = \sum_{k=0}^{n-1} a_k \, r^k \tag{9.0-2}$$

All 3-digit radix-4 numbers are shown in various orders in figure 9.0-A. Note that the least significant digit (a_0) is at the left side of each number (array representation).

9.1 Counting order

An implementation for mixed radix counting is [FXT: `class mixedradix_lex` in comb/mixedradix-lex.h]:

```
class mixedradix_lex
{
public:
    ulong *a_;  // digits
    ulong *m1_; // radix (minus one) for each digit
    ulong n_;   // Number of digits
    ulong j_;   // position of last change
public:
    mixedradix_lex(const ulong *m, ulong n, ulong mm=0)
    {
        n_ = n;
        a_ = new ulong[n_+1];
        m1_ = new ulong[n_+1];
        a_[n_] = 1;   // sentinel: !=0, and !=m1[n]
        m1_[n_] = 0;  // sentinel
        mixedradix_init(n_, mm, m, m1_);
        first();
    }
  [--snip--]
```

The initialization routine is given in [FXT: comb/mixedradix-init.cc]:

	counting	Gray	modular Gray	gslex	endo	endo Gray
0:	[. . .]	[. . .]	[. . .]	[1 . .]	[. . .]	[. . .]
1:	[1 . .]	[1 . .]	[1 . .]	[2 . .]	[1 . .]	[1 . .]
2:	[2 . .]	[2 . .]	[2 . .]	[3 . .]	[3 . .]	[3 . .]
3:	[3 . .]	[3 . .]	[3 . .]	[1 1 .]	[2 . .]	[2 . .]
4:	[. 1 .]	[3 1 .]	[3 1 .]	[2 1 .]	[. 1 .]	[2 1 .]
5:	[1 1 .]	[2 1 .]	[. 1 .]	[3 1 .]	[1 1 .]	[3 1 .]
6:	[2 1 .]	[1 1 .]	[1 1 .]	[. 1 .]	[3 1 .]	[1 1 .]
7:	[3 1 .]	[. 1 .]	[2 1 .]	[1 2 .]	[2 1 .]	[. 1 .]
8:	[. 2 .]	[. 2 .]	[2 2 .]	[2 2 .]	[. 3 .]	[. 3 .]
9:	[1 2 .]	[1 2 .]	[3 2 .]	[3 2 .]	[1 3 .]	[1 3 .]
10:	[2 2 .]	[2 2 .]	[. 2 .]	[. 2 .]	[3 3 .]	[3 3 .]
11:	[3 2 .]	[3 2 .]	[1 2 .]	[1 3 .]	[2 3 .]	[2 3 .]
12:	[. 3 .]	[3 3 .]	[1 3 .]	[2 3 .]	[. 2 .]	[2 2 .]
13:	[1 3 .]	[2 3 .]	[2 3 .]	[3 3 .]	[1 2 .]	[3 2 .]
14:	[2 3 .]	[1 3 .]	[3 3 .]	[. 3 .]	[3 2 .]	[1 2 .]
15:	[3 3 .]	[. 3 .]	[. 3 .]	[1 . 1]	[2 2 .]	[. 2 .]
16:	[. . 1]	[. 3 1]	[. 3 1]	[2 . 1]	[. . 1]	[. 2 1]
17:	[1 . 1]	[1 3 1]	[1 3 1]	[3 . 1]	[1 . 1]	[1 2 1]
18:	[2 . 1]	[2 3 1]	[2 3 1]	[1 1 1]	[3 . 1]	[3 2 1]
19:	[3 . 1]	[3 3 1]	[3 3 1]	[2 1 1]	[2 . 1]	[2 2 1]
20:	[. 1 1]	[3 2 1]	[3 . 1]	[3 1 1]	[. 1 1]	[2 3 1]
21:	[1 1 1]	[2 2 1]	[. . 1]	[. 1 1]	[1 1 1]	[3 3 1]
22:	[2 1 1]	[1 2 1]	[1 . 1]	[1 2 1]	[3 1 1]	[1 3 1]
23:	[3 1 1]	[. 2 1]	[2 . 1]	[2 2 1]	[2 1 1]	[. 3 1]
24:	[. 2 1]	[. 1 1]	[2 1 1]	[3 2 1]	[. 3 1]	[. 1 1]
25:	[1 2 1]	[1 1 1]	[3 1 1]	[. 2 1]	[1 3 1]	[1 1 1]
26:	[2 2 1]	[2 1 1]	[. 1 1]	[1 3 1]	[3 3 1]	[3 1 1]
27:	[3 2 1]	[3 1 1]	[1 1 1]	[2 3 1]	[2 3 1]	[2 1 1]
28:	[. 3 1]	[3 . 1]	[1 2 1]	[3 3 1]	[. 2 1]	[2 . 1]
29:	[1 3 1]	[2 . 1]	[2 2 1]	[. 3 1]	[1 2 1]	[3 . 1]
30:	[2 3 1]	[1 . 1]	[3 2 1]	[. . 1]	[3 2 1]	[1 . 1]
31:	[3 3 1]	[. . 1]	[. 2 1]	[1 . 2]	[2 2 1]	[. . 1]
32:	[. . 2]	[. . 2]	[. 2 2]	[2 . 2]	[. . 3]	[. . 3]
33:	[1 . 2]	[1 . 2]	[1 2 2]	[3 . 2]	[1 . 3]	[1 . 3]
34:	[2 . 2]	[2 . 2]	[2 2 2]	[1 1 2]	[3 . 3]	[3 . 3]
35:	[3 . 2]	[3 . 2]	[3 2 2]	[2 1 2]	[2 . 3]	[2 . 3]
36:	[. 1 2]	[3 1 2]	[3 3 2]	[3 1 2]	[. 1 3]	[2 1 3]
37:	[1 1 2]	[2 1 2]	[. 3 2]	[. 1 2]	[1 1 3]	[3 1 3]
38:	[2 1 2]	[1 1 2]	[1 3 2]	[1 2 2]	[3 1 3]	[1 1 3]
39:	[3 1 2]	[. 1 2]	[2 3 2]	[2 2 2]	[2 1 3]	[. 1 3]
40:	[. 2 2]	[. 2 2]	[2 . 2]	[3 2 2]	[. 3 3]	[. 3 3]
41:	[1 2 2]	[1 2 2]	[3 . 2]	[. 2 2]	[1 3 3]	[1 3 3]
42:	[2 2 2]	[2 2 2]	[. . 2]	[1 3 2]	[3 3 3]	[3 3 3]
43:	[3 2 2]	[3 2 2]	[1 . 2]	[2 3 2]	[2 3 3]	[2 3 3]
44:	[. 3 2]	[3 3 2]	[1 1 2]	[3 3 2]	[. 2 3]	[2 2 3]
45:	[1 3 2]	[2 3 2]	[2 1 2]	[. 3 2]	[1 2 3]	[3 2 3]
46:	[2 3 2]	[1 3 2]	[3 1 2]	[. . 2]	[3 2 3]	[1 2 3]
47:	[3 3 2]	[. 3 2]	[. 1 2]	[1 . 3]	[2 2 3]	[. 2 3]
48:	[. . 3]	[. 3 3]	[. 1 3]	[2 . 3]	[. . 2]	[. 2 2]
49:	[1 . 3]	[1 3 3]	[1 1 3]	[3 . 3]	[1 . 2]	[1 2 2]
50:	[2 . 3]	[2 3 3]	[2 1 3]	[1 1 3]	[3 . 2]	[3 2 2]
51:	[3 . 3]	[3 3 3]	[3 1 3]	[2 1 3]	[2 . 2]	[2 2 2]
52:	[. 1 3]	[3 2 3]	[3 2 3]	[3 1 3]	[. 1 2]	[2 3 2]
53:	[1 1 3]	[2 2 3]	[. 2 3]	[. 1 3]	[1 1 2]	[3 3 2]
54:	[2 1 3]	[1 2 3]	[1 2 3]	[1 2 3]	[3 1 2]	[1 3 2]
55:	[3 1 3]	[. 2 3]	[2 2 3]	[2 2 3]	[2 1 2]	[. 3 2]
56:	[. 2 3]	[. 1 3]	[2 3 3]	[3 2 3]	[. 3 2]	[. 1 2]
57:	[1 2 3]	[1 1 3]	[3 3 3]	[. 2 3]	[1 3 2]	[1 1 2]
58:	[2 2 3]	[2 1 3]	[. 3 3]	[1 3 3]	[3 3 2]	[3 1 2]
59:	[3 2 3]	[3 1 3]	[1 3 3]	[2 3 3]	[2 3 2]	[2 1 2]
60:	[. 3 3]	[3 . 3]	[1 . 3]	[3 3 3]	[. 2 2]	[2 . 2]
61:	[1 3 3]	[2 . 3]	[2 . 3]	[. 3 3]	[1 2 2]	[3 . 2]
62:	[2 3 3]	[1 . 3]	[3 . 3]	[. . 3]	[3 2 2]	[1 . 2]
63:	[3 3 3]	[. . 3]	[. . 3]	[. . .]	[2 2 2]	[. . 2]

Figure 9.0-A: All 3-digit, radix-4 numbers in various orders (dots denote zeros): counting-, Gray-, modular Gray-, gslex-, endo-, and endo Gray order. The least significant digit is on the left of each word (array notation).

```
                M=[ 2 3 4 ]       M=[ 4 3 2 ]
           0:      [ . . . ]         [ . . . ]
           1:      [ 1 . . ]         [ 1 . . ]
           2:      [ . 1 . ]         [ 2 . . ]
           3:      [ 1 1 . ]         [ 3 . . ]
           4:      [ . 2 . ]         [ . 1 . ]
           5:      [ 1 2 . ]         [ 1 1 . ]
           6:      [ . . 1 ]         [ 2 1 . ]
           7:      [ 1 . 1 ]         [ 3 1 . ]
           8:      [ . 1 1 ]         [ . 2 . ]
           9:      [ 1 1 1 ]         [ 1 2 . ]
          10:      [ . 2 1 ]         [ 2 2 . ]
          11:      [ 1 2 1 ]         [ 3 2 . ]
          12:      [ . . 2 ]         [ . . 1 ]
          13:      [ 1 . 2 ]         [ 1 . 1 ]
          14:      [ . 1 2 ]         [ 2 . 1 ]
          15:      [ 1 1 2 ]         [ 3 . 1 ]
          16:      [ . 2 2 ]         [ . 1 1 ]
          17:      [ 1 2 2 ]         [ 1 1 1 ]
          18:      [ . . 3 ]         [ 2 1 1 ]
          19:      [ 1 . 3 ]         [ 3 1 1 ]
          20:      [ . 1 3 ]         [ . 2 1 ]
          21:      [ 1 1 3 ]         [ 1 2 1 ]
          22:      [ . 2 3 ]         [ 2 2 1 ]
          23:      [ 1 2 3 ]         [ 3 2 1 ]
```

Figure 9.1-A: Mixed radix numbers in counting order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (rising factorial basis, left) and $M = [4, 3, 2]$ (falling factorial basis, right). The least significant digit is on the left of each word (array notation).

```
void
mixedradix_init(ulong n, ulong mm, const ulong *m, ulong *m1)
// Auxiliary function used to initialze vector of nines in mixed radix classes.
{
    if ( m )  // all radices given
    {
        for (ulong k=0; k<n; ++k)  m1[k] = m[k] - 1;
    }
    else
    {
        if ( mm>1 )  // use mm as radix for all digits:
            for (ulong k=0; k<n; ++k)  m1[k] = mm - 1;
        else
        {
            if ( mm==0 )  // falling factorial basis
                for (ulong k=0; k<n; ++k)  m1[k] = n - k;
            else // rising factorial basis
                for (ulong k=0; k<n; ++k)  m1[k] = k + 1;
        }
    }
}
```

Instead of the vector $M = [m_0, m_1, m_2, \ldots, m_{n-1}]$ the class uses the vector of 'nines', that is $M' = [m_0 - 1, m_1 - 1, m_2 - 1, \ldots, m_{n-1} - 1]$ (variable `m1_`). This modification leads to slightly faster generation. The first n-digit number is all-zero, the last is all-nines:

```
    void first()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = 0;
        j_ = n_;
    }

    void last()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = m1_[k];
        j_ = n_;
    }
  [--snip--]
```

A number is incremented by setting all nines (digits a_j that are equal to $m_j - 1$) at the lower end to zero and incrementing the next digit:

```
    bool next()  // increment
    {
        ulong j = 0;
        while ( a_[j]==m1_[j] )  { a_[j]=0;  ++j; }  // can touch sentinels
        j_ = j;
        if ( j==n_ )  return false;  // current is last
        ++a_[j];
        return true;
    }
  [--snip--]
```

A number is decremented by setting all zero digits at the lower end to nine and decrementing the next digit:

```
    bool prev()  // decrement
    {
        ulong j = 0;
        while ( a_[j]==0 )  { a_[j]=m1_[j];  ++j; }  // can touch sentinels
        j_ = j;
        if ( j==n_ )  return false;  // current is first
        --a_[j];
        return true;
    }
  [--snip--]
```

Figure 9.1-A shows the 3-digit mixed radix numbers for basis vector $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). The listings where created with the program [FXT: comb/mixedradix-lex-demo.cc].

The routine `next()` generates between about 140 million (radix-2 numbers, $M = [2, 2, 2, \ldots, 2]$), 210 million (radix-3), and about 300 million (radix-8) numbers per second. Note that radix-2 leads to the slowest generation as the average carries are long compared to higher radices. The number of carries with incrementing is on average:

$$C \;=\; \frac{1}{m_0}\left(1+\frac{1}{m_1}\left(1+\frac{1}{m_2}\,(\ldots)\right)\right) \;=\; \sum_{k=0}^{n}\frac{1}{\prod_{j=0}^{k} m_j} \tag{9.1-1}$$

The number of digits changed on average equals $C + 1$. For $M = [r, r, r, \ldots, r]$ (and $n = \infty$) we obtain $C = \frac{1}{r-1}$. For the worst case ($r = 2$) we have $C = 1$, so two digits are changed on average.

9.2 Gray code order

Figure 9.2-A shows the 3-digit mixed radix numbers for radix vectors $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right) in Gray code order. An constant amortized time (CAT) implementation for mixed radix numbers in a Gray code order is [FXT: `class mixedradix_gray` in comb/mixedradix-gray.h]:

```
class mixedradix_gray
{
public:
    ulong *a_;  // mixed radix digits
    ulong *m1_; // radices (minus one)
    ulong *i_;  // direction
    ulong n_;   // n_ digits
    ulong j_;   // position of last change
    int dm_;    // direction of last move
public:
    mixedradix_gray(const ulong *m, ulong n, ulong mm=0)
    {
        n_ = n;
        a_ = new ulong[n_+1];
        a_[n] = -1UL;  // sentinel
        i_ = new ulong[n_+1];
        i_[n_] = 0;    // sentinel
        m1_ = new ulong[n_+1];
        mixedradix_init(n_, mm, m, m1_);
```

```
         M=[ 2 3 4 ]     x     j  d          M=[ 4 3 2 ]     x     j  d
  0:       [ . . . ]     0                     [ . . . ]     0
  1:       [ 1 . . ]     1     0  1           [ 1 . . ]     1     0  1
  2:       [ 1 1 . ]     3     1  1           [ 2 . . ]     2     0  1
  3:       [ . 1 . ]     2     0 -1           [ 3 . . ]     3     0  1
  4:       [ . 2 . ]     4     1  1           [ 3 1 . ]     7     1  1
  5:       [ 1 2 . ]     5     0  1           [ 2 1 . ]     6     0 -1
  6:       [ 1 2 1 ]    11     2  1           [ 1 1 . ]     5     0 -1
  7:       [ . 2 1 ]    10     0 -1           [ . 1 . ]     4     0 -1
  8:       [ . 1 1 ]     8     1 -1           [ . 2 . ]     8     1  1
  9:       [ 1 1 1 ]     9     0  1           [ 1 2 . ]     9     0  1
 10:       [ 1 . 1 ]     7     1 -1           [ 2 2 . ]    10     0  1
 11:       [ . . 1 ]     6     0 -1           [ 3 2 . ]    11     0  1
 12:       [ . . 2 ]    12     2  1           [ 3 2 1 ]    23     2  1
 13:       [ 1 . 2 ]    13     0  1           [ 2 2 1 ]    22     0 -1
 14:       [ 1 1 2 ]    15     1  1           [ 1 2 1 ]    21     0 -1
 15:       [ . 1 2 ]    14     0 -1           [ . 2 1 ]    20     0 -1
 16:       [ . 2 2 ]    16     1  1           [ . 1 1 ]    16     1 -1
 17:       [ 1 2 2 ]    17     0  1           [ 1 1 1 ]    17     0  1
 18:       [ 1 2 3 ]    23     2  1           [ 2 1 1 ]    18     0  1
 19:       [ . 2 3 ]    22     0 -1           [ 3 1 1 ]    19     0  1
 20:       [ . 1 3 ]    20     1 -1           [ 3 . 1 ]    15     1 -1
 21:       [ 1 1 3 ]    21     0  1           [ 2 . 1 ]    14     0 -1
 22:       [ 1 . 3 ]    19     1 -1           [ 1 . 1 ]    13     0 -1
 23:       [ . . 3 ]    18     0 -1           [ . . 1 ]    12     0 -1
```

Figure 9.2-A: Mixed radix numbers in Gray code order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Columns '`x`' give the values, columns '`j`' and '`d`' give the position of last change and its direction, respectively.

```
        first();
    }
[--snip--]
```

The array `i_[]` contains the 'directions' for each digits: it contains +1 or -1 if the computation of the successor will increase or decrease the corresponding digit. It has to be filled when the first or last number is computed:

```
    void first()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = 0;
        for (ulong k=0; k<n_; ++k)  i_[k] = +1;
        j_ = n_;
        dm_ = 0;
    }

    void last()
    {
        // find position of last even radix:
        ulong z = 0;
        for (ulong i=0; i<n_; ++i)  if ( m1_[i]&1 )  z = i;
        while ( z<n_ )  // last even .. end:
        {
            a_[z] = m1_[z];
            i_[z] = +1;
            ++z;
        }

        j_ = 0;
        dm_ = -1;
    }
[--snip--]
```

A sentinel element (`i_[n]`=0) is used to optimize the computations of the successor and predecessor:

```
    bool next()
    {
        ulong j = 0;
        ulong ij;
        while ( (ij=i_[j]) )  // can touch sentinel i[n]==0
        {
            ulong dj = a_[j] + ij;
```

```
            if ( dj>m1_[j] )  // =^= if ( (dj>m1_[j]) || ((long)dj<0) )
            {
                i_[j] = -ij;  // flip direction
            }
            else  // can update
            {
                a_[j] = dj;  // update digit
                dm_ = ij;    // save for dir()
                j_ = j;      // save for pos()
                return true;
            }
            ++j;
        }
        return false;
    }
  [--snip--]
```

Note the if-clause, it is an optimized expression equivalent to the one given as comment. The following methods are often useful:

```
    ulong pos()  const  { return j_; }  // position of last change
    int dir()  const  { return dm_; }   // direction of last change
```

The routine for the computation of the predecessor is obtained by changing the statement `ulong dj = a_[j] + ij;` to `ulong dj = a_[j] - ij;`. About 120 million numbers per second for radix 2, and 245 million for radix 8 are generated [FXT: comb/mixedradix-gray-demo.cc].

A loopless algorithm for the computation of the successor taken from [157] is given in [FXT: comb/mixedradix-gray2.h]. It generates about 185 million numbers per second for radix 2, and 225 million for radix 8 [FXT: comb/mixedradix-gray2-demo.cc]. The crucial trick to make the algorithm loopless is the use of 'focus pointers':

```
class mixedradix_gray2
{
public:
    ulong *a_;  // digits
    ulong *m1_; // radix minus one ('nines')
    ulong *f_;  // focus pointer
    ulong *d_;  // direction
    ulong n_;   // number of digits
    ulong j_;   // position of last change
    int dm_;    // direction of last move
  [--snip--]
    void first()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = 0;
        for (ulong k=0; k<n_; ++k)  d_[k] = 1;
        for (ulong k=0; k<=n_; ++k)  f_[k] = k;
        dm_ = 0;
        j_ = n_;
    }

    bool next()
    {
        const ulong j = f_[0];
        f_[0] = 0;

        if ( j>=n_ )  { first();  return false; }

        const ulong dj = d_[j];
        const ulong aj = a_[j] + dj;
        a_[j] = aj;

        dm_ = (int)dj;  // save for dir()
        j_ = j;         // save for pos()

        if ( aj+dj > m1_[j] )  // was last move?
        {
            d_[j] = -dj;      // change direction
            f_[j] = f_[j+1];  // lookup next position
            f_[j+1] = j + 1;
        }
        return true;
    }
```

Modular Gray code order

```
          M=[ 2 3 4 ]     j              M=[ 4 3 2 ]     j
     0:     [ . . . ]                0:     [ . . . ]
     1:     [ 1 . . ]     0          1:     [ 1 . . ]     0
     2:     [ 1 1 . ]     1          2:     [ 2 . . ]     0
     3:     [ . 1 . ]     0          3:     [ 3 . . ]     0
     4:     [ . 2 . ]     1          4:     [ 3 1 . ]     1
     5:     [ 1 2 . ]     0          5:     [ . 1 . ]     0
     6:     [ 1 2 1 ]     2          6:     [ 1 1 . ]     0
     7:     [ . 2 1 ]     0          7:     [ 2 1 . ]     0
     8:     [ . . 1 ]     1          8:     [ 2 2 . ]     1
     9:     [ 1 . 1 ]     0          9:     [ 3 2 . ]     0
    10:     [ 1 1 1 ]     1         10:     [ . 2 . ]     0
    11:     [ . 1 1 ]     0         11:     [ 1 2 . ]     0
    12:     [ . 1 2 ]     2         12:     [ 1 2 1 ]     2
    13:     [ 1 1 2 ]     0         13:     [ 2 2 1 ]     0
    14:     [ 1 2 2 ]     1         14:     [ 3 2 1 ]     0
    15:     [ . 2 2 ]     0         15:     [ . 2 1 ]     0
    16:     [ . . 2 ]     1         16:     [ . . 1 ]     1
    17:     [ 1 . 2 ]     0         17:     [ 1 . 1 ]     0
    18:     [ 1 . 3 ]     2         18:     [ 2 . 1 ]     0
    19:     [ . . 3 ]     0         19:     [ 3 . 1 ]     0
    20:     [ . 1 3 ]     1         20:     [ 3 1 1 ]     1
    21:     [ 1 1 3 ]     0         21:     [ . 1 1 ]     0
    22:     [ 1 2 3 ]     1         22:     [ 1 1 1 ]     0
    23:     [ . 2 3 ]     0         23:     [ 2 1 1 ]     0
```

Figure 9.2-B: Mixed radix numbers in Gray code order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). The columns 'j' give the position of last change.

Figure 9.2-B shows the 3-digit mixed radix numbers for radix vectors $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right) in *modular Gray code* order. The transitions are either $k \to k + 1$ or, if k is maximal, $k \to 0$. The listing was created with the program [FXT: comb/mixedradix-modular-gray-demo.cc]. The loopless implementation [FXT: `class mixedradix_modular_gray` in comb/mixedradix-modular-gray.h] taken from [157] generates between about 135 million (radix 2) and 230 million (radix 16) numbers per second.

9.3 gslex order

The algorithm for the generation of subsets in lexicographic order given in section 8.1.2 on page 192 can be generalized for mixed radix numbers. Figure 9.3-A shows the 3-digit mixed radix numbers for basis vector $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Note that zero is the last word in this order. For lack of a better name we call the order *gslex* (for generalized subset-lex) order. A routine for generating successive words in gslex order is implemented in [FXT: `class mixedradix_gslex` in comb/mixedradix-gslex.h]:

```
class mixedradix_gslex
{
public:
    ulong n_;   // n-digit numbers
    ulong *a_;  // digits
    ulong *m1_; // m1[k] == radix-1 at position k
public:
    mixedradix_gslex(ulong n, ulong mm, const ulong *m=0)
    {
        n_ = n;
        a_ = new ulong[n_ + 1];
        a_[n_] = 1; // sentinel
        m1_ = new ulong[n_];
        mixedradix_init(n_, mm, m, m1_);
        first();
    }
  [--snip--]
    void first()
```

```
            M=[ 2 3 4 ]     x                M=[ 4 3 2 ]     x
 0:           [ 1 . . ]     1           0:       [ 1 . . ]     1
 1:           [ 1 1 . ]     3           1:       [ 2 . . ]     2
 2:           [ . 1 . ]     2           2:       [ 3 . . ]     3
 3:           [ 1 2 . ]     5           3:       [ 1 1 . ]     5
 4:           [ . 2 . ]     4           4:       [ 2 1 . ]     6
 5:           [ 1 . 1 ]     7           5:       [ 3 1 . ]     7
 6:           [ 1 1 1 ]     9           6:       [ . 1 . ]     4
 7:           [ . 1 1 ]     8           7:       [ 1 2 . ]     9
 8:           [ 1 2 1 ]    11           8:       [ 2 2 . ]    10
 9:           [ . 2 1 ]    10           9:       [ 3 2 . ]    11
10:           [ . . 1 ]     6          10:       [ . 2 . ]     8
11:           [ 1 . 2 ]    13          11:       [ 1 . 1 ]    13
12:           [ 1 1 2 ]    15          12:       [ 2 . 1 ]    14
13:           [ . 1 2 ]    14          13:       [ 3 . 1 ]    15
14:           [ 1 2 2 ]    17          14:       [ 1 1 1 ]    17
15:           [ . 2 2 ]    16          15:       [ 2 1 1 ]    18
16:           [ . . 2 ]    12          16:       [ 3 1 1 ]    19
17:           [ 1 . 3 ]    19          17:       [ . 1 1 ]    16
18:           [ 1 1 3 ]    21          18:       [ 1 2 1 ]    21
19:           [ . 1 3 ]    20          19:       [ 2 2 1 ]    22
20:           [ 1 2 3 ]    23          20:       [ 3 2 1 ]    23
21:           [ . 2 3 ]    22          21:       [ . 2 1 ]    20
22:           [ . . 3 ]    18          22:       [ . . 1 ]    12
23:           [ . . . ]     0          23:       [ . . . ]     0
```

Figure 9.3-A: Mixed radix numbers in gslex (generalized subset lex) order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Successive words differ in at most three positions. Columns 'x' give the values.

```
{
    for (ulong k=0; k<n_; ++k)  a_[k] = 0;
    a_[0] = 1;
}
void last()
{
    for (ulong k=0; k<n_; ++k)  a_[k] = 0;
}
```

The method `next()` computes the successor:

```
bool next()
{
    ulong e = 0;
    while ( 0==a_[e] )  ++e;  // can touch sentinel
    if ( e==n_ )  { first();  return false; }  // current is last
    ulong ae = a_[e];
    if ( ae != m1_[e] )  // easy case: simple increment
    {
        a_[0] = 1;
        a_[e] = ae + 1;
    }
    else
    {
        a_[e] = 0;
        if ( a_[e+1]==0 )  // can touch sentinel
        {
            a_[0] = 1;
            ++a_[e+1];
        }
    }
    return true;
}
```

The predecessor is computed by the method `prev()`:

```
bool prev()
{
    ulong e = 0;
    while ( 0==a_[e] )  ++e;  // can touch sentinel
    if ( 0!=e )  // easy case: prepend nine
```

```
        {
            --e;
            a_[e] = m1_[e];
        }
        else
        {
            ulong a0 = a_[0];
            --a0;
            a_[0] = a0;
            if ( 0==a0 )
            {
                do { ++e; }  while ( 0==a_[e] );  // can touch sentinel
                if ( e==n_ )  { last();  return false; }  // current is first
                ulong ae = a_[e];
                --ae;
                a_[e] = ae;
                if ( 0==ae )
                {
                    --e;
                    a_[e] = m1_[e];
                }
            }
        }
        return true;
    }
```

The algorithm is constant amortized time (CAT) and fast in practice. The worst performance occurs when all digits are radix 2, then about 123 million objects can be created per second. With radix 4 about 198 million, with radix 16 about 273 million objects per second are computed [FXT: comb/mixedradix-gslex-demo.cc].

Alternative gslex order

	M=[2 3 4]	x		M=[4 3 2]	x
0:	[. . .]	0	0:	[. . .]	0
1:	[1 . .]	1	1:	[1 . .]	1
2:	[1 1 .]	3	2:	[1 1 .]	5
3:	[1 1 1]	9	3:	[1 1 1]	17
4:	[1 1 2]	15	4:	[1 2 .]	9
5:	[1 1 3]	21	5:	[1 2 1]	21
6:	[1 2 .]	5	6:	[1 . 1]	13
7:	[1 2 1]	11	7:	[2 . .]	2
8:	[1 2 2]	17	8:	[2 1 .]	6
9:	[1 2 3]	23	9:	[2 1 1]	18
10:	[1 . 1]	7	10:	[2 2 .]	10
11:	[1 . 2]	13	11:	[2 2 1]	22
12:	[1 . 3]	19	12:	[2 . 1]	14
13:	[. 1 .]	2	13:	[3 . .]	3
14:	[. 1 1]	8	14:	[3 1 .]	7
15:	[. 1 2]	14	15:	[3 1 1]	19
16:	[. 1 3]	20	16:	[3 2 .]	11
17:	[. 2 .]	4	17:	[3 2 1]	23
18:	[. 2 1]	10	18:	[3 . 1]	15
19:	[. 2 2]	16	19:	[. 1 .]	4
20:	[. 2 3]	22	20:	[. 1 1]	16
21:	[. . 1]	6	21:	[. 2 .]	8
22:	[. . 2]	12	22:	[. 2 1]	20
23:	[. . 3]	18	23:	[. . 1]	12

Figure 9.3-B: Mixed radix numbers in alternative gslex (generalized subset lex) order, dots denote zeros. The radix vectors are $M = [2, 3, 4]$ (left) and $M = [4, 3, 2]$ (right). Successive words differ in at most three positions. Columns '`x`' give the values.

A variant of the gslex order is shown in figure 9.3-B. The ordering can be obtained from the gslex order by reversing the list, reversing the words, and replacing all nonzero digits d_i by $r_i - d_i$ where r_i is the radix at position i. The implementation is given in [FXT: `class mixedradix_gslex_alt` in comb/mixedradix-gslex-alt.h], the rate of generation is about the same as with gslex order [FXT: comb/mixedradix-gslex-alt-demo.cc].

9.4 endo order

	M=[5 6]	x			x
0:	[. .]	0	15:	[. 5]	25
1:	[1 .]	1	16:	[1 5]	26
2:	[3 .]	3	17:	[3 5]	28
3:	[4 .]	4	18:	[4 5]	29
4:	[2 .]	2	19:	[2 5]	27
5:	[. 1]	5	20:	[. 4]	20
6:	[1 1]	6	21:	[1 4]	21
7:	[3 1]	8	22:	[3 4]	23
8:	[4 1]	9	23:	[4 4]	24
9:	[2 1]	7	24:	[2 4]	22
10:	[. 3]	15	25:	[. 2]	10
11:	[1 3]	16	26:	[1 2]	11
12:	[3 3]	18	27:	[3 2]	13
13:	[4 3]	19	28:	[4 2]	14
14:	[2 3]	17	29:	[2 2]	12

Figure 9.4-A: Mixed radix numbers in endo order, dots denote zeros. The radix vector is $M = [5, 6]$. Columns 'x' give the values.

The computation of the successor in mixed radix endo order (see section 6.5.1 on page 175) is very similar to the counting order described section 9.1 on page 207. The implementation [FXT: `class mixedradix_endo` in comb/mixedradix-endo.h] uses an additional array `le_[]` of the last nonzero elements in endo order. Its entries are 2 for $m > 1$, else 1:

```
class mixedradix_endo
{
public:
    ulong *a_;  // digits, sentinel a[n]
    ulong *m1_; // radix (minus one) for each digit
    ulong *le_; // last positive digit in endo order, sentinel le[n]
    ulong n_;   // Number of digits
    ulong j_;   // position of last change

    mixedradix_endo(const ulong *m, ulong n, ulong mm=0)
    {
        n_ = n;
        a_ = new ulong[n_+1];
        a_[n_] = 1;  // sentinel:  != 0
        m1_ = new ulong[n_];

        mixedradix_init(n_, mm, m, m1_);

        le_ = new ulong[n_+1];
        le_[n_] = 0;  // sentinel:  != a[n]
        for (ulong k=0; k<n_; ++k)  le_[k] = 2 - (m1_[k]==1);

        first();
    }
  [--snip--]
```

The first number is all zero, the last can be read form the array `le_[]`:

```
    void first()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = 0;
        j_ = n_;
    }

    void last()
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = le_[k];
        j_ = n_;
    }
  [--snip--]
```

In the computation of the successor the function `next_endo()` is used instead of a simple increment:

```
    bool next()
    {
        bool ret = false;
        ulong j = 0;
```

```
        while ( a_[j]==le_[j] )  { a_[j]=0; ++j; }  // can touch sentinel
        if ( j<n_ ) // only if no overflow
        {
            a_[j] = next_endo(a_[j], m1_[j]);  // increment
            ret = true;
        }
        j_ = j;
        return ret;
    }

    bool prev()
    {
        bool ret = false;
        ulong j = 0;
        while ( a_[j]==0 )  { a_[j]=le_[j]; ++j; }  // can touch sentinel
        if ( j<n_ ) // only if no overflow
        {
            a_[j] = prev_endo(a_[j], m1_[j]);  // decrement
            ret = true;
        }
        j_ = j;
        return ret;
    }
  [--snip--]
```

The function `next()` generates between about 115 million (radix 2) and 180 million (radix 16) numbers per second. The listing in figure 9.4-A was created with the program [FXT: comb/mixedradix-endo-demo.cc].

9.5 Gray code for endo order

	M=[5 6]	x	j	d			x	j	d
0:	[. .]	0			15:	[2 5]	27	1	1
1:	[1 .]	1	0	1	16:	[4 5]	29	0	-1
2:	[3 .]	3	0	1	17:	[3 5]	28	0	-1
3:	[4 .]	4	0	1	18:	[1 5]	26	0	-1
4:	[2 .]	2	0	1	19:	[. 5]	25	0	-1
5:	[2 1]	7	1	1	20:	[. 4]	20	1	1
6:	[4 1]	9	0	-1	21:	[1 4]	21	0	1
7:	[3 1]	8	0	-1	22:	[3 4]	23	0	1
8:	[1 1]	6	0	-1	23:	[4 4]	24	0	1
9:	[. 1]	5	0	-1	24:	[2 4]	22	0	1
10:	[. 3]	15	1	1	25:	[2 2]	12	1	1
11:	[1 3]	16	0	1	26:	[4 2]	14	0	-1
12:	[3 3]	18	0	1	27:	[3 2]	13	0	-1
13:	[4 3]	19	0	1	28:	[1 2]	11	0	-1
14:	[2 3]	17	0	1	29:	[. 2]	10	0	-1

Figure 9.5-A: Mixed radix numbers in endo Gray code, dots denote zeros. The radix vector is $M = [4, 5]$. Columns '`x`' give the values, columns '`j`' and '`d`' give the position of last change and its direction, respectively.

A Gray code for mixed radix numbers in endo order can be obtained by a modification of the CAT algorithm for the Gray code described in section 9.2 on page 210. In the computation of the last number, the last digit have to be set to the last endo digit [FXT: `class mixedradix_endo_gray` in comb/mixedradix-endo-gray.h]:

```
class mixedradix_endo_gray
{
public:
    ulong *a_;  // mixed radix digits
    ulong *m1_; // radices (minus one)
    ulong *i_;  // direction
    ulong *le_; // last positive digit in endo order
    ulong n_;   // n_ digits
    ulong j_;   // position of last change
```

```
  int dm_;    // direction of last move
[--snip--]
  void first()
  {
      for (ulong k=0; k<n_; ++k)  a_[k] = 0;
      for (ulong k=0; k<n_; ++k)  i_[k] = +1;
      j_ = n_;
      dm_ = 0;
  }

  void last()
  {
      for (ulong k=0; k<n_; ++k)  a_[k] = 0;
      for (ulong k=0; k<n_; ++k)  i_[k] = -1UL;

      // find position of last even radix:
      ulong z = 0;
      for (ulong i=0; i<n_; ++i)  if ( m1_[i]&1 )  z = i;
      while ( z<n_ )  // last even .. end:
      {
          a_[z] = m1_[z];
          i_[z] = +1;
          ++z;
      }

      j_ = 0;
      dm_ = -1;
  }
[--snip--]
```

The successor is computed as follows:

```
  bool next()
  {
      ulong j = 0;
      ulong ij;
      while ( (ij=i_[j]) )  // can touch sentinel i[n]==0
      {
          ulong dj;
          bool ovq;  // overflow?
          if ( ij == 1 )
          {
              dj = next_endo(a_[j], m1_[j]);
              ovq = (dj==0);
          }
          else
          {
              ovq = (a_[j]==0);
              dj = prev_endo(a_[j], m1_[j]);
          }

          if ( ovq )  i_[j] = -ij;
          else
          {
              a_[j] = dj;
              dm_ = ij;
              j_ = j;
              return true;
          }

          ++j;
      }
      return false;
  }
[--snip--]
```

The routine for computation of the predecessor is obtained by changing the condition `if ( ij == 1 )` to `if ( ij != 1 )`. About 65 million (radix 2) and 110 million (radix 16) numbers per second are generated. The listing in figure 9.5-A was created with the program [FXT: comb/mixedradix-endo-gray-demo.cc].

Chapter 10

Permutations

In this section algorithms for the generation of all permutations are presented. These are typically useful in situations where an exhaustive search over all permutations is needed. Some algorithms use mixed radix numbers with factorial base, see section 10.3 on page 222.

Algorithms for application, inversion and composition of permutations and the generation of random permutations are given in chapter 2 on page 85. The sign (parity) of a permutation is defined in section 2.11.5 on page 108.

A important optimization technique is to use arrays instead of pointers. One would change the pointer declarations to array declarations in the corresponding class as follows:

```
    //ulong *p_;   // permutation data (pointer version)
    ulong p_[32];  // permutation data (array version)
```

One also needs to disables the statements to allocate and free memory with the pointers. Here we assume that nobody would attempt to compute all permutations of 31 or more elements ($31! \approx 8.22 \cdot 10^{33}$, taking about $1.3 \cdot 10^{18}$ years to finish). To use arrays uncomment (in most implementations) a line like

```
#define PERM_REV2_FIXARRAYS // use arrays instead of pointers (speedup)
```

near the top of the header file. Whether the use of arrays tends to give a speedup is noted in the comment, as above.

10.1 Lexicographic order

When generated in lexicographic order the permutations appear as if (read as numbers and) sorted numerically in ascending order, see figure 10.1-A. The first half of the inverse permutations are the reversed inverse permutations in the second half: the position of zero in the first half of the inverse permutations lies in the first half of each permutation, so their reversal gives the second half. Write I for the operator that inverts a permutation, C for the complement, and R for reversal. Then we have

$$C \;=\; I\,R\,I \tag{10.1-1}$$

and thereby the first half of the permutations are the complements of the permutations in the second half. An implementation of an iterative algorithm is [FXT: `class perm_lex` in comb/perm-lex.h].

```
class perm_lex
{
public:
    ulong *p_;  // permutation in 0, 1, ..., n-1, sentinel at [-1]
    ulong n_;   // number of elements to permute
public:
    perm_lex(ulong n)
```

	permutation	inv. perm.	compl. inv. perm.	reversed perm.
0:	[. 1 2 3]	[. 1 2 3]	[3 2 1 .]	[3 2 1 .]
1:	[. 1 3 2]	[. 1 3 2]	[3 2 . 1]	[2 3 1 .]
2:	[. 2 1 3]	[. 2 1 3]	[3 1 2 .]	[3 1 2 .]
3:	[. 2 3 1]	[. 3 1 2]	[3 . 2 1]	[1 3 2 .]
4:	[. 3 1 2]	[. 2 3 1]	[3 1 . 2]	[2 1 3 .]
5:	[. 3 2 1]	[. 3 2 1]	[3 . 1 2]	[1 2 3 .]
6:	[1 . 2 3]	[1 . 2 3]	[2 3 1 .]	[3 2 . 1]
7:	[1 . 3 2]	[1 . 3 2]	[2 3 . 1]	[2 3 . 1]
8:	[1 2 . 3]	[2 . 1 3]	[1 3 2 .]	[3 . 2 1]
9:	[1 2 3 .]	[3 . 1 2]	[. 3 2 1]	[. 3 2 1]
10:	[1 3 . 2]	[2 . 3 1]	[1 3 . 2]	[2 . 3 1]
11:	[1 3 2 .]	[3 . 2 1]	[. 3 1 2]	[. 2 3 1]
12:	[2 . 1 3]	[1 2 . 3]	[2 1 3 .]	[3 1 . 2]
13:	[2 . 3 1]	[1 3 . 2]	[2 . 3 1]	[1 3 . 2]
14:	[2 1 . 3]	[2 1 . 3]	[1 2 3 .]	[3 . 1 2]
15:	[2 1 3 .]	[3 1 . 2]	[. 2 3 1]	[. 3 1 2]
16:	[2 3 . 1]	[2 3 . 1]	[1 . 3 2]	[1 . 3 2]
17:	[2 3 1 .]	[3 2 . 1]	[. 1 3 2]	[. 1 3 2]
18:	[3 . 1 2]	[1 2 3 .]	[2 1 . 3]	[2 1 . 3]
19:	[3 . 2 1]	[1 3 2 .]	[2 . 1 3]	[1 2 . 3]
20:	[3 1 . 2]	[2 1 3 .]	[1 2 . 3]	[2 . 1 3]
21:	[3 1 2 .]	[3 1 2 .]	[. 2 1 3]	[. 2 1 3]
22:	[3 2 . 1]	[2 3 1 .]	[1 . 2 3]	[1 . 2 3]
23:	[3 2 1 .]	[3 2 1 .]	[. 1 2 3]	[. 1 2 3]

Figure 10.1-A: All permutations of 4 elements in lexicographic order, their inverses, the complements of the inverses, and the reversed permutations. Dots denote zeros.

```
    {
        n_ = n;
        p_ = new ulong[n_+1];
        p_[0] = 0;  // sentinel
        ++p_;
        first();
    }

    ~perm_lex() { --p_;  delete [] p_; }

    void first()  { for (ulong i=0; i<n_; i++)  p_[i] = i; }

    const ulong *data()  const  { return p; }
  [--snip--]
```

The only nontrivial part is the `next()`-method that computes the next permutation with each call. The routine `perm_lex::next()` is based on code by Glenn Rhoads

```
    bool next()
    {
        // find for rightmost pair with p_[i] < p_[i+1]:
        const ulong n1 = n_ - 1;
        ulong i = n1;
        do  { --i; }  while ( p_[i] > p_[i+1] );
        if ( (long)i<0 )  return false;  // last sequence is falling seq.

        // find rightmost element p[j] smaller than p[i]:
        ulong j = n1;
        while ( p_[i] > p_[j] )  { --j; }

        swap2(p_[i], p_[j]);

        // Here the elements p[i+1], ..., p[n-1] are a falling sequence.
        // Reverse order to the right:
        ulong r = n1;
        ulong s = i + 1;
        while ( r > s )  { swap2(p_[r], p_[s]);  --r;  ++s; }

        return true;
    }
```

The routine generates about 113 million permutations per second. Using the class is no black magic [FXT: comb/perm-lex-demo.cc]:

```
    ulong n = 4;
    perm_lex P(n);
```

```
    do
    {
        // visit permutation
    }
    while ( P.next() );
```

A slightly faster algorithm is obtained by making the changes with the update operation for the co-lexicographic order (section 10.2) on the right end of the permutations [FXT: comb/perm-lex2.h]. When arrays are used instead of pointers the rate is about 133 million per second [FXT: comb/perm-lex2-demo.cc]. With pointers the rate is about 115 million per second.

10.2 Co-lexicographic order

	permutation	rfact	inv. perm.
0:	[3 2 1 .]	[. . .]	[3 2 1 .]
1:	[2 3 1 .]	[1 . .]	[3 2 . 1]
2:	[3 1 2 .]	[. 1 .]	[3 1 2 .]
3:	[1 3 2 .]	[1 1 .]	[3 . 2 1]
4:	[2 1 3 .]	[. 2 .]	[3 1 . 2]
5:	[1 2 3 .]	[1 2 .]	[3 . 1 2]
6:	[3 2 . 1]	[. . 1]	[2 3 1 .]
7:	[2 3 . 1]	[1 . 1]	[2 3 . 1]
8:	[3 . 2 1]	[. 1 1]	[1 3 2 .]
9:	[. 3 2 1]	[1 1 1]	[. 3 2 1]
10:	[2 . 3 1]	[. 2 1]	[1 3 . 2]
11:	[. 2 3 1]	[1 2 1]	[. 3 1 2]
12:	[3 1 . 2]	[. . 2]	[2 1 3 .]
13:	[1 3 . 2]	[1 . 2]	[2 . 3 1]
14:	[3 . 1 2]	[. 1 2]	[1 2 3 .]
15:	[. 3 1 2]	[1 1 2]	[. 2 3 1]
16:	[1 . 3 2]	[. 2 2]	[1 . 3 2]
17:	[. 1 3 2]	[1 2 2]	[. 1 3 2]
18:	[2 1 . 3]	[. . 3]	[2 1 . 3]
19:	[1 2 . 3]	[1 . 3]	[2 . 1 3]
20:	[2 . 1 3]	[. 1 3]	[1 2 . 3]
21:	[. 2 1 3]	[1 1 3]	[. 2 1 3]
22:	[1 . 2 3]	[. 2 3]	[1 . 2 3]
23:	[. 1 2 3]	[1 2 3]	[. 1 2 3]

Figure 10.2-A: The permutations of 4 elements in co-lexicographic order. Dots denote zeros.

Figure 10.2-A shows the permutations of 4 elements in co-lexicographic (colex) order. An algorithm for the generation is implemented in [FXT: `class perm_colex` in comb/perm-colex.h]:

```
class perm_colex
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, ...]
    ulong *x_;  // permutation
    ulong n_;   // permutations of n elements
public:
    perm_colex(ulong n)
    // Must have n>=2
    {
        n_ = n;
        d_ = new ulong[n_];
        d_[n-1] = 0;  // sentinel
        x_ = new ulong[n_];
        first();
    }
  [--snip--]
    void first()
    {
        for (ulong k=0; k<n_; ++k)  x_[k] = n_-1-k;
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
    }
```

The update process uses the falling factorial numbers. Let j be the position where the digit is incremented, and d the value before the increment. The update

```
  permutation       ffact
                          v-- increment at j=3
[ 0 3 4 5 2 1 ]   [ 1 2 3 1 1 ]  <--= digit before increment is d=1
[ 5 4 2 0 3 1 ]   [ . . . 2 1 ]
```

is done in three steps:

```
[ 0 3 4 5 2 1 ]   [ 1 2 3 1 1 ]
[ 0 2 4 5 3 1 ]   [ 1 2 3 2 1 ]  <--= swap positions d=1 and j+1=4
[ 5 4 2 0 3 1 ]   [ . . . 2 1 ]  <--= reverse range 0...j
```

```
bool next()
{
    if ( d_[0]==0 ) // easy case
    {
        d_[0] = 1;
        swap2(x_[0], x_[1]);
        return true;
    }
    else
    {
        d_[0] = 0;
        ulong j = 1;
        ulong m1 = 2;  // nine in falling factorial base
        while ( d_[j]==m1 )
        {
            d_[j] = 0;
            ++m1;
            ++j;
        }

        if ( j==n_-1 )  return false;  // current permutation is last

        const ulong dj = d_[j];
        d_[j] = dj + 1;

        swap2( x_[dj], x_[j+1] );  // swap positions dj and j+1

        { // reverse range [0...j]:
            ulong a = 0,  b = j;
            do
            {
                swap2(x_[a], x_[b]);
                ++a;
                --b;
            }
            while ( a<b );
        }

        return true;
    }
}
}
```

About 194 million permutations per second can be generated [FXT: comb/perm-colex-demo.cc]. With arrays instead of pointers the rate is 210 million per second.

10.3 Factorial representations of permutations

The factorial number system corresponds to the mixed radix bases $M = [2, 3, 4, \ldots]$ (*rising factorial basis*) or $M = [\ldots, 4, 3, 2]$ (*falling factorial basis*). A $(n-1)$-digit factorial number can have $n!$ different values. We develop different methods to convert factorial numbers to permutations and vice versa.

10.3.1 The Lehmer code

Each permutation of n distinct elements can be converted to a unique $(n-1)$-digit factorial number $A = [a_0, a_1, \ldots, a_{n-2}]$ in the falling factorial base by counting, for each index k, the number of elements

with indices $j > k$ that are bigger than the current element [FXT: comb/fact2perm.cc]:

```
void perm2ffact(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into
//   the (n-1) digit factorial representation in fc[0,...,n-2].
// One has: fc[0]<n, fc[1]<n-1, ... , fc[n-2]<2 (falling radices)
{
    for (ulong k=0; k<n-1; ++k)
    {
        ulong xk = x[k];
        ulong i = 0;
        for (ulong j=k; j<n; ++j)  if ( x[j]<xk )  ++i;
        fc[k] = i;
    }
}
```

The routine works as long as all elements of the permutation are distinct. The factorial representation obtained by this method is called the *Lehmer code* of the permutation. For example, the permutation $[3, 0, 1, 4, 2]$ has the Lehmer code $[3, 0, 0, 1]$, because three elements smaller than the first element (3) lie right to it, no elements smaller than the second element (0) lies right to it, etc.

A routine that computes the permutation for a given Lehmer code is

```
void ffact2perm(const ulong *fc, ulong n, ulong *x)
// Inverse of perm2ffact():
// Convert the (n-1) digit factorial representation in fc[0,...,n-2].
// into permutation in x[0,...,n-1]
// Must have: fc[0]<n, fc[1]<n-1, ... , fc[n-2]<2 (falling radices)
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    for (ulong k=0; k<n-1; ++k)
    {
        ulong fa = fc[k];
        if ( fa ) rotate_right1(x+k, fa+1);
    }
}
```

A routine to compute the inverse permutation is

```
void ffact2invperm(const ulong *fc, ulong n, ulong *x)
// Convert the (n-1) digit factorial representation in fc[0,...,n-2].
// into permutation in x[0,...,n-1] such that
// the permutation is the inverse of the one computed via ffact2perm().
{
    for (ulong k=0; k<n; ++k)  x[k] = n-1;  // "empty"
    for (ulong k=0; k<n-1; ++k)
    {
        ulong fa = fc[k];
        for (ulong j=0;  ; ++j)
        {
            if ( x[j]==n-1 )  // if empty
            {
                if ( 0==fa )  { x[j] = k;  break; }
                --fa;
            }
        }
    }
}
```

A similar method can compute a representation in the rising factorial base:

```
void perm2rfact(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into
//   the (n-1) digit factorial representation in fc[0,...,n-2].
// One has: fc[0]<2, fc[1]<3, ... , fc[n-2]<n (rising radices)
{
    for (ulong k=1; k<n; ++k)
    {
        ulong xk = x[k];
        ulong i = 0;
        for (ulong j=0; j<k; ++j)  if ( x[j]>xk )  ++i;
        fc[k-1] = i;
    }
```

	ffact	permutation	rev.compl.perm.	rfact
0:	[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
1:	[1 . .]	[1 . 2 3]	[. 1 3 2]	[. . 1]
2:	[2 . .]	[2 . 1 3]	[. 2 3 1]	[. . 2]
3:	[3 . .]	[3 . 1 2]	[1 2 3 .]	[. . 3]
4:	[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
5:	[1 1 .]	[1 2 . 3]	[. 3 1 2]	[. 1 1]
6:	[2 1 .]	[2 1 . 3]	[. 3 2 1]	[. 1 2]
7:	[3 1 .]	[3 1 . 2]	[1 3 2 .]	[. 1 3]
8:	[. 2 .]	[. 3 1 2]	[1 2 . 3]	[. 2 .]
9:	[1 2 .]	[1 3 . 2]	[1 3 . 2]	[. 2 1]
10:	[2 2 .]	[2 3 . 1]	[2 3 . 1]	[. 2 2]
11:	[3 2 .]	[3 2 . 1]	[2 3 1 .]	[. 2 3]
12:	[. . 1]	[. 1 3 2]	[1 . 2 3]	[1 . .]
13:	[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
14:	[2 . 1]	[2 . 3 1]	[2 . 3 1]	[1 . 2]
15:	[3 . 1]	[3 . 2 1]	[2 1 3 .]	[1 . 3]
16:	[. 1 1]	[. 2 3 1]	[2 . 1 3]	[1 1 .]
17:	[1 1 1]	[1 2 3 .]	[3 . 1 2]	[1 1 1]
18:	[2 1 1]	[2 1 3 .]	[3 . 2 1]	[1 1 2]
19:	[3 1 1]	[3 1 2 .]	[3 1 2 .]	[1 1 3]
20:	[. 2 1]	[. 3 2 1]	[2 1 . 3]	[1 2 .]
21:	[1 2 1]	[1 3 2 .]	[3 1 . 2]	[1 2 1]
22:	[2 2 1]	[2 3 1 .]	[3 2 . 1]	[1 2 2]
23:	[3 2 1]	[3 2 1 .]	[3 2 1 .]	[1 2 3]

Figure 10.3-A: Numbers in falling factorial basis and permutations so that the number is the Lehmer code of it (left columns). Dots denote zeros. The rising factorial representation of the reversed and complemented permutation equals the reversed Lehmer code (right columns).

	rfact	permutation	rev.compl.perm.	ffact
0:	[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
1:	[1 . .]	[1 . 2 3]	[. 1 3 2]	[. . 1]
2:	[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
3:	[1 1 .]	[2 . 1 3]	[. 2 3 1]	[. 1 1]
4:	[. 2 .]	[1 2 . 3]	[. 3 1 2]	[. 2 .]
5:	[1 2 .]	[2 1 . 3]	[. 3 2 1]	[. 2 1]
6:	[. . 1]	[. 1 3 2]	[1 . 2 3]	[1 . .]
7:	[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
8:	[. 1 1]	[. 3 1 2]	[1 2 . 3]	[1 1 .]
9:	[1 1 1]	[3 . 1 2]	[1 2 3 .]	[1 1 1]
10:	[. 2 1]	[1 3 . 2]	[1 3 . 2]	[1 2 .]
11:	[1 2 1]	[3 1 . 2]	[1 3 2 .]	[1 2 1]
12:	[. . 2]	[. 2 3 1]	[2 . 1 3]	[2 . .]
13:	[1 . 2]	[2 . 3 1]	[2 . 3 1]	[2 . 1]
14:	[. 1 2]	[. 3 2 1]	[2 1 . 3]	[2 1 .]
15:	[1 1 2]	[3 . 2 1]	[2 1 3 .]	[2 1 1]
16:	[. 2 2]	[2 3 . 1]	[2 3 . 1]	[2 2 .]
17:	[1 2 2]	[3 2 . 1]	[2 3 1 .]	[2 2 1]
18:	[. . 3]	[1 2 3 .]	[3 . 1 2]	[3 . .]
19:	[1 . 3]	[2 1 3 .]	[3 . 2 1]	[3 . 1]
20:	[. 1 3]	[1 3 2 .]	[3 1 . 2]	[3 1 .]
21:	[1 1 3]	[3 1 2 .]	[3 1 2 .]	[3 1 1]
22:	[. 2 3]	[2 3 1 .]	[3 2 . 1]	[3 2 .]
23:	[1 2 3]	[3 2 1 .]	[3 2 1 .]	[3 2 1]

Figure 10.3-B: Numbers in rising factorial basis and permutations so that the number is the Lehmer code of it (left columns). The reversed and complemented permutations and their falling factorial representations are shown in the right columns. They appear in lexicographic order.

```
}
```

Here we count, starting with the second element of the permutation, the number of elements to the left that are bigger than the current element. The inverse routine is

```
void rfact2perm(const ulong *fc, ulong n, ulong *x)
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    ulong *y = x+n;
    for (ulong k=n-1;  k!=0;  --k, --y)
    {
        ulong fa = fc[k-1];
        if ( fa )
        {
            ++fa;
            rotate_left1(y-fa, fa);
        }
    }
}
```

A routine for the inverse permutation is

```
void rfact2invperm(const ulong *fc, ulong n, ulong *x)
// Convert the (n-1) digit factorial representation in fc[0,...,n-2].
//  into permutation in x[0,...,n-1] such that
// the permutation is the inverse of the one computed via rfact2perm().
{
    for (ulong k=0; k<n; ++k)  x[k] = 0;  // "empty"
    for (ulong k=n-2; (long)k>=0; --k)
    {
        ulong fa = fc[k];
        for (ulong j=0;  ; ++j)
        {
            if ( x[j]==0 )  // if empty
            {
                if ( 0==fa )  { x[j] = k+1;  break; }
                --fa;
            }
        }
    }
}
```

The permutations corresponding to the Lehmer codes (in counting order) are shown in figure 10.3-A (left columns). The permutation whose rising factorial representation is the digit-reversed Lehmer code is obtained be reversing and complementing (replacing each element x by $n-1-x$) the original permutation:

Lehmer code	permutation	rev.perm	compl.rev.perm	rising fact
[3,0,0,1]	[3,0,1,4,2]	[2,4,1,0,3]	[2,0,3,4,1]	[1,0,0,3]

The permutations obtained from counting in the rising factorial base are shown in figure 10.3-B.

10.3.2 An representation via reversals

Replacing the rotations in the computation of a permutation from its Lehmer code by reversals one obtains a different one to one relation between factorial numbers and permutations. For the falling factorial basis one gets [FXT: comb/fact2perm-rev.cc]:

```
void perm2ffact_rev(const ulong *x, ulong n, ulong *fc)
{
    ALLOCA(ulong, ti, n);  // inverse permutation
    for (ulong k=0; k<n; ++k)  ti[x[k]] = k;
    for (ulong k=0; k<n-1; ++k)
    {
        ulong j; // find element k
        for (j=k; j<n; ++j)  if ( ti[j]==k )  break;
        j -= k;
        fc[k] = j;
        reverse(ti+k, j+1);
    }
}
```

	ffact	permutation	inv.perm.	ffact
0:	[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
1:	[1 . .]	[1 . 2 3]	[1 . 2 3]	[1 . .]
2:	[2 . .]	[2 1 . 3]	[2 1 . 3]	[2 . .]
3:	[3 . .]	[3 2 1 .]	[3 2 1 .]	[3 . .]
4:	[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
5:	[1 1 .]	[1 2 . 3]	[2 . 1 3]	[2 1 .]
6:	[2 1 .]	[2 . 1 3]	[1 2 . 3]	[1 1 .]
7:	[3 1 .]	[3 1 2 .]	[3 1 2 .]	[3 1 .]
8:	[. 2 .]	[. 3 2 1]	[. 3 2 1]	[. 2 .]
9:	[1 2 .]	[1 3 2 .]	[3 . 2 1]	[3 2 1]
10:	[2 2 .]	[2 3 . 1]	[2 3 . 1]	[2 2 .]
11:	[3 2 .]	[3 . 1 2]	[1 2 3 .]	[1 1 1]
12:	[. . 1]	[. 1 3 2]	[. 1 3 2]	[. . 1]
13:	[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
14:	[2 . 1]	[2 1 3 .]	[3 1 . 2]	[3 1 1]
15:	[3 . 1]	[3 2 . 1]	[2 3 1 .]	[2 2 1]
16:	[. 1 1]	[. 2 3 1]	[. 3 1 2]	[. 2 1]
17:	[1 1 1]	[1 2 3 .]	[3 . 1 2]	[3 2 .]
18:	[2 1 1]	[2 . 3 1]	[1 3 . 2]	[1 2 1]
19:	[3 1 1]	[3 1 . 2]	[2 1 3 .]	[2 . 1]
20:	[. 2 1]	[. 3 1 2]	[. 2 3 1]	[. 1 1]
21:	[1 2 1]	[1 3 . 2]	[2 . 3 1]	[2 1 1]
22:	[2 2 1]	[2 3 1 .]	[3 2 . 1]	[3 . 1]
23:	[3 2 1]	[3 . 2 1]	[1 3 2 .]	[1 2 .]

Figure 10.3-C: Numbers in falling factorial basis and permutations so that the number is the alternative (reversal-) code of it (left columns). The inverse permutations and their rising factorial representations are shown in the right columns.

	rfact	permutation	inv.perm.	rfact
0:	[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
1:	[1 . .]	[1 . 2 3]	[1 . 2 3]	[1 . .]
2:	[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
3:	[1 1 .]	[2 . 1 3]	[1 2 . 3]	[1 2 .]
4:	[. 2 .]	[2 1 . 3]	[2 1 . 3]	[. 2 .]
5:	[1 2 .]	[1 2 . 3]	[2 . 1 3]	[1 1 .]
6:	[. . 1]	[. 1 3 2]	[. 1 3 2]	[. . 1]
7:	[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
8:	[. 1 1]	[. 3 1 2]	[. 2 3 1]	[. 1 2]
9:	[1 1 1]	[3 . 1 2]	[1 2 3 .]	[. 2 3]
10:	[. 2 1]	[3 1 . 2]	[2 1 3 .]	[1 2 3]
11:	[1 2 1]	[1 3 . 2]	[2 . 3 1]	[1 1 2]
12:	[. . 2]	[. 3 2 1]	[. 3 2 1]	[. . 2]
13:	[1 . 2]	[3 . 2 1]	[1 3 2 .]	[1 1 3]
14:	[. 1 2]	[. 2 3 1]	[. 3 1 2]	[. 1 1]
15:	[1 1 2]	[2 . 3 1]	[1 3 . 2]	[1 2 1]
16:	[. 2 2]	[2 3 . 1]	[2 3 . 1]	[. 2 2]
17:	[1 2 2]	[3 2 . 1]	[2 3 1 .]	[1 . 3]
18:	[. . 3]	[3 2 1 .]	[3 2 1 .]	[. . 3]
19:	[1 . 3]	[2 3 1 .]	[3 2 . 1]	[1 2 2]
20:	[. 1 3]	[3 1 2 .]	[3 1 2 .]	[. 1 3]
21:	[1 1 3]	[1 3 2 .]	[3 . 2 1]	[1 . 2]
22:	[. 2 3]	[1 2 3 .]	[3 . 1 2]	[1 1 1]
23:	[1 2 3]	[2 1 3 .]	[3 1 . 2]	[. 2 1]

Figure 10.3-D: Numbers in rising factorial basis and permutations so that the number is the alternative (reversal-) code of it (left columns). The inverse permutations and their falling factorial representations are shown in the right columns.

The routine is the inverse of

```
void ffact2perm_rev(const ulong *fc, ulong n, ulong *x)
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    for (ulong k=0; k<n-1; ++k)
    {
        ulong fa = fc[k];
        // Lehmer: rotate_right1(x+k, fa+1);
        if ( fa ) reverse(x+k, fa+1);
    }
}
```

Figure 10.3-C shows the permutations of 4 elements and their falling factorial representations, it was created with the program [FXT: comb/fact2perm-rev-demo.cc]. The routines for the rising factorial (figure 10.3-D) basis are

```
void perm2rfact_rev(const ulong *x, ulong n, ulong *fc)
{
    ALLOCA(ulong, ti, n);  // inverse permutation
    for (ulong k=0; k<n; ++k)  ti[x[k]] = k;
    for (ulong k=n-1; k!=0; --k)
    {
        ulong j; // find element k
        for (j=0; j<=k; ++j)  if ( ti[j]==k )  break;
        j = k - j;
        fc[k-1] = j;
        reverse(ti+k-j, j+1);
    }
}
```

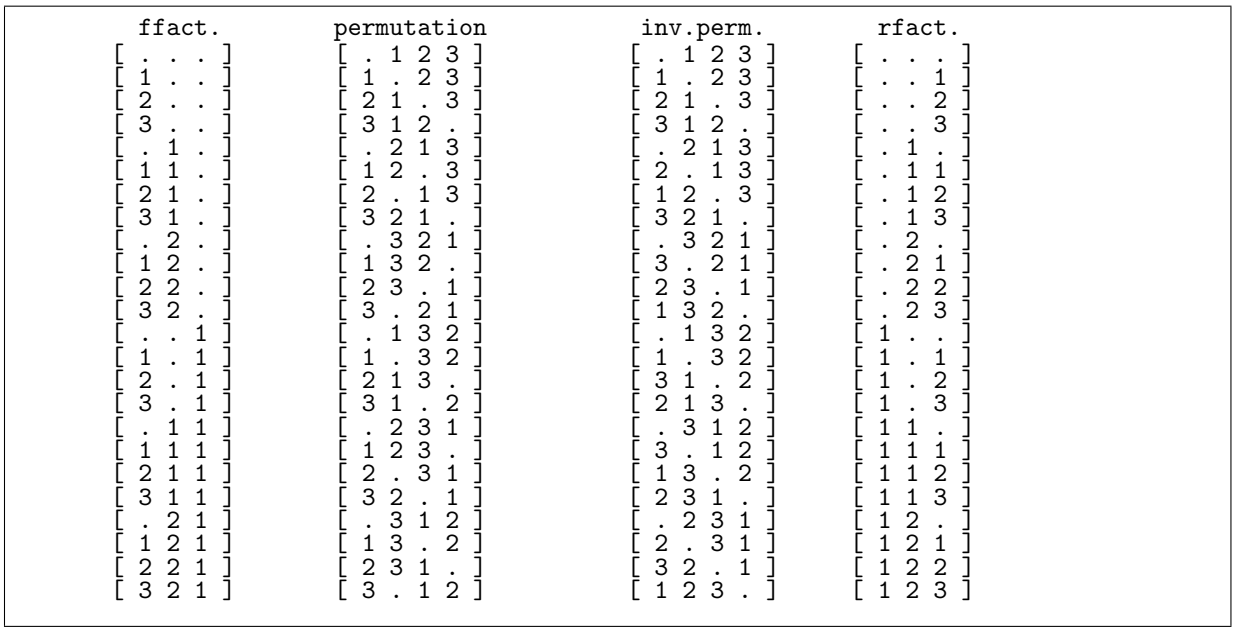
and

```
void rfact2perm_rev(const ulong *fc, ulong n, ulong *x)
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    ulong *y = x+n;
    for (ulong k=n-1;  k!=0;  --k, --y)
    {
        ulong fa = fc[k-1];
        if ( fa )
        {
            ++fa;
            // Lehmer: rotate_left1(y-fa, fa);
            reverse(y-fa, fa);
        }
    }
}
```

10.3.3 A representation via swaps

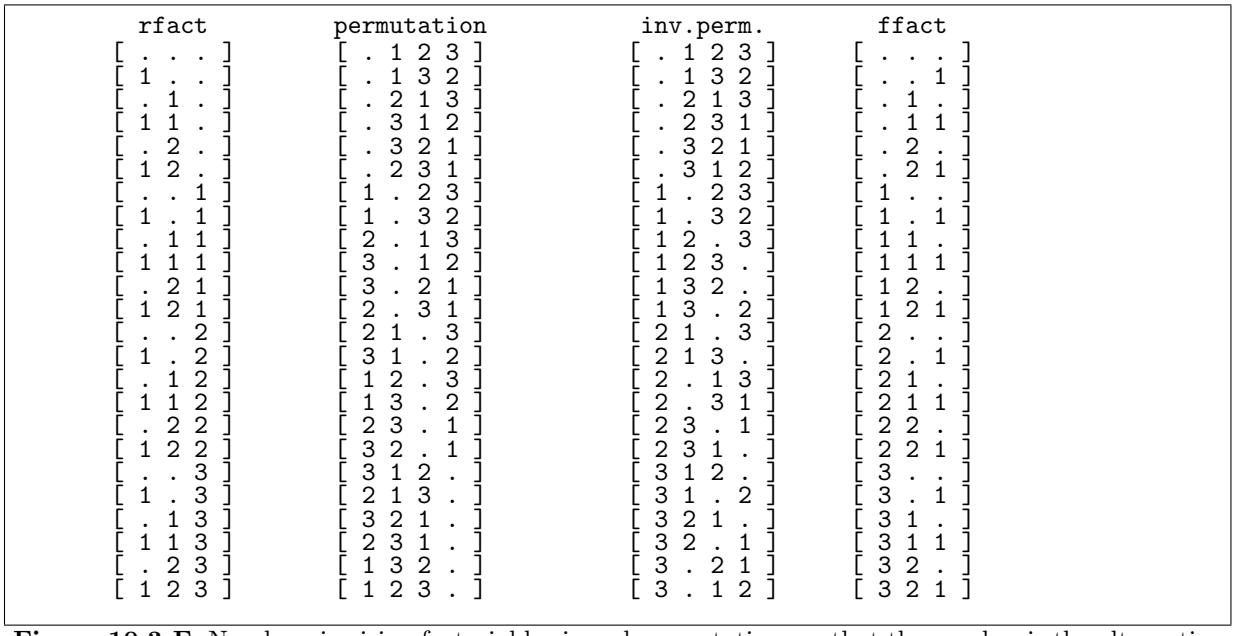
The routines for the conversion from permutations to factorial representations shown so far have complexity n^2. The following routines compute different factorial representations with complexity n [FXT: comb/fact2perm-swp.cc]:

```
void
perm2ffact_swp(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into
//   the (n-1) digit (swaps-) factorial representation in fc[0,...,n-2].
// One has: fc[0]<n, fc[1]<n-1, ... , fc[n-2]<2 (falling radices)
{
    ALLOCA(ulong, t, n);
    for (ulong k=0; k<n; ++k)  t[k] = x[k];
    ALLOCA(ulong, ti, n);  // inverse permutation
    for (ulong k=0; k<n; ++k)  ti[t[k]] = k;

    for (ulong k=0; k<n-1; ++k)
    {
        ulong tk = t[k];  // >= k
        fc[k] = tk - k;
        ulong j = ti[k];  // location of element k
        ti[tk] = ti[t[j]];
```



ffact.	permutation	inv.perm.	rfact.
[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
[1 . .]	[1 . 2 3]	[1 . 2 3]	[. . 1]
[2 . .]	[2 1 . 3]	[2 1 . 3]	[. . 2]
[3 . .]	[3 1 2 .]	[3 1 2 .]	[. . 3]
[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
[1 1 .]	[1 2 . 3]	[2 . 1 3]	[. 1 1]
[2 1 .]	[2 . 1 3]	[1 2 . 3]	[. 1 2]
[3 1 .]	[3 2 1 .]	[3 2 1 .]	[. 1 3]
[. 2 .]	[. 3 2 1]	[. 3 2 1]	[. 2 .]
[1 2 .]	[1 3 2 .]	[3 . 2 1]	[. 2 1]
[2 2 .]	[2 3 . 1]	[2 3 . 1]	[. 2 2]
[3 2 .]	[3 . 2 1]	[1 3 2 .]	[. 2 3]
[. . 1]	[. 1 3 2]	[. 1 3 2]	[1 . .]
[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
[2 . 1]	[2 1 3 .]	[3 1 . 2]	[1 . 2]
[3 . 1]	[3 1 . 2]	[2 1 3 .]	[1 . 3]
[. 1 1]	[. 2 3 1]	[. 3 1 2]	[1 1 .]
[1 1 1]	[1 2 3 .]	[3 . 1 2]	[1 1 1]
[2 1 1]	[2 . 3 1]	[1 3 . 2]	[1 1 2]
[3 1 1]	[3 2 . 1]	[2 3 1 .]	[1 1 3]
[. 2 1]	[. 3 1 2]	[. 2 3 1]	[1 2 .]
[1 2 1]	[1 3 . 2]	[2 . 3 1]	[1 2 1]
[2 2 1]	[2 3 1 .]	[3 2 . 1]	[1 2 2]
[3 2 1]	[3 . 1 2]	[1 2 3 .]	[1 2 3]

Figure 10.3-E: Numbers in falling factorial basis and permutations so that the number is the alternative (swaps-) code of it (left columns). The inverse permutations and their rising factorial representations are shown in the right columns.



rfact	permutation	inv.perm.	ffact
[. . .]	[. 1 2 3]	[. 1 2 3]	[. . .]
[1 . .]	[. 1 3 2]	[. 1 3 2]	[. . 1]
[. 1 .]	[. 2 1 3]	[. 2 1 3]	[. 1 .]
[1 1 .]	[. 3 1 2]	[. 2 3 1]	[. 1 1]
[. 2 .]	[. 3 2 1]	[. 3 2 1]	[. 2 .]
[1 2 .]	[. 2 3 1]	[. 3 1 2]	[. 2 1]
[. . 1]	[1 . 2 3]	[1 . 2 3]	[1 . .]
[1 . 1]	[1 . 3 2]	[1 . 3 2]	[1 . 1]
[. 1 1]	[2 . 1 3]	[1 2 . 3]	[1 1 .]
[1 1 1]	[3 . 1 2]	[1 2 3 .]	[1 1 1]
[. 2 1]	[3 . 2 1]	[1 3 2 .]	[1 2 .]
[1 2 1]	[2 . 3 1]	[1 3 . 2]	[1 2 1]
[. . 2]	[2 1 . 3]	[2 1 . 3]	[2 . .]
[1 . 2]	[3 1 . 2]	[2 1 3 .]	[2 . 1]
[. 1 2]	[1 2 . 3]	[2 . 1 3]	[2 1 .]
[1 1 2]	[1 3 . 2]	[2 . 3 1]	[2 1 1]
[. 2 2]	[2 3 . 1]	[2 3 . 1]	[2 2 .]
[1 2 2]	[3 2 . 1]	[2 3 1 .]	[2 2 1]
[. . 3]	[3 1 2 .]	[3 1 2 .]	[3 . .]
[1 . 3]	[2 1 3 .]	[3 1 . 2]	[3 . 1]
[. 1 3]	[3 2 1 .]	[3 2 1 .]	[3 1 .]
[1 1 3]	[2 3 1 .]	[3 2 . 1]	[3 1 1]
[. 2 3]	[1 3 2 .]	[3 . 2 1]	[3 2 .]
[1 2 3]	[1 2 3 .]	[3 . 1 2]	[3 2 1]

Figure 10.3-F: Numbers in rising factorial basis and permutations so that the number is the alternative (swaps-) code of it (left columns). The inverse permutations and their falling factorial representations are shown in the right columns.

```
        t[j] = tk;
    }
}

void perm2rfact_swp(const ulong *x, ulong n, ulong *fc)
// Convert permutation in x[0,...,n-1] into
//   the (n-1) digit (swaps-) factorial representation in fc[0,...,n-2].
// One has: fc[0]<2, fc[1]<3, ... , fc[n-2]<n (rising radices)
{
    ALLOCA(ulong, t, n);
    for (ulong k=0; k<n; ++k)  t[k] = x[k];
    ALLOCA(ulong, ti, n);  // inverse permutation
    for (ulong k=0; k<n; ++k)  ti[t[k]] = k;

    for (ulong k=0; k<n-1; ++k)
    {
        ulong j = ti[k];  // location of element k, j>=k
        fc[n-2-k] = j - k;
        ulong tk = t[k];
        ti[tk] = ti[t[j]];
        t[j] = tk;
    }
}
```

Their inverses also have complexity n. The routine for falling base is

```
void ffact2perm_swp(const ulong *fc, ulong n, ulong *x)
// Inverse of perm2ffact_swp().
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    for (ulong k=0; k<n-1; ++k)
    {
        ulong fa = fc[k];
        swap2( x[k], x[k+fa] );
    }
}
```

The routine for the rising base is

```
void rfact2perm_swp(const ulong *fc, ulong n, ulong *x)
// Inverse of perm2rfact_swp().
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    for (ulong k=0,j=n-2; k<n-1; ++k,--j)
    {
        ulong fa = fc[k];
        swap2( x[j], x[j+fa] );
    }
}
```

The permutations corresponding to the alternative codes for the falling basis are shown in figure 10.3-E (left columns). The inverse permutation has the rising factorial representation that is digit-reversed representation (right columns). The permutations corresponding to the alternative codes for rising basis are shown in figure 10.3-F The listings were created with the program [FXT: comb/fact2perm-swp-demo.cc]. The routines can serve as a means to find interesting orders of permutations. Indeed, the permutation generator shown in section 10.4 on page 231 was found this way. A recursive algorithm for the (inverse) permutations shown in figure 10.3-F is given in section 10.13 on page 263.

10.3.4 Cyclic permutations

Cyclic permutations of n elements (permutations that consists of one cycle of size n, see section 2.11.4 on page 108) can be obtained from length-$(n-2)$ factorial numbers. We give routines for both falling and rising base [FXT: comb/fact2cyclic.cc]:

```
void ffact2cyclic(const ulong *fc, ulong n, ulong *x)
// Generate cyclic permutation (standard representation) in x[]
//   from the (n-2) digit factorial number in fc[0,...,n-3].
// Falling radices:  [n-1, ..., 3, 2]
{
```

falling fact.	permutation	cycle	inv.perm.
[. . .]	[1 2 3 4 0]	(0, 1, 2, 3, 4,)	[4 0 1 2 3]
[1 . .]	[4 2 3 0 1]	(0, 4, 1, 2, 3,)	[3 4 1 2 0]
[2 . .]	[1 4 3 0 2]	(0, 1, 4, 2, 3,)	[3 0 4 2 1]
[3 . .]	[1 2 4 0 3]	(0, 1, 2, 4, 3,)	[3 0 1 4 2]
[. 1 .]	[3 2 4 1 0]	(0, 3, 1, 2, 4,)	[4 3 1 0 2]
[1 1 .]	[3 2 0 4 1]	(0, 3, 4, 1, 2,)	[2 4 1 0 3]
[2 1 .]	[3 4 0 1 2]	(0, 3, 1, 4, 2,)	[2 3 4 0 1]
[3 1 .]	[4 2 0 1 3]	(0, 4, 3, 1, 2,)	[2 3 1 4 0]
[. 2 .]	[1 3 4 2 0]	(0, 1, 3, 2, 4,)	[4 0 3 1 2]
[1 2 .]	[4 3 0 2 1]	(0, 4, 1, 3, 2,)	[2 4 3 1 0]
[2 2 .]	[1 3 0 4 2]	(0, 1, 3, 4, 2,)	[2 0 4 1 3]
[3 2 .]	[1 4 0 2 3]	(0, 1, 4, 3, 2,)	[2 0 3 4 1]
[. . 1]	[2 3 1 4 0]	(0, 2, 1, 3, 4,)	[4 2 0 1 3]
[1 . 1]	[2 3 4 0 1]	(0, 2, 4, 1, 3,)	[3 4 0 1 2]
[2 . 1]	[4 3 1 0 2]	(0, 4, 2, 1, 3,)	[3 2 4 1 0]
[3 . 1]	[2 4 1 0 3]	(0, 2, 1, 4, 3,)	[3 2 0 4 1]
[. 1 1]	[2 4 3 1 0]	(0, 2, 3, 1, 4,)	[4 3 0 2 1]
[1 1 1]	[2 0 3 4 1]	(0, 2, 3, 4, 1,)	[1 4 0 2 3]
[2 1 1]	[4 0 3 1 2]	(0, 4, 2, 3, 1,)	[1 3 4 2 0]
[3 1 1]	[2 0 4 1 3]	(0, 2, 4, 3, 1,)	[1 3 0 4 2]
[. 2 1]	[3 4 1 2 0]	(0, 3, 2, 1, 4,)	[4 2 3 0 1]
[1 2 1]	[3 0 4 2 1]	(0, 3, 2, 4, 1,)	[1 4 3 0 2]
[2 2 1]	[3 0 1 4 2]	(0, 3, 4, 2, 1,)	[1 2 4 0 3]
[3 2 1]	[4 0 1 2 3]	(0, 4, 3, 2, 1,)	[1 2 3 4 0]

Figure 10.3-G: Numbers in falling factorial basis and the corresponding cyclic permutations.

rising fact.	permutation	cycle	inv.perm.
[. . .]	[1 2 3 4 0]	(0, 1, 2, 3, 4,)	[4 0 1 2 3]
[1 . .]	[2 3 1 4 0]	(0, 2, 1, 3, 4,)	[4 2 0 1 3]
[. 1 .]	[3 2 4 1 0]	(0, 3, 1, 2, 4,)	[4 3 1 0 2]
[1 1 .]	[2 4 3 1 0]	(0, 2, 3, 1, 4,)	[4 3 0 2 1]
[. 2 .]	[1 3 4 2 0]	(0, 1, 3, 2, 4,)	[4 0 3 1 2]
[1 2 .]	[3 4 1 2 0]	(0, 3, 2, 1, 4,)	[4 2 3 0 1]
[. . 1]	[4 2 3 0 1]	(0, 4, 1, 2, 3,)	[3 4 1 2 0]
[1 . 1]	[2 3 4 0 1]	(0, 2, 4, 1, 3,)	[3 4 0 1 2]
[. 1 1]	[3 2 0 4 1]	(0, 3, 4, 1, 2,)	[2 4 1 0 3]
[1 1 1]	[2 0 3 4 1]	(0, 2, 3, 4, 1,)	[1 4 0 2 3]
[. 2 1]	[4 3 0 2 1]	(0, 4, 1, 3, 2,)	[2 4 3 1 0]
[1 2 1]	[3 0 4 2 1]	(0, 3, 2, 4, 1,)	[1 4 3 0 2]
[. . 2]	[1 4 3 0 2]	(0, 1, 4, 2, 3,)	[3 0 4 2 1]
[1 . 2]	[4 3 1 0 2]	(0, 4, 2, 1, 3,)	[3 2 4 1 0]
[. 1 2]	[3 4 0 1 2]	(0, 3, 1, 4, 2,)	[2 3 4 0 1]
[1 1 2]	[4 0 3 1 2]	(0, 4, 2, 3, 1,)	[1 3 4 2 0]
[. 2 2]	[1 3 0 4 2]	(0, 1, 3, 4, 2,)	[2 0 4 1 3]
[1 2 2]	[3 0 1 4 2]	(0, 3, 4, 2, 1,)	[1 2 4 0 3]
[. . 3]	[1 2 4 0 3]	(0, 1, 2, 4, 3,)	[3 0 1 4 2]
[1 . 3]	[2 4 1 0 3]	(0, 2, 1, 4, 3,)	[3 2 0 4 1]
[. 1 3]	[4 2 0 1 3]	(0, 4, 3, 1, 2,)	[2 3 1 4 0]
[1 1 3]	[2 0 4 1 3]	(0, 2, 4, 3, 1,)	[1 3 0 4 2]
[. 2 3]	[1 4 0 2 3]	(0, 1, 4, 3, 2,)	[2 0 3 4 1]
[1 2 3]	[4 0 1 2 3]	(0, 4, 3, 2, 1,)	[1 2 3 4 0]

Figure 10.3-H: Numbers in rising factorial basis and corresponding cyclic permutations.

```
    for (ulong k=0; k<n; ++k)  x[k] = k;

    for (ulong k=n-1; k>1; --k)
    {
        ulong z = n-1-k; // 0, ..., n-3
        ulong i = fc[z];
        swap2(x[k], x[i]);
    }

    if ( n>1 )  swap2(x[0], x[1]);
}

void rfact2cyclic(const ulong *fc, ulong n, ulong *x)
// Generate cyclic permutation (standard representation) in x[]
//   from the (n-2) digit factorial number in fc[0,...,n-3].
// Rising radices:  [2, 3, ..., n-1]
{
    for (ulong k=0; k<n; ++k)  x[k] = k;

    for (ulong k=n-1; k>1; --k)
    {
        ulong i = fc[k-2];  // k-2 == n-3, ..., 0
        swap2(x[k], x[i]);
    }

    if ( n>1 )  swap2(x[0], x[1]);
}
```

The cyclic permutations of 5 elements are shown in figures 10.3-G (falling base) and 10.3-H (rising base). The listings were created with the program [FXT: comb/fact2cyclic-demo.cc]. Note that the cycle representation could be obtained by applying the transformations to (all) permutations to all but the first element. That is, one can generate all cyclic permutations in cycle form by permuting all elements but the first with any permutation algorithm.

10.4 An order from reversing prefixes

	permutation	rfact	inv. perm.
0:	[. 1 2 3]	[. . .]	[. 1 2 3]
1:	[1 . 2 3]	[1 . .]	[1 . 2 3]
2:	[2 . 1 3]	[. 1 .]	[1 2 . 3]
3:	[. 2 1 3]	[1 1 .]	[. 2 1 3]
4:	[1 2 . 3]	[. 2 .]	[2 . 1 3]
5:	[2 1 . 3]	[1 2 .]	[2 1 . 3]
6:	[3 . 1 2]	[. . 1]	[1 2 3 .]
7:	[. 3 1 2]	[1 . 1]	[. 2 3 1]
8:	[1 3 . 2]	[. 1 1]	[2 . 3 1]
9:	[3 1 . 2]	[1 1 1]	[2 1 3 .]
10:	[. 1 3 2]	[. 2 1]	[. 1 3 2]
11:	[1 . 3 2]	[1 2 1]	[1 . 3 2]
12:	[2 3 . 1]	[. . 2]	[2 3 . 1]
13:	[3 2 . 1]	[1 . 2]	[2 3 1 .]
14:	[. 2 3 1]	[. 1 2]	[. 3 1 2]
15:	[2 . 3 1]	[1 1 2]	[1 3 . 2]
16:	[3 . 2 1]	[. 2 2]	[1 3 2 .]
17:	[. 3 2 1]	[1 2 2]	[. 3 2 1]
18:	[1 2 3 .]	[. . 3]	[3 . 1 2]
19:	[2 1 3 .]	[1 . 3]	[3 1 . 2]
20:	[3 1 2 .]	[. 1 3]	[3 1 2 .]
21:	[1 3 2 .]	[1 1 3]	[3 . 2 1]
22:	[2 3 1 .]	[. 2 3]	[3 2 . 1]
23:	[3 2 1 .]	[1 2 3]	[3 2 1 .]

Figure 10.4-A: All permutations of 4 elements in an order where the first $j+1$ elements are reversed when the first j digits change in the mixed radix counting sequence with radices [2, 3, 4, ...].

A surprisingly simple algorithm for the generation of all permutations is obtained by mixed radix counting with the radices [2, 3, 4, ...] (column `digits` in figure 10.4-A). Whenever the first j digits change with

an increment then the permutation is updated by reversing the first $j+1$ elements. As with lex order the first half of the permutations are the complements of the permutations in the second half, now rewrite relation 10.1-1 on page 219 as

$$R \;=\; I\,C\,I \tag{10.4-1}$$

to see that the first half of the inverse permutations are the reversed inverse permutations in the second half. This can (for n even) also be observed from the positions of the largest element in the inverse permutations. An implementation is [FXT: `class perm_rev` in comb/perm-rev.h]:

```
class perm_rev
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
    ulong *p_;  // permutation
    ulong n_;   // permutations of n elements
public:
    perm_rev(ulong n)
    {
        n_ = n;
        p_ = new ulong[n_];
        d_ = new ulong[n_];
        d_[n-1] = -1UL;  // sentinel
        first();
    }

    ~perm_rev()
    {
        delete [] p_;
        delete [] d_;
    }

    void first()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
        for (ulong k=0; k<n_; ++k)  p_[k] = k;
    }

    void last()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = k+1;
        for (ulong k=0; k<n_; ++k)  p_[k] = n_-1-k;
    }
```

The update routines are quite concise:

```
    bool next()
    {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 )  { d_[j]=0; ++j; }

        // j==n-1 for last permutation
        if ( j!=n_-1 ) // only if no overflow
        {
            ++d_[j];
            reverse(p_, j+2); // update permutation
            return true;
        }
        else  return false;
    }

    bool prev()
    {
        // decrement mixed radix number:
        ulong j = 0;
        while ( d_[j]==0 )  { d_[j]=j+1; ++j; }

        // j==n-1 for last permutation
        if ( j!=n_-1 ) // only if no overflow
        {
            --d_[j];
            reverse(p_, j+2); // update permutation
            return true;
        }
        else  return false;
```

```
    }
};
```

Note that the routines work for arbitrary (distinct) entries of the array `p_[]`.

An upper bound for the average number of elements that are moved in the transitions when generating all $N = n!$ permutations is $e \approx 2.7182818$ so the algorithm is CAT. The implementation is actually fast, it generates more than 110 million permutations per second. Usage of the class is as simple as:

```
ulong n = 4;  // Number of elements to permute
perm_rev P(n);
P.first();
do
{
    // Use permutation here
}
while ( P.next() );
```

Figure 10.4-A was produced with [FXT: comb/perm-rev-demo.cc]. The described method is given in [250].

10.4.1 Optimizing the update routine

One can optimize the update routine by observing that 5 out of six updates are the swaps

```
(0,1)  (0,2)  (0,1)  (0,2)  (0,1)
```

We use a counter `ct_` and modify the methods `first()` and `next()` [FXT: `class perm_rev2` in comb/perm-rev2.h]:

```
class perm_rev2
{
  [--snip--]
    void first()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
        for (ulong k=0; k<n_; ++k)  p_[k] = k;
        ct_ = 5;
    }

    bool next()
    {
        if ( ct_!=0 )  // easy case(s)
        {
            --ct_;
            swap2(p_[0], p_[1 + (ct_ & 1)]);
            return true;
        }
        else  // increment mixed radix number:
        {
            ct_ = 5;  // reset counter

            ulong j = 2;  // note: start with 2
            while ( d_[j]==j+1 )  { d_[j]=0; ++j; }

            // j==n-1 for last permutation
            if ( j!=n_-1 ) // only if no overflow
            {
                ++d_[j];
                reverse(p_, j+2); // update permutation
                return true;
            }
            else  return false;
        }
    }
  [--snip--]
```

The speedup is remarkable, about 186 million permutations per second can be generated (about 11.8 cycles per update). If arrays are used instead of pointers, the rate is about 200 million per second (10.8 cycles per update). The routine can be used for permutations of at least three elements [FXT: comb/perm-rev2-demo.cc].

10.4.2 Method for unranking

Conversion of a mixed radix (rising factorial) number into the corresponding permutation proceeds as exemplified for the 16-th permutation ($15 = 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 6$, so `d=[1,1,2]`):

```
 1:   p=[ 0, 1, 2, 3 ]   d=[ 0, 0, 0 ] // start
13:   p=[ 2, 3, 0, 1 ]   d=[ 0, 0, 2 ] // right rotate all elements twice
15:   p=[ 0, 2, 3, 1 ]   d=[ 0, 1, 2 ] // right rotate first three elements
16:   p=[ 2, 0, 3, 1 ]   d=[ 1, 1, 2 ] // right rotate first two elements
```

The idea can be implemented as

```
void goto_rfact(const ulong *d)
// Goto permutation corresponding to d[] (i.e. unrank d[]).
// d[] must be a valid (rising) factorial mixed radix string:
// d[]==[d(0), d(1), d(2), ..., d(n-2)]  (n-1 elements) where 0<=d(j)<=j+1
{
    for (ulong k=0; k<n_; ++k)  p_[k] = k;
    for (ulong k=0; k<n_-1; ++k)  d_[k] = d[k];
    for (long j=n_-2; j>=0; --j)  rotate_right(p_, j+2, d_[j]);
}
```

10.5 Minimal-change order (Heap's algorithm)

	permutation	swap	digits	rfact(perm)	inv. perm.
0:	[. 1 2 3]	(0, 0)	[. . .]	[. . .]	[. 1 2 3]
1:	[1 . 2 3]	(1, 0)	[1 . .]	[1 . .]	[1 . 2 3]
2:	[2 . 1 3]	(2, 0)	[. 1 .]	[1 1 .]	[1 2 . 3]
3:	[. 2 1 3]	(1, 0)	[1 1 .]	[. 1 .]	[. 2 1 3]
4:	[1 2 . 3]	(2, 0)	[. 2 .]	[. 2 .]	[2 . 1 3]
5:	[2 1 . 3]	(1, 0)	[1 2 .]	[1 2 .]	[2 1 . 3]
6:	[3 1 . 2]	(3, 0)	[. . 1]	[1 2 1]	[2 1 3 .]
7:	[1 3 . 2]	(1, 0)	[1 . 1]	[. 2 1]	[2 . 3 1]
8:	[. 3 1 2]	(2, 0)	[. 1 1]	[. 1 1]	[. 2 3 1]
9:	[3 . 1 2]	(1, 0)	[1 1 1]	[1 1 1]	[1 2 3 .]
10:	[1 . 3 2]	(2, 0)	[. 2 1]	[1 . 1]	[1 . 3 2]
11:	[. 1 3 2]	(1, 0)	[1 2 1]	[. . 1]	[. 1 3 2]
12:	[. 2 3 1]	(3, 1)	[. . 2]	[. . 2]	[. 3 1 2]
13:	[2 . 3 1]	(1, 0)	[1 . 2]	[1 . 2]	[1 3 . 2]
14:	[3 . 2 1]	(2, 0)	[. 1 2]	[1 1 2]	[1 3 2 .]
15:	[. 3 2 1]	(1, 0)	[1 1 2]	[. 1 2]	[. 3 2 1]
16:	[2 3 . 1]	(2, 0)	[. 2 2]	[. 2 2]	[2 3 . 1]
17:	[3 2 . 1]	(1, 0)	[1 2 2]	[1 2 2]	[2 3 1 .]
18:	[3 2 1 .]	(3, 2)	[. . 3]	[1 2 3]	[3 2 1 .]
19:	[2 3 1 .]	(1, 0)	[1 . 3]	[. 2 3]	[3 2 . 1]
20:	[1 3 2 .]	(2, 0)	[. 1 3]	[. 1 3]	[3 . 2 1]
21:	[3 1 2 .]	(1, 0)	[1 1 3]	[1 1 3]	[3 1 2 .]
22:	[2 1 3 .]	(2, 0)	[. 2 3]	[1 . 3]	[3 1 . 2]
23:	[1 2 3 .]	(1, 0)	[1 2 3]	[. . 3]	[3 . 1 2]

Figure 10.5-A: The permutations of 4 elements in a minimal-change order. Dots denote zeros.

Figure 10.5-A shows the permutations of 4 elements in a *minimal-change order*: just two elements are swapped with each update. The column labeled `digits` shows the mixed radix numbers (rising factorial base, see section 10.3 on page 222) in counting order. Let j be the position of the rightmost change of the mixed radix string R. Then the swap is $(j+1, x)$ where $x = 0$ if j is odd, and $x = R_j - 1$ if j is even. The sequence of values $j + 1$ starts

```
1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 3, 1, 2, 1, 2, 1, 4, 1, 2, 1, ...
```

The n-th value (starting with $n = 1$) is the largest z such that $z!$ divides n (entry A055881 of [214]). The column labeled `rfact(perm)` of the figure shows the rising factorial representations of the permutation, see section 10.8 on page 244. The column is a Gray code only for permutations of up to four elements.

An implementation of the algorithm (given in [131]) is [FXT: `class perm_heap` in comb/perm-heap.h]:

```
class perm_heap
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
    ulong *p_;  // permutation
    ulong n_;   // permutations of n elements
    ulong sw1_, sw2_;   // indices of swapped elements
  [--snip--]
```

The computation of the successor is simple:

```
    bool next()
    {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 )  { d_[j]=0; ++j; }

        // j==n-1 for last permutation
        if ( j!=n_-1 ) // only if no overflow
        {
            ulong k = j+1;
            ulong x = ( (k&1)  ? d_[j] : 0);
            swap2(p_[k], p_[x]);
            sw1_ = k;  sw2_ = x;

            ++d_[j];
            return true;
        }
        else  return false;
    }
  [--snip--]
```

About 115 million permutations are generated per second. Often one will only use the indices of the swapped elements to update the visited configurations:

```
    void get_swap(ulong &s1, ulong &s2)  const  { s1=sw1_; s2=sw2_; }
```

Then the statement `swap2(p_[k], p_[x]);` in the update routine can be omitted which leads to a rate of 165 million permutations per second. Figure 10.5-A shows the permutations of 4 elements, it was created with the program [FXT: comb/perm-heap-demo.cc].

10.5.1 Optimized implementation

The algorithm can be optimized by treating 5 out of 6 cases separately, those where the first or second digit in the mixed radix number changes. We use a counter `ct_` that is decremented [FXT: `class perm_heap2` in comb/perm-heap2.h]:

```
class perm_heap2
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, 5, ..., n-1, (sentinel=-1)]
    ulong *p_;  // permutation
    ulong n_;   // permutations of n elements
    ulong sw1_, sw2_;   // indices of swapped elements
    ulong ct_;  // count 5,4,3,2,1,(0); nonzero ==> easy cases
 [--snip--]
```

The counter is set to 5 in the method `first()`. The update routine is

```
    bool next()
    {
        if ( ct_!=0 )  // easy case(s)
        {
            --ct_;
            sw1_ = 1 + (ct_ & 1);  // == 1,2,1,2,1
            sw2_ = 0;
            swap2(p_[sw1_], p_[sw2_]);
            return true;
        }
        else
        {
            ct_ = 5;  // reset counter

            // increment mixed radix number:
```

```
            ulong j = 2;
            while ( d_[j]==j+1 )  { d_[j]=0; ++j; }
            // j==n-1 for last permutation
            if ( j!=n_-1 ) // only if no overflow
            {
                ulong k = j+1;
                ulong x = ( (k&1)  ? d_[j] : 0);
                sw1_ = k;  sw2_ = x;
                swap2(p_[sw1_], p_[sw2_]);
                ++d_[j];
                return true;
            }
            else  return false;
        }
    }
```

Note that the routine only works for permutations of at least three elements. Usage of the class is shown in [FXT: comb/perm-heap2-demo.cc]. The rate is about 190 million updates per second.

10.5.2 Computing just the swaps

If only the swaps are of interest we can simply omit all statements involving the permutation array `p_[]`. The implementation is [FXT: `class perm_heap2_swaps` in comb/perm-heap2-swaps.h], usage of the class is shown in [FXT: comb/perm-heap2-swaps-demo.cc]:

```
ulong n = 4;
ulong *x = new ulong[n];  // permutations
for (ulong k=0; k<n; ++k)  x[k] = k;
perm_heap2_swaps P(n);
do
{
    ulong sw1, sw2;
    P.get_swap(sw1, sw2);
    swap2( x[sw1], x[sw2] );  // update permutation
    // visit permutation
}
while ( P.next() );
```

The update routine works at a rate about 310 million per second (about 7 CPU cycles per update). Using arrays instead of pointers results in a rate of about 420 million per second (5.25 cycles per update).

Heap's algorithm and the optimization idea was taken from the excellent survey [211] which gives several permutation algorithms and implementations in pseudo code.

10.6 Lipski's Minimal-change orders

10.6.1 Variants of Heap's algorithm

Various algorithms similar to Heap's method are given in Lipski's paper [171], we take three of those and add a similar one. The four orderings obtained for the permutations of five elements are shown in figure 10.6-A. The leftmost order is Heap's order. The implementation is given in [FXT: `class perm_gray_lipski` in comb/perm-gray-lipski.h], the variable `r` determines the order that is generated:

```
class perm_gray_lipski
{
  [--snip--]
    ulong r_;   // order (0<=r<4):
  [--snip--]
    bool next()
    {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 )  { d_[j]=0; ++j; }
        if ( j<n_-1 ) // only if no overflow
```

```
     x=(j&1 ? 0 : d);        x=(j&1 ? 0 : j-d);      x=(j&1 ? j-1 : d);      x=(j&1 ? j-1 : j-d);
  1:  [ . 1 2 3 4 ]           [ . 1 2 3 4 ]           [ . 1 2 3 4 ]           [ . 1 2 3 4 ]
  2:  [ 1 . 2 3 4 ] (1)       [ 1 . 2 3 4 ] (1)       [ 1 . 2 3 4 ] (1)       [ 1 . 2 3 4 ] (1)
  3:  [ 2 . 1 3 4 ] (2)       [ 2 . 1 3 4 ] (2)       [ 2 . 1 3 4 ] (2)       [ 2 . 1 3 4 ] (2)
  4:  [ . 2 1 3 4 ] (1)       [ . 2 1 3 4 ] (1)       [ . 2 1 3 4 ] (1)       [ . 2 1 3 4 ] (1)
  5:  [ 1 2 . 3 4 ] (2)       [ 1 2 . 3 4 ] (2)       [ 1 2 . 3 4 ] (2)       [ 1 2 . 3 4 ] (2)
  6:  [ 2 1 . 3 4 ] (1)       [ 2 1 . 3 4 ] (1)       [ 2 1 . 3 4 ] (1)       [ 2 1 . 3 4 ] (1)
  7:  [ 3 1 . 2 4 ] (3)       [ 2 1 3 . 4 ] (3,2)     [ 3 1 . 2 4 ] (3)       [ 2 1 3 . 4 ] (3,2)
  8:  [ 1 3 . 2 4 ] (1)       [ 1 2 3 . 4 ] (1)       [ 1 3 . 2 4 ] (1)       [ 1 2 3 . 4 ] (1)
  9:  [ . 3 1 2 4 ] (2)       [ 3 2 1 . 4 ] (2)       [ . 3 1 2 4 ] (2)       [ 3 2 1 . 4 ] (2)
 10:  [ 3 . 1 2 4 ] (1)       [ 2 3 1 . 4 ] (1)       [ 3 . 1 2 4 ] (1)       [ 2 3 1 . 4 ] (1)
 11:  [ 1 . 3 2 4 ] (2)       [ 1 3 2 . 4 ] (2)       [ 1 . 3 2 4 ] (2)       [ 1 3 2 . 4 ] (2)
 12:  [ . 1 3 2 4 ] (1)       [ 3 1 2 . 4 ] (1)       [ . 1 3 2 4 ] (1)       [ 3 1 2 . 4 ] (1)
 13:  [ . 2 3 1 4 ] (3,1)     [ 3 . 2 1 4 ] (3,1)     [ . 2 3 1 4 ] (3,1)     [ 3 . 2 1 4 ] (3,1)
 14:  [ 2 . 3 1 4 ] (1)       [ . 3 2 1 4 ] (1)       [ 2 . 3 1 4 ] (1)       [ . 3 2 1 4 ] (1)
 15:  [ 3 . 2 1 4 ] (2)       [ 2 3 . 1 4 ] (2)       [ 3 . 2 1 4 ] (2)       [ 2 3 . 1 4 ] (2)
 16:  [ . 3 2 1 4 ] (1)       [ 3 2 . 1 4 ] (1)       [ . 3 2 1 4 ] (1)       [ 3 2 . 1 4 ] (1)
 17:  [ 2 3 . 1 4 ] (2)       [ . 2 3 1 4 ] (2)       [ 2 3 . 1 4 ] (2)       [ . 2 3 1 4 ] (2)
 18:  [ 3 2 . 1 4 ] (1)       [ 2 . 3 1 4 ] (1)       [ 3 2 . 1 4 ] (1)       [ 2 . 3 1 4 ] (1)
 19:  [ 3 2 1 . 4 ] (3,2)     [ 1 . 3 2 4 ] (3)       [ 3 2 1 . 4 ] (3,2)     [ 1 . 3 2 4 ] (3)
 20:  [ 2 3 1 . 4 ] (1)       [ . 1 3 2 4 ] (1)       [ 2 3 1 . 4 ] (1)       [ . 1 3 2 4 ] (1)
 21:  [ 1 3 2 . 4 ] (2)       [ 3 1 . 2 4 ] (2)       [ 1 3 2 . 4 ] (2)       [ 3 1 . 2 4 ] (2)
 22:  [ 3 1 2 . 4 ] (1)       [ 1 3 . 2 4 ] (1)       [ 3 1 2 . 4 ] (1)       [ 1 3 . 2 4 ] (1)
 23:  [ 2 1 3 . 4 ] (2)       [ . 3 1 2 4 ] (2)       [ 2 1 3 . 4 ] (2)       [ . 3 1 2 4 ] (2)
 24:  [ 1 2 3 . 4 ] (1)       [ 3 . 1 2 4 ] (1)       [ 1 2 3 . 4 ] (1)       [ 3 . 1 2 4 ] (1)
 25:  [ 4 2 3 . 1 ] (4)       [ 4 . 1 2 3 ] (4)       [ 1 2 4 . 3 ] (4,2)     [ 3 . 4 2 1 ] (4,2)
 26:  [ 2 4 3 . 1 ] (1)       [ . 4 1 2 3 ] (1)       [ 2 1 4 . 3 ] (1)       [ . 3 4 2 1 ] (1)
 27:  [ 3 4 2 . 1 ] (2)       [ 1 4 . 2 3 ] (2)       [ 4 1 2 . 3 ] (2)       [ 4 3 . 2 1 ] (2)
 28:  [ 4 3 2 . 1 ] (1)       [ 4 1 . 2 3 ] (1)       [ 1 4 2 . 3 ] (1)       [ 3 4 . 2 1 ] (1)
 29:  [ 2 3 4 . 1 ] (2)       [ . 1 4 2 3 ] (2)       [ 2 4 1 . 3 ] (2)       [ . 4 3 2 1 ] (2)
 30:  [ 3 2 4 . 1 ] (1)       [ 1 . 4 2 3 ] (1)       [ 4 2 1 . 3 ] (1)       [ 4 . 3 2 1 ] (1)
 31:  [ . 2 4 3 1 ] (3)       [ 1 . 2 4 3 ] (3,2)     [ . 2 1 4 3 ] (3)       [ 4 . 2 3 1 ] (3,2)
 32:  [ 2 . 4 3 1 ] (1)       [ . 1 2 4 3 ] (1)       [ 2 . 1 4 3 ] (1)       [ . 4 2 3 1 ] (1)
 33:  [ 4 . 2 3 1 ] (2)       [ 2 1 . 4 3 ] (2)       [ 1 . 2 4 3 ] (2)       [ 2 4 . 3 1 ] (2)
 34:  [ . 4 2 3 1 ] (1)       [ 1 2 . 4 3 ] (1)       [ . 1 2 4 3 ] (1)       [ 4 2 . 3 1 ] (1)
 35:  [ 2 4 . 3 1 ] (2)       [ . 2 1 4 3 ] (2)       [ 2 1 . 4 3 ] (2)       [ . 2 4 3 1 ] (2)
 36:  [ 4 2 . 3 1 ] (1)       [ 2 . 1 4 3 ] (1)       [ 1 2 . 4 3 ] (1)       [ 2 . 4 3 1 ] (1)
 37:  [ 4 3 . 2 1 ] (3,1)     [ 2 4 1 . 3 ] (3,1)     [ 1 4 . 2 3 ] (3,1)     [ 2 3 4 . 1 ] (3,1)
 38:  [ 3 4 . 2 1 ] (1)       [ 4 2 1 . 3 ] (1)       [ 4 1 . 2 3 ] (1)       [ 3 2 4 . 1 ] (1)
 39:  [ . 4 3 2 1 ] (2)       [ 1 2 4 . 3 ] (2)       [ . 1 4 2 3 ] (2)       [ 4 2 3 . 1 ] (2)
 40:  [ 4 . 3 2 1 ] (1)       [ 2 1 4 . 3 ] (1)       [ 1 . 4 2 3 ] (1)       [ 2 4 3 . 1 ] (1)
 41:  [ 3 . 4 2 1 ] (2)       [ 4 1 2 . 3 ] (2)       [ 4 . 1 2 3 ] (2)       [ 3 4 2 . 1 ] (2)
 42:  [ . 3 4 2 1 ] (1)       [ 1 4 2 . 3 ] (1)       [ . 4 1 2 3 ] (1)       [ 4 3 2 . 1 ] (1)
 43:  [ . 3 2 4 1 ] (3,2)     [ . 4 2 1 3 ] (3)       [ . 4 2 1 3 ] (3,2)     [ . 3 2 4 1 ] (3)
 44:  [ 3 . 2 4 1 ] (1)       [ 4 . 2 1 3 ] (1)       [ 4 . 2 1 3 ] (1)       [ 3 . 2 4 1 ] (1)
 45:  [ 2 . 3 4 1 ] (2)       [ 2 . 4 1 3 ] (2)       [ 2 . 4 1 3 ] (2)       [ 2 . 3 4 1 ] (2)
 46:  [ . 2 3 4 1 ] (1)       [ . 2 4 1 3 ] (1)       [ . 2 4 1 3 ] (1)       [ . 2 3 4 1 ] (1)
 47:  [ 3 2 . 4 1 ] (2)       [ 4 2 . 1 3 ] (2)       [ 4 2 . 1 3 ] (2)       [ 3 2 . 4 1 ] (2)
 48:  [ 2 3 . 4 1 ] (1)       [ 2 4 . 1 3 ] (1)       [ 2 4 . 1 3 ] (1)       [ 2 3 . 4 1 ] (1)
 49:  [ 1 3 . 4 2 ] (4)       [ 3 4 . 1 2 ] (4)       [ 2 4 3 1 . ] (4,2)     [ 2 3 1 4 . ] (4,2)
 50:  [ 3 1 . 4 2 ] (1)       [ 4 3 . 1 2 ] (1)       [ 4 2 3 1 . ] (1)       [ 3 2 1 4 . ] (1)
 51:  [ . 1 3 4 2 ] (2)       [ . 3 4 1 2 ] (2)       [ 3 2 4 1 . ] (2)       [ 1 2 3 4 . ] (2)
 52:  [ 1 . 3 4 2 ] (1)       [ 3 . 4 1 2 ] (1)       [ 2 3 4 1 . ] (1)       [ 2 1 3 4 . ] (1)
 53:  [ 3 . 1 4 2 ] (2)       [ 4 . 3 1 2 ] (2)       [ 4 3 2 1 . ] (2)       [ 3 1 2 4 . ] (2)
 54:  [ . 3 1 4 2 ] (1)       [ . 4 3 1 2 ] (1)       [ 3 4 2 1 . ] (1)       [ 1 3 2 4 . ] (1)
 55:  [ 4 3 1 . 2 ] (3)       [ . 4 1 3 2 ] (3,2)     [ 1 4 2 3 . ] (3)       [ 1 3 4 2 . ] (3,2)
 56:  [ 3 4 1 . 2 ] (1)       [ 4 . 1 3 2 ] (1)       [ 4 1 2 3 . ] (1)       [ 3 1 4 2 . ] (1)
 57:  [ 1 4 3 . 2 ] (2)       [ 1 . 4 3 2 ] (2)       [ 2 1 4 3 . ] (2)       [ 4 1 3 2 . ] (2)
 58:  [ 4 1 3 . 2 ] (1)       [ . 1 4 3 2 ] (1)       [ 1 2 4 3 . ] (1)       [ 1 4 3 2 . ] (1)
 59:  [ 3 1 4 . 2 ] (2)       [ 4 1 . 3 2 ] (2)       [ 4 2 1 3 . ] (2)       [ 3 4 1 2 . ] (2)
 60:  [ 1 3 4 . 2 ] (1)       [ 1 4 . 3 2 ] (1)       [ 2 4 1 3 . ] (1)       [ 4 3 1 2 . ] (1)

108:  [ 3 4 2 1 . ] (1)       [ 4 2 3 1 . ] (1)       [ 3 . 4 1 2 ] (1)       [ . 4 3 1 2 ] (1)
109:  [ 3 1 2 4 . ] (3,1)     [ 4 1 3 2 . ] (3,1)     [ 3 1 4 . 2 ] (3,1)     [ . 1 3 4 2 ] (3,1)
110:  [ 1 3 2 4 . ] (1)       [ 1 4 3 2 . ] (1)       [ 1 3 4 . 2 ] (1)       [ 1 . 3 4 2 ] (1)
111:  [ 2 3 1 4 . ] (2)       [ 3 4 1 2 . ] (2)       [ 4 3 1 . 2 ] (2)       [ 3 . 1 4 2 ] (2)
112:  [ 3 2 1 4 . ] (1)       [ 4 3 1 2 . ] (1)       [ 3 4 1 . 2 ] (1)       [ . 3 1 4 2 ] (1)
113:  [ 1 2 3 4 . ] (2)       [ 1 3 4 2 . ] (2)       [ 1 4 3 . 2 ] (2)       [ 1 3 . 4 2 ] (2)
114:  [ 2 1 3 4 . ] (1)       [ 3 1 4 2 . ] (1)       [ 4 1 3 . 2 ] (1)       [ 3 1 . 4 2 ] (1)
115:  [ 2 1 4 3 . ] (3,2)     [ 2 1 4 3 . ] (3)       [ 4 1 . 3 2 ] (3,2)     [ 4 1 . 3 2 ] (3)
116:  [ 1 2 4 3 . ] (1)       [ 1 2 4 3 . ] (1)       [ 1 4 . 3 2 ] (1)       [ 1 4 . 3 2 ] (1)
117:  [ 4 2 1 3 . ] (2)       [ 4 2 1 3 . ] (2)       [ . 4 1 3 2 ] (2)       [ . 4 1 3 2 ] (2)
118:  [ 2 4 1 3 . ] (1)       [ 2 4 1 3 . ] (1)       [ 4 . 1 3 2 ] (1)       [ 4 . 1 3 2 ] (1)
119:  [ 1 4 2 3 . ] (2)       [ 1 4 2 3 . ] (2)       [ 1 . 4 3 2 ] (2)       [ 1 . 4 3 2 ] (2)
120:  [ 4 1 2 3 . ] (1)       [ 4 1 2 3 . ] (1)       [ . 1 4 3 2 ] (1)       [ . 1 4 3 2 ] (1)
```

Figure 10.6-A: First half and last permutations of five elements as obtained by variants of Heap's method. Next to the permutations the swaps are shown as (x, y), a swap $(x, 0)$ is given as (x).

```
      {
          const ulong d = d_[j];
          ulong x;
          switch ( r_ )
          {
          case 0:  x = (j&1 ? 0 : d);  break;     // Lipski(9) == Heap
          case 1:  x = (j&1 ? 0 : j-d);  break;   // Lipski(16)
          case 2:  x = (j&1 ? j-1 : d);  break;   // Lipski(10)
          default: x = (j&1 ? j-1 : j-d);  break; // not in Lipski's paper
          }
          const ulong k = j+1;
          swap2(p_[k], p_[x]);
          sw1_ = k;  sw2_ = x;
          d_[j] = d + 1;
          return true;
      }
      else  return  false;  // j==n-1 for last permutation
    }
  [--snip--]
};
```

The top lines in figure 10.6-A repeat the statements in the switch-block. For three or less elements all orderings coincide, with $n = 4$ elements the orderings for $r = 0$ and $r = 2$, and the orderings for $r = 1$ and $r = 3$ coincide. About 110 million permutations per second are generated [FXT: comb/perm-gray-lipski-demo.cc]. Optimizations similar to those for Heaps method should be obvious.

10.6.2 Variants of Wells' algorithm

```
x=( (j&1) || (d<=1) ? j : j-d);          x=( (j&1) || (d==0) ? 0 : d-1);
   1:    [ . 1 2 3 ]                        1:    [ . 1 2 3 ]
   2:    [ 1 . 2 3 ]  (1, 0)                2:    [ 1 . 2 3 ]  (1, 0)
   3:    [ 1 2 . 3 ]  (2, 1)                3:    [ 2 . 1 3 ]  (2, 0)
   4:    [ 2 1 . 3 ]  (1, 0)                4:    [ . 2 1 3 ]  (1, 0)
   5:    [ 2 . 1 3 ]  (2, 1)                5:    [ 1 2 . 3 ]  (2, 0)
   6:    [ . 2 1 3 ]  (1, 0)                6:    [ 2 1 . 3 ]  (1, 0)
   7:    [ . 2 3 1 ]  (3, 2)                7:    [ 3 1 . 2 ]  (3, 0)
   8:    [ 2 . 3 1 ]  (1, 0)                8:    [ 1 3 . 2 ]  (1, 0)
   9:    [ 2 3 . 1 ]  (2, 1)                9:    [ . 3 1 2 ]  (2, 0)
  10:    [ 3 2 . 1 ]  (1, 0)               10:    [ 3 . 1 2 ]  (1, 0)
  11:    [ 3 . 2 1 ]  (2, 1)               11:    [ 1 . 3 2 ]  (2, 0)
  12:    [ . 3 2 1 ]  (1, 0)               12:    [ . 1 3 2 ]  (1, 0)
  13:    [ . 3 1 2 ]  (3, 2)               13:    [ 2 1 3 . ]  (3, 0)
  14:    [ 3 . 1 2 ]  (1, 0)               14:    [ 1 2 3 . ]  (1, 0)
  15:    [ 3 1 . 2 ]  (2, 1)               15:    [ 3 2 1 . ]  (2, 0)
  16:    [ 1 3 . 2 ]  (1, 0)               16:    [ 2 3 1 . ]  (1, 0)
  17:    [ 1 . 3 2 ]  (2, 1)               17:    [ 1 3 2 . ]  (2, 0)
  18:    [ . 1 3 2 ]  (1, 0)               18:    [ 3 1 2 . ]  (1, 0)
  19:    [ 2 1 3 . ]  (3, 0)               19:    [ 3 . 2 1 ]  (3, 1)
  20:    [ 1 2 3 . ]  (1, 0)               20:    [ . 3 2 1 ]  (1, 0)
  21:    [ 1 3 2 . ]  (2, 1)               21:    [ 2 3 . 1 ]  (2, 0)
  22:    [ 3 1 2 . ]  (1, 0)               22:    [ 3 2 . 1 ]  (1, 0)
  23:    [ 3 2 1 . ]  (2, 1)               23:    [ . 2 3 1 ]  (2, 0)
  24:    [ 2 3 1 . ]  (1, 0)               24:    [ 2 . 3 1 ]  (1, 0)
```

Figure 10.6-B: Wells' order for the permutations of four elements (left), and an order where most swaps are with the first position (right). Dots denote the element zero.

A Gray code for permutations given by Wells [241] is shown in the left of figure 10.6-B. The following implementation, following Lipski's paper [171], includes two variants of the algorithm. We just give the crucial assignments in the method that computes the successor [FXT: `class perm_gray_wells` in comb/perm-gray-wells.h]:

```
class perm_gray_wells
{
  [--snip--]
```

```
            switch ( r_ )
            {
            case 1:  x = ( (j&1) || (d==0) ? 0 : d-1);  break;  // Lipski(14)
            case 2:  x = ( (j&1) || (d==0) ? j : d-1);  break;  // Lipski(15)
            default: x = ( (j&1) || (d<=1) ? j : j-d);  break;  // Wells' order == Lipski(8)
            }
  [--snip--]
};
```

Both expressions `(d==0)` can be changed to `(d<=1)` without changing the algorithm. More than 90 million permutations per second are generated [FXT: comb/perm-gray-wells-demo.cc].

10.7 Strong minimal-change order (Trotter's algorithm)

10.7.1 Variant where smallest element moves most often

	permutation	swap	inverse p.	direction
0:	[. 1 2 3]	(3, 2)	[. 1 2 3]	+ + + +
1:	[1 . 2 3]	(0, 1)	[1 . 2 3]	+ + + +
2:	[1 2 . 3]	(1, 2)	[2 . 1 3]	+ + + +
3:	[1 2 3 .]	(2, 3)	[3 . 1 2]	+ + + +
4:	[2 1 3 .]	(0, 1)	[3 1 . 2]	- + + +
5:	[2 1 . 3]	(3, 2)	[2 1 . 3]	- + + +
6:	[2 . 1 3]	(2, 1)	[1 2 . 3]	- + + +
7:	[. 2 1 3]	(1, 0)	[. 2 1 3]	- + + +
8:	[. 2 3 1]	(2, 3)	[. 3 1 2]	+ + + +
9:	[2 . 3 1]	(0, 1)	[1 3 . 2]	+ + + +
10:	[2 3 . 1]	(1, 2)	[2 3 . 1]	+ + + +
11:	[2 3 1 .]	(2, 3)	[3 2 . 1]	+ + + +
12:	[3 2 1 .]	(0, 1)	[3 2 1 .]	- - + +
13:	[3 2 . 1]	(3, 2)	[2 3 1 .]	- - + +
14:	[3 . 2 1]	(2, 1)	[1 3 2 .]	- - + +
15:	[. 3 2 1]	(1, 0)	[. 3 2 1]	- - + +
16:	[. 3 1 2]	(3, 2)	[. 2 3 1]	+ - + +
17:	[3 . 1 2]	(0, 1)	[1 2 3 .]	+ - + +
18:	[3 1 . 2]	(1, 2)	[2 1 3 .]	+ - + +
19:	[3 1 2 .]	(2, 3)	[3 1 2 .]	+ - + +
20:	[1 3 2 .]	(1, 0)	[3 . 2 1]	- - + +
21:	[1 3 . 2]	(3, 2)	[2 . 3 1]	- - + +
22:	[1 . 3 2]	(2, 1)	[1 . 3 2]	- - + +
23:	[. 1 3 2]	(1, 0)	[. 1 3 2]	- - + +

Figure 10.7-A: The permutations of 4 elements in a strong minimal-change order (smallest element moves most often). Dots denote zeros.

Figure 10.7-A shows the permutations of 4 elements in a *strong minimal-change order*: just two elements are swapped with each update and these are adjacent. Note that, in the sequence of the inverse permutations the swapped pair always consists of elements x and $x+1$. Also the first and last permutation differ by an adjacent transposition (of the last two elements). The ordering can be obtained by an interleaving process shown in figure 10.7-B. The first half of the permutations in this order are the reversals of the second half: the relative order of the two smallest elements is changed only with the transition just after the first half and reversal changes the order of these two elements. Mutual permutations lie $n!/2$ positions apart.

A computer program to obtain all permutations in the shown order was given 1962 by H. F. Trotter [230] (see also [142] and [107]). However, the order was already known long before in connection with bell ringing, under the name *Plain Changes*, see [157].

We compute both the permutation and its inverse [FXT: `class perm_trotter` in comb/perm-trotter.h]:

```
class perm_trotter
```

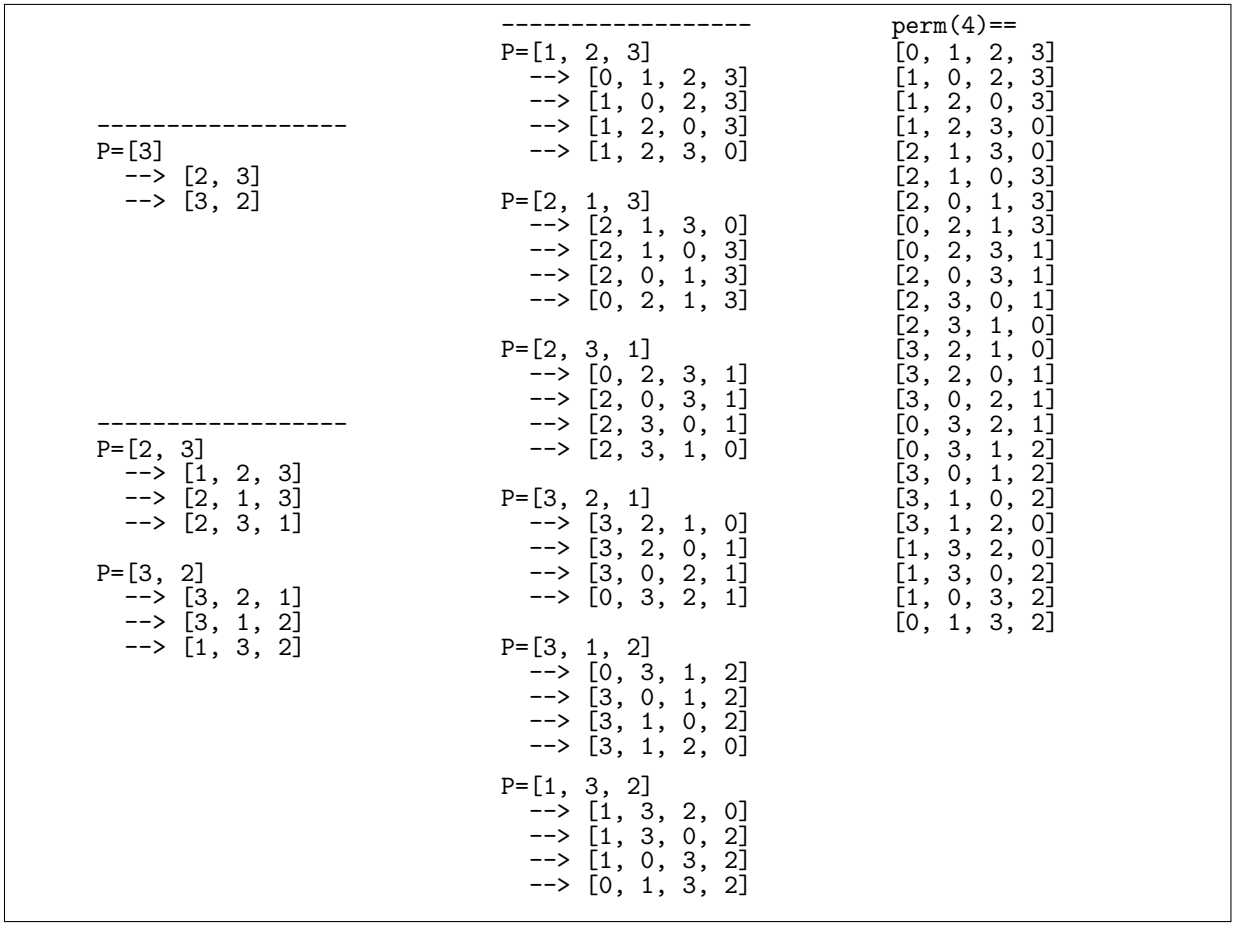


Figure 10.7-B: Trotter's construction as an interleaving process.

```
{
public:
    ulong n_;     // number of elements to permute
    ulong *x_;    // permutation of {0, 1, ..., n-1}
    ulong *xi_;   // inverse permutation
    ulong *d_;    // auxiliary: directions
    ulong sw1_, sw2_; // indices of elements swapped most recently
public:
    perm_trotter(ulong n)
    {
        n_ = n;
        x_ = new ulong[n_+2];
        xi_ = new ulong[n_];
        d_ = new ulong[n_];
        ulong sen = 0;  // sentinel value minimal
        x_[0] = x_[n_+1] = sen;
        ++x_;
        first();
    }
 [--snip--]
```

Note that sentinel elements are at the lower and higher end of the array for the permutation. For each element we store a direction-flag $= \pm 1$ in an array `d_[]`. Initially all are set to $+1$:

```
    void first()
    {
        for (ulong i=0; i<n_; i++)  xi_[i] = i;
        for (ulong i=0; i<n_; i++)  x_[i] = i;
        for (ulong i=0; i<n_; i++)  d_[i] = 1;
        sw1_ = n_ - 1;  sw2_ = n_ - 2;  // relative to last permutation
```

```
    }
 [--snip--]
```

To compute the successor, find the smallest element `e1` whose neighbor `e2` (left or right neighbor, according to the direction) is greater than `e1`. Swap the elements `e1` with `e2` and change the direction of all elements that could not be moved. The location of the elements, `i1` and `i2` are found with the inverse permutation, which has to be updated accordingly:

```
    bool next()
    {
        for (ulong e1=0; e1<n_; ++e1)
        {
            // e1 is the element we try to move
            ulong i1 = xi_[e1];   // position of element e1
            ulong d = d_[e1];     // direction to move e1
            ulong i2 = i1 + d;    // position to swap with
            ulong e2 = x_[i2];    // element to swap with

            if ( e1 < e2 )  // can we swap?
            {
                xi_[e1] = i2;
                xi_[e2] = i1;
                x_[i1] = e2;
                x_[i2] = e1;
                sw1_ = i1;  sw2_ = i2;
                while ( e1-- )  d_[e1] = -d_[e1];
                return true;
            }
        }

        first();
        return false;
    }
```

The locations of the swap can be obtained with the method

```
    void get_swap(ulong &s1, ulong &s2)  const
    { s1=sw1_; s2=sw2_; }
```

The last permutation can be obtained via

```
    void last()
    {
        for (ulong i=0; i<n_; i++)  xi_[i] = i;
        for (ulong i=0; i<n_; i++)  x_[i] = i;
        for (ulong i=0; i<n_; i++)  d_[i] = -1UL;
        sw1_ = n_ - 1;  sw2_ = n_ - 2;  // relative to first permutation
        d_[sw1_] = +1;
        d_[sw2_] = +1;
        swap2(x_[sw1_], x_[sw2_]);
        swap2(xi_[sw1_], xi_[sw2_]);
    }
```

The routine for the predecessor is obtained by adding *one character* to the routine `next()`, it's a minus:

```
    bool next()
    {
  [--snip--]
            ulong d = -d_[e1];     // direction to move e1 (NOTE: negated)
  [--snip--]
        last();
        return false;
    }
```

Well, we also changed the call `first()` to `last()`.

The routines `next()` and `prev()` generate about 137 million permutations per second. Figure 10.7-A was created with the program [FXT: comb/perm-trotter-demo.cc]:

```
    ulong n = 4;
    perm_trotter P(n);
    do
    {
        // visit permutation
    }
    while ( P.next() );
```

10.7.2 Optimized routines

The element zero is moved most often, so we can treat that case separately [FXT: comb/perm-trotter.h]:

```
    bool next()
    {
        {  // most frequent case: e1 == 0
            ulong i1 = xi_[0];   // position of element e1
            ulong d = d_[0];     // direction to move e1
            ulong i2 = i1 + d;    // position to swap with
            ulong e2 = x_[i2];    // element to swap with
            if ( 0 < e2 )  // can we swap?
            {
                xi_[0] = i2;
                xi_[e2] = i1;
                x_[i1] = e2;
                x_[i2] = 0;
                sw1_ = i1;  sw2_ = i2;
                return true;
            }
        }
        for (ulong e1=1; e1<n_; ++e1)  // note: start at e1=1
    [--snip--]
```

The very same modification can be applied to the method `prev()`, only the minus has to be added:

```
            ulong d = -d_[0];     // direction to move e1 (NOTE: negated)
```

Now both methods compute about 174 million permutations per second, corresponding to less than 12.6 CPU cycles per update.

We can also treat the second most frequent case separately by adding the block:

```
    bool next()
    {
        {  // most frequent case: e1 == 0
  [--snip--]
        }
        {  // second most frequent case: e1 == 1
            ulong i1 = xi_[1];   // position of element e1
            ulong d = d_[1];     // direction to move e1
            ulong i2 = i1 + d;    // position to swap with
            ulong e2 = x_[i2];    // element to swap with
            if ( 1 < e2 )  // can we swap?
            {
                xi_[1] = i2;
                xi_[e2] = i1;
                x_[i1] = e2;
                x_[i2] = 1;
                d_[0] = -d_[0];  // negate
                sw1_ = i1;  sw2_ = i2;
                return true;
            }
        }
        for (ulong e1=2; e1<n_; ++e1)  // note: start at e1=2
  [--snip--]
```

With this modification the rate increases to about 179 million per second (12.3 cycles per update).

10.7.3 Variant where largest element moves most often

A variant of the algorithm moves the largest element most often as shown in figure 10.7-C (created with [FXT: comb/perm-trotter-lg-demo.cc]). Only a few modifications have to be made to the code [FXT: `class perm_trotter_lg` in comb/perm-trotter-lg.h]. The sentinel needs to be greater than all elements of the permutations, and the directions start with minus one:

```
class perm_trotter_lg
{
  [--snip--]
public:
```

```
             permutation     swap      inverse p.     direction
      0:    [ . 1 2 3 ]    (0, 1)     [ . 1 2 3 ]     - - - -
      1:    [ . 1 3 2 ]    (3, 2)     [ . 1 3 2 ]     - - - -
      2:    [ . 3 1 2 ]    (2, 1)     [ . 2 3 1 ]     - - - -
      3:    [ 3 . 1 2 ]    (1, 0)     [ 1 2 3 . ]     - - - -
      4:    [ 3 . 2 1 ]    (3, 2)     [ 1 3 2 . ]     - - - +
      5:    [ . 3 2 1 ]    (0, 1)     [ . 3 2 1 ]     - - - +
      6:    [ . 2 3 1 ]    (1, 2)     [ . 3 1 2 ]     - - - +
      7:    [ . 2 1 3 ]    (2, 3)     [ . 2 1 3 ]     - - - +
      8:    [ 2 . 1 3 ]    (1, 0)     [ 1 2 . 3 ]     - - - -
      9:    [ 2 . 3 1 ]    (3, 2)     [ 1 3 . 2 ]     - - - -
     10:    [ 2 3 . 1 ]    (2, 1)     [ 2 3 . 1 ]     - - - -
     11:    [ 3 2 . 1 ]    (1, 0)     [ 2 3 1 . ]     - - - -
     12:    [ 3 2 1 . ]    (3, 2)     [ 3 2 1 . ]     - - + +
     13:    [ 2 3 1 . ]    (0, 1)     [ 3 2 . 1 ]     - - + +
     14:    [ 2 1 3 . ]    (1, 2)     [ 3 1 . 2 ]     - - + +
     15:    [ 2 1 . 3 ]    (2, 3)     [ 2 1 . 3 ]     - - + +
     16:    [ 1 2 . 3 ]    (0, 1)     [ 2 . 1 3 ]     - - + -
     17:    [ 1 2 3 . ]    (3, 2)     [ 3 . 1 2 ]     - - + -
     18:    [ 1 3 2 . ]    (2, 1)     [ 3 . 2 1 ]     - - + -
     19:    [ 3 1 2 . ]    (1, 0)     [ 3 1 2 . ]     - - + -
     20:    [ 3 1 . 2 ]    (2, 3)     [ 2 1 3 . ]     - - + +
     21:    [ 1 3 . 2 ]    (0, 1)     [ 2 . 3 1 ]     - - + +
     22:    [ 1 . 3 2 ]    (1, 2)     [ 1 . 3 2 ]     - - + +
     23:    [ 1 . 2 3 ]    (2, 3)     [ 1 . 2 3 ]     - - + +
```

Figure 10.7-C: The permutations of 4 elements in a strong minimal-change order (largest element moves most often). Dots denote zeros.

```
  perm_trotter_lg(ulong n)
  {
[--snip--]
      ulong sen = n_;  // sentinel value maximal
      x_[0] = x_[n_+1] = sen;
      ++x_;
      first();
  }
[--snip--]
  void first()
  {
      for (ulong i=0; i<n_; i++)  xi_[i] = i;
      for (ulong i=0; i<n_; i++)  x_[i] = i;
      for (ulong i=0; i<n_; i++)  d_[i] = -1UL;
      sw1_ = 0;  sw2_ = 1;  // relative to last permutation
  }
[--snip--]
```

In the update routine we look for the largest element whose neighbor is smaller than itself:

```
  bool next()
  {
      ulong e1 = n_;
      while ( e1-- )
      {
          // e1 is the element we try to move
          ulong i1 = xi_[e1];   // position of element e1
          ulong d = d_[e1];     // direction to move e1
          ulong i2 = i1 + d;    // position to swap with
          ulong e2 = x_[i2];    // element to swap with

          if ( e1 > e2 )  // can we swap?
          {
              xi_[e1] = i2;
              xi_[e2] = i1;
              x_[i1] = e2;
              x_[i2] = e1;
              sw1_ = i1;  sw2_ = i2;
              while ( ++e1<n_ )  d_[e1] = -d_[e1];
              return true;
          }
      }
```

```
        first();
        return false;
    }
```

The last permutation can be obtained via

```
    void last()
    {
        for (ulong i=0; i<n_; i++)  xi_[i] = i;
        for (ulong i=0; i<n_; i++)  x_[i] = i;
        for (ulong i=0; i<n_; i++)  d_[i] = +1;
        sw1_ = 0;  sw2_ = 1;  // relative to first permutation
        d_[sw1_] = -1UL;
        d_[sw2_] = -1UL;
        swap2(x_[sw1_], x_[sw2_]);
        swap2(xi_[sw1_], xi_[sw2_]);
    }
```

The method to compute the predecessor is obtained as before. The routines `next()` and `prev()` generate about 126 million permutations per second.

10.8 Minimal-change orders from factorial numbers

10.8.1 Permutations with falling factorial numbers

	permutation	ffact	pos	dir	inverse perm.
0:	[. 1 2 3]	[. . .]			[. 1 2 3]
1:	[1 . 2 3]	[1 . .]	0	+1	[1 . 2 3]
2:	[1 2 . 3]	[2 . .]	0	+1	[2 . 1 3]
3:	[1 2 3 .]	[3 . .]	0	+1	[3 . 1 2]
4:	[2 1 3 .]	[3 1 .]	1	+1	[3 1 . 2]
5:	[2 1 . 3]	[2 1 .]	0	-1	[2 1 . 3]
6:	[2 . 1 3]	[1 1 .]	0	-1	[1 2 . 3]
7:	[. 2 1 3]	[. 1 .]	0	-1	[. 2 1 3]
8:	[. 2 3 1]	[. 2 .]	1	+1	[. 3 1 2]
9:	[2 . 3 1]	[1 2 .]	0	+1	[1 3 . 2]
10:	[2 3 . 1]	[2 2 .]	0	+1	[2 3 . 1]
11:	[2 3 1 .]	[3 2 .]	0	+1	[3 2 . 1]
12:	[3 2 1 .]	[3 2 1]	2	+1	[3 2 1 .]
13:	[3 2 . 1]	[2 2 1]	0	-1	[2 3 1 .]
14:	[3 . 2 1]	[1 2 1]	0	-1	[1 3 2 .]
15:	[. 3 2 1]	[. 2 1]	0	-1	[. 3 2 1]
16:	[. 3 1 2]	[. 1 1]	1	-1	[. 2 3 1]
17:	[3 . 1 2]	[1 1 1]	0	+1	[1 2 3 .]
18:	[3 1 . 2]	[2 1 1]	0	+1	[2 1 3 .]
19:	[3 1 2 .]	[3 1 1]	0	+1	[3 1 2 .]
20:	[1 3 2 .]	[3 . 1]	1	-1	[3 . 2 1]
21:	[1 3 . 2]	[2 . 1]	0	-1	[2 . 3 1]
22:	[1 . 3 2]	[1 . 1]	0	-1	[1 . 3 2]
23:	[. 1 3 2]	[. . 1]	0	-1	[. 1 3 2]

Figure 10.8-A: Permutations in minimal-change order (left) and Gray code for mixed radix numbers with falling factorial base. The two rightmost columns give the place of change with the mixed radix numbers and its direction. Whenever digit p (=`pos`) changes by $d = \pm 1$ (=`dir`) in the mixed radix sequence the element p of the permutation is swapped with its right ($d = +1$) or left ($d = -1$) neighbor.

The Gray code for the mixed radix numbers with falling factorial base allows the computation of the permutations in minimal-change order in an elegant way. See figure 10.8-A which was created with the program [FXT: comb/perm-gray-ffact2-demo.cc]. The algorithm is implemented in [FXT: `class perm_gray_ffact2` in comb/perm-gray-ffact2.h]:

```
class perm_gray_ffact2
{
public:
```

```
    mixedradix_gray2 *mrg_;  // loopless routine
    ulong n_;   // number of elements to permute
    ulong *x_;  // current permutation (of {0, 1, ..., n-1})
    ulong *ix_;  // inverse permutation
    ulong sw1_, sw2_; // indices of elements swapped most recently
public:
    perm_gray_ffact2(ulong n)
        : n_(n)
    {
        x_ = new ulong[n_];
        ix_ = new ulong[n_];
        mrg_ = new mixedradix_gray2(n_-1, 0);  // falling factorial base
        first();
    }
  [--snip--]
    void first()
    {
        mrg_->first();
        for (ulong k=0; k<n_; ++k)  x_[k] = ix_[k] = k;
        sw1_=n_-1;  sw2_=n_-2;
    }
```

The crucial part is the computation of the successor:

```
    bool next()
    {
        // Compute next mixed radix number in Gray code order:
        if ( false == mrg_->next() )  { first(); return false; }
        const ulong j = mrg_->pos();  // position of changed digit
        const int d = mrg_->dir();    // direction of change

        // swap:
        const ulong x1 = j;        // element j
        const ulong i1 = ix_[x1];  // position of j
        const ulong i2 = i1 + d;   // neighbor
        const ulong x2 = x_[i2];   // position of neighbor
        x_[i1] = x2;   x_[i2] = x1;   // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
        sw1_=i1;  sw2_=i2;
        return true;
    }
```

The class uses the loopless algorithm for the computation of the mixed radix Gray code, so it is loopless itself. An alternative (CAT) algorithm is implemented in [FXT: `class perm_gray_ffact` in comb/perm-gray-ffact.h], we give just the routine for the successor:

```
private:
    void swap(ulong j, ulong im)  // used with next() and prev()
    {
        const ulong x1 = j;        // element j
        const ulong i1 = ix_[x1];  // position of j
        const ulong i2 = i1 + im;  // neighbor
        const ulong x2 = x_[i2];   // position of neighbor
        x_[i1] = x2;   x_[i2] = x1;   // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
        sw1_=i1;  sw2_=i2;
    }
public:
    bool next()
    {
        ulong j = 0;
        ulong m1 = n_ - 1;  // nine in falling factorial base
        ulong ij;
        while ( (ij=i_[j]) )
        {
            ulong im = i_[j];
            ulong dj = d_[j] + im;
            if ( dj>m1 )  // =^= if ( (dj>m1) || ((long)dj<0) )
            {
                i_[j] = -ij;
            }
            else
```

```
            {
                d_[j] = dj;
                swap(j, im);
                return true;
            }
            --m1;
            ++j;
        }
        return false;
    }
```

The routine for the predecessor (`prev()`) is obtained by replacing the statement `ulong im = i_[j];` by `ulong im = -i_[j];`. The loopless routine computes about 80 million permutations per second, the CAT version about 110 million per second [FXT: comb/perm-gray-ffact-demo.cc]. Both are slower than the implementations given in section 10.7 on page 239.

10.8.2 Permutations with rising factorial numbers

	permutation	rfact	pos	dir	inverse perm.
0:	[. 1 2 3]	[. . .]			[. 1 2 3]
1:	[1 . 2 3]	[1 . .]	0	+1	[1 . 2 3]
2:	[2 . 1 3]	[1 1 .]	1	+1	[1 2 . 3]
3:	[. 2 1 3]	[. 1 .]	0	-1	[. 2 1 3]
4:	[1 2 . 3]	[. 2 .]	1	+1	[2 . 1 3]
5:	[2 1 . 3]	[1 2 .]	0	+1	[2 1 . 3]
6:	[3 1 . 2]	[1 2 1]	2	+1	[2 1 3 .]
7:	[1 3 . 2]	[. 2 1]	0	-1	[2 . 3 1]
8:	[. 3 1 2]	[. 1 1]	1	-1	[. 2 3 1]
9:	[3 . 1 2]	[1 1 1]	0	+1	[1 2 3 .]
10:	[1 . 3 2]	[1 . 1]	1	-1	[1 . 3 2]
11:	[. 1 3 2]	[. . 1]	0	-1	[. 1 3 2]
12:	[. 2 3 1]	[. . 2]	2	+1	[. 3 1 2]
13:	[2 . 3 1]	[1 . 2]	0	+1	[1 3 . 2]
14:	[3 . 2 1]	[1 1 2]	1	+1	[1 3 2 .]
15:	[. 3 2 1]	[. 1 2]	0	-1	[. 3 2 1]
16:	[2 3 . 1]	[. 2 2]	1	+1	[2 3 . 1]
17:	[3 2 . 1]	[1 2 2]	0	+1	[2 3 1 .]
18:	[3 2 1 .]	[1 2 3]	2	+1	[3 2 1 .]
19:	[2 3 1 .]	[. 2 3]	0	-1	[3 2 . 1]
20:	[1 3 2 .]	[. 1 3]	1	-1	[3 . 2 1]
21:	[3 1 2 .]	[1 1 3]	0	+1	[3 1 2 .]
22:	[2 1 3 .]	[1 . 3]	1	-1	[3 1 . 2]
23:	[1 2 3 .]	[. . 3]	0	-1	[3 . 1 2]

Figure 10.8-B: Permutations in minimal-change order (left) and Gray code for mixed radix numbers with rising factorial base. For even n the first and last permutations are cyclic shifts by one of each other.

Figure 10.8-B shows a Gray code for permutations based on the Gray code for numbers in rising factorial base. The ordering coincides with Heap's algorithm (see section 10.5 on page 234) for up to four elements. A recursive construction for the order is shown in figure 10.8-C. The figure was created with the program [FXT: comb/perm-gray-rfact-demo.cc] (see also [FXT: comb/fact2perm-demo.cc]). A constant amortized time (CAT) algorithm for generating the permutations is [FXT: `class perm_gray_rfact` in comb/perm-gray-rfact.h]:

```
class perm_gray_rfact
{
public:
    mixedradix_gray *M_;  // loopless routine
    ulong n_;   // number of elements to permute
    ulong *x_;  // current permutation (of {0, 1, ..., n-1})
    ulong *ix_;  // inverse permutation
    ulong sw1_, sw2_; // indices of elements swapped most recently
public:
    perm_gray_rfact(ulong n)
        : n_(n)
    {
```

```
perm(2)=
  01
  10

append 2:
  01 2
  10 2

reverse and swap (2,1)
  20 1
  02 1

reverse and swap (1,0)
  12 0
  21 0

==> perm(3)
  012
  102
  201
  021
  120
  210
```

```
append 3:
  012 3
  102 3
  201 3
  021 3
  120 3
  210 3

reverse and swap (3,2):
  310 2
  130 2
  031 2
  301 2
  103 2
  013 2

reverse and swap (2,1):
  023 1
  203 1
  302 1
  032 1
  230 1
  320 1

reverse and swap (1,0):
  321 0
  231 0
  132 0
  312 0
  213 0
  123 0
```

```
==> perm(4):
  0123
  1023
  2013
  0213
  1203
  2103
  3102
  1302
  0312
  3012
  1032
  0132
  0231
  2031
  3021
  0321
  2301
  3201
  3210
  2310
  1320
  3120
  2130
  1230
```

Figure 10.8-C: Recursive construction of the permutations.

```
        x_ = new ulong[n_];
        ix_ = new ulong[n_];
        M_ = new mixedradix_gray(n_-1, 1);  // rising factorial base
        first();
    }
  [--snip--]
    void first()
    {
        M_->first();
        for (ulong k=0; k<n_; ++k)  x_[k] = ix_[k] = k;
        sw1_=n_-1;  sw2_=n_-2;
    }
```

Let $j \geq 0$ be the position of the digit changed with incrementing the mixed radix number, and $d = \pm 1$ the increment or decrement of that digit. The swap to obtain the successor permutation swaps the element x_1 at position $j+1$ with the element x_2 where x_2 is lying to the left of x_1 and it is the greatest element smaller than x_1 for $d > 0$, and the smallest element greater than x_1 for $d < 0$:

```
    bool next()
    {
        // Compute next mixed radix number in Gray code order:
        if ( false == M_->next() )  { first(); return false; }
        ulong j = M_->pos();  // position of changed digit

        if ( j<=1 )  // easy cases: swap == (0,j+1)
        {
            const ulong i2 = j+1;  // i1 == 0
            const ulong x1 = x_[0],  x2 = x_[i2];
            x_[0] = x2;    x_[i2] = x1;  // swap2(x_[i1], x_[i2]);
            ix_[x1] = i2;  ix_[x2] = 0;  // swap2(ix_[x1], ix_[x2]);
            sw1_=0;  sw2_=i2;
            return true;
        }
        else
        {
            ulong i1 = j+1,  i2 = i1;
            ulong x1 = x_[i1],  x2;
            int d = M_->dir();    // direction of change
            if ( d>0 )
            {
                x2 = 0;
```

```
            for (ulong t=0; t<i1; ++t)  // search maximal smaller element left
            {
                ulong xt = x_[t];
                if ( (xt < x1) && (xt >= x2) )  { i2=t; x2=xt; }
            }
        }
        else
        {
            x2 = n_;
            for (ulong t=0; t<i1; ++t)  // search minimal larger element
            {
                ulong xt = x_[t];
                if ( (xt > x1) && (xt <= x2) )  { i2=t; x2=xt; }
            }
        }
        x_[i1] = x2;    x_[i2] = x1;  // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
        sw1_=i2;  sw2_=i1;
        return true;
    }
}
```

There is a slightly more efficient algorithm to compute the successor using the inverse permutations:

```
  bool next()
  {
[--snip--]  /* easy cases as before */
      else
      {
          ulong i1 = j+1,  i2 = i1;
          ulong x1 = x_[i1],  x2;
          int d = M_->dir();    // direction of change
          if ( d>0 )  // in the inverse permutation search first smaller element left:
          {
              for (x2=x1-1;  ; --x2)  if ( (i2=ix_[x2]) < i1 )  break;
          }
          else  // in the inverse permutation search first smaller element right:
          {
              for (x2=x1+1;  ; ++x2)  if ( (i2=ix_[x2]) < i1 )  break;
          }
[--snip--]  /* swaps as before */
      }
  }
```

The method is chosen by defining `SUCC_BY_INV` in the file [FXT: comb/perm-gray-rfact.h]. About 68 million permutations per second are generated, about 58 million with the first method.

10.8.3 Permutations with permuted factorial numbers

The rising and falling factorial numbers are special cases of factorial numbers with permuted digits. We give a method to compute the Gray code for permutations from the Gray code for permuted (falling) factorial numbers. A permutation of the radices determines how often a digit at any position is changed: the leftmost changes most often, the rightmost least often. The permutations corresponding to the mixed radix numbers with radix vector $[2, 3, 5, 4]$, the falling factorial last two radices swapped, is shown in figure 10.8-D [FXT: comb/perm-gray-rot1-demo.cc]. The desired property of this ordering is that the last permutation is as close to a cyclic shift by one of the first as possible. With even n the Gray code with the falling factorial basis the last permutation is a shift by one. With odd n no such Gray code exists: the total number of transpositions with any Gray code is odd for all $n > 1$, but the cyclic rotation by one corresponds to an even number of transpositions. The best we can get is that the first e elements where $e \le n$ is the greatest possible even number. For example,

```
            first                 last
  n=6:   [ 0 1 2 3 4 5 ]     [ 1 2 3 4 5 0 ]
  n=7:   [ 0 1 2 3 4 5 6 ]   [ 1 2 3 4 5 0 6 ]
```

We use this ordering to show the general method [FXT: `class perm_gray_rot1` in comb/perm-gray-rot1.h]:

```
class perm_gray_rot1
{
public:
```

```
              permutation    swap        xfact      pos dir      inv.perm.
       0:     [ . 1 2 3 4 ]              [ . . . . ]              [ . 1 2 3 4 ]
       1:     [ 1 . 2 3 4 ]  (0, 1)      [ 1 . . . ]   0   +1     [ 1 . 2 3 4 ]
       2:     [ 2 . 1 3 4 ]  (0, 2)      [ 1 1 . . ]   1   +1     [ 1 2 . 3 4 ]
       3:     [ . 2 1 3 4 ]  (0, 1)      [ . 1 . . ]   0   -1     [ . 2 1 3 4 ]
       4:     [ 1 2 . 3 4 ]  (0, 2)      [ . 2 . . ]   1   +1     [ 2 . 1 3 4 ]
       5:     [ 2 1 . 3 4 ]  (0, 1)      [ 1 2 . . ]   0   +1     [ 2 1 . 3 4 ]
       6:     [ 2 1 . 4 3 ]  (3, 4)      [ 1 2 1 . ]   2   +1     [ 2 1 . 4 3 ]
       7:     [ 1 2 . 4 3 ]  (0, 1)      [ . 2 1 . ]   0   -1     [ 2 . 1 4 3 ]
    [--snip--]
      91:     [ 3 4 2 1 . ]  (0, 1)      [ . 2 4 3 ]   0   -1     [ 4 3 2 . 1 ]
      92:     [ 2 4 3 1 . ]  (0, 2)      [ . 1 4 3 ]   1   -1     [ 4 3 . 2 1 ]
      93:     [ 4 2 3 1 . ]  (0, 1)      [ 1 1 4 3 ]   0   +1     [ 4 3 1 2 . ]
      94:     [ 3 2 4 1 . ]  (0, 2)      [ 1 . 4 3 ]   1   -1     [ 4 3 1 . 2 ]
      95:     [ 2 3 4 1 . ]  (0, 1)      [ . . 4 3 ]   0   -1     [ 4 3 . 1 2 ]
      96:     [ 2 3 4 . 1 ]  (3, 4)      [ . . 3 3 ]   2   -1     [ 3 4 . 1 2 ]
      97:     [ 3 2 4 . 1 ]  (0, 1)      [ 1 . 3 3 ]   0   +1     [ 3 4 1 . 2 ]
    [--snip--]
     106:     [ 3 1 4 . 2 ]  (0, 2)      [ 1 . 2 3 ]   1   -1     [ 3 1 4 . 2 ]
     107:     [ 1 3 4 . 2 ]  (0, 1)      [ . . 2 3 ]   0   -1     [ 3 . 4 1 2 ]
     108:     [ 1 2 4 . 3 ]  (1, 4)      [ . . 1 3 ]   2   -1     [ 3 . 1 4 2 ]
     109:     [ 2 1 4 . 3 ]  (0, 1)      [ 1 . 1 3 ]   0   +1     [ 3 1 . 4 2 ]
     110:     [ 4 1 2 . 3 ]  (0, 2)      [ 1 1 1 3 ]   1   +1     [ 3 1 2 4 . ]
     111:     [ 1 4 2 . 3 ]  (0, 1)      [ . 1 1 3 ]   0   -1     [ 3 . 2 4 1 ]
     112:     [ 2 4 1 . 3 ]  (0, 2)      [ . 2 1 3 ]   1   +1     [ 3 2 . 4 1 ]
     113:     [ 4 2 1 . 3 ]  (0, 1)      [ 1 2 1 3 ]   0   +1     [ 3 2 1 4 . ]
     114:     [ 3 2 1 . 4 ]  (0, 4)      [ 1 2 . 3 ]   2   -1     [ 3 2 1 . 4 ]
     115:     [ 2 3 1 . 4 ]  (0, 1)      [ . 2 . 3 ]   0   -1     [ 3 2 . 1 4 ]
     116:     [ 1 3 2 . 4 ]  (0, 2)      [ . 1 . 3 ]   1   -1     [ 3 . 2 1 4 ]
     117:     [ 3 1 2 . 4 ]  (0, 1)      [ 1 1 . 3 ]   0   +1     [ 3 1 2 . 4 ]
     118:     [ 2 1 3 . 4 ]  (0, 2)      [ 1 . . 3 ]   1   -1     [ 3 1 . 2 4 ]
     119:     [ 1 2 3 . 4 ]  (0, 1)      [ . . . 3 ]   0   -1     [ 3 . 1 2 4 ]
```

Figure 10.8-D: Permutations with mixed radix numbers with radix vector $[2, 3, 5, 4]$.

```
    mixedradix_gray *M_;  // Gray code for factorial numbers
    ulong n_;   // number of elements to permute
    ulong *x_;  // current permutation (of {0, 1, ..., n-1})
    ulong *ix_;  // inverse permutation
    ulong sw1_, sw2_; // indices of elements swapped most recently
public:
    perm_gray_rot1(ulong n)
    // Must have: n>=1
    {
        n_ = (n ? n : 1);  // at least one
        x_ = new ulong[n_];
        ix_ = new ulong[n_];

        M_ = new mixedradix_gray(n_-1, 1);  // rising factorial base

        // apply permutation of radix vector with mixed radix number:
        if ( (n_ >= 3) && (n & 1) )  // odd n>=3
        {
            ulong *m1 = M_->m1_;
            swap2(m1[n_-2], m1[n_-3]);  // swap last two factorial nines
        }

        first();
    }
  [--snip--]
```

The permutation applied here can be replaced by any permutation, the following update routines will still work:

```
    bool next()
    {
        // Compute next mixed radix number in Gray code order:
        if ( false == M_->next() )  { first();  return false; }

        const ulong j = M_->pos();  // position of changed digit
        const ulong i1 = M_->m1_[j];  // valid for any permutation of factorial radices
```

```
        const ulong x1 = x_[i1];
        ulong  i2 = i1,  x2;
        const int d = M_->dir();    // direction of change
        if ( d>0 )  // in the inverse permutation search first smaller element left:
        {
            for (x2=x1-1;  ; --x2)  if ( (i2=ix_[x2]) < i1 )  break;
        }
        else  // in the inverse permutation search first smaller element right:
        {
            for (x2=x1+1;  ; ++x2)  if ( (i2=ix_[x2]) < i1 )  break;
        }
        x_[i1] = x2;    x_[i2] = x1;  // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
        sw1_=i2;  sw2_=i1;
        return true;
    }
  [--snip--]
```

Note that instead of taking $j+1$ as the position of the element to move, we take the value of the nine at the position j. The special ordering obtained here can be used to construct a Gray code with the single track property, see section 10.10.2 on page 257.

10.9 Orders where the smallest element always moves right

10.9.1 A variant of Trotter's construction

An ordering for the permutations where the first element always moves right can be obtained by the interleaving process shown in figure 10.9-A. The process is the same as that for Trotter's order shown in figure 10.7-B on page 240, but without changing the directions. The second half of the permutations is the reversed list of the reversed permutations in the first half. The permutations are shown in figure 10.9-B, they are the inverses of the permutations corresponding to the falling factorial numbers, see figure 10.3-A on page 224. An implementation is [FXT: `class perm_mv0` in comb/perm-mv0.h]:

```
class perm_mv0
{
public:
    ulong *d_;  // mixed radix digits with radix = [n-1, n-2, n-3, ..., 2]
    ulong *x_;  // permutation
    ulong ect_; // counter for easy case
    ulong n_;   // permutations of n elements
public:
    perm_mv0(ulong n)
    // Must have n>=2
    {
        n_ = n;
        d_ = new ulong[n_];
        d_[n-1] = 1;  // sentinel (must be nonzero)
        x_ = new ulong[n_];
        first();
    }
  [--snip--]
    void first()
    {
        for (ulong k=0; k<n_; ++k)  x_[k] = k;
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
        ect_ = 0;
    }
  [--snip--]
```

The update process uses the falling factorial numbers. Let j be the position where the digit is incremented, and d the value before the increment. The update

```
      permutation       ffact
                             v-- increment at j=2
    [ 4 2 3 5 1 0 ]    [ 5 4 1 1 . ]  <--= digit before increment is d=1
```

```
------------------
P=[3]
  --> [2, 3]
  --> [3, 2]

------------------
P=[2, 3]
  --> [1, 2, 3]
  --> [2, 1, 3]
  --> [2, 3, 1]

P=[3, 2]
  --> [1, 3, 2]
  --> [3, 1, 2]
  --> [3, 2, 1]
```

```
------------------
P=[1, 2, 3]
  --> [0, 1, 2, 3]
  --> [1, 0, 2, 3]
  --> [1, 2, 0, 3]
  --> [1, 2, 3, 0]

P=[2, 1, 3]
  --> [0, 2, 1, 3]
  --> [2, 0, 1, 3]
  --> [2, 1, 0, 3]
  --> [2, 1, 3, 0]

P=[2, 3, 1]
  --> [0, 2, 3, 1]
  --> [2, 0, 3, 1]
  --> [2, 3, 0, 1]
  --> [2, 3, 1, 0]

P=[1, 3, 2]
  --> [0, 1, 3, 2]
  --> [1, 0, 3, 2]
  --> [1, 3, 0, 2]
  --> [1, 3, 2, 0]

P=[3, 1, 2]
  --> [0, 3, 1, 2]
  --> [3, 0, 1, 2]
  --> [3, 1, 0, 2]
  --> [3, 1, 2, 0]

P=[3, 2, 1]
  --> [0, 3, 2, 1]
  --> [3, 0, 2, 1]
  --> [3, 2, 0, 1]
  --> [3, 2, 1, 0]
```

```
perm(4)==
[0, 1, 2, 3]
[1, 0, 2, 3]
[1, 2, 0, 3]
[1, 2, 3, 0]
[0, 2, 1, 3]
[2, 0, 1, 3]
[2, 1, 0, 3]
[2, 1, 3, 0]
[0, 2, 3, 1]
[2, 0, 3, 1]
[2, 3, 0, 1]
[2, 3, 1, 0]
[0, 1, 3, 2]
[1, 0, 3, 2]
[1, 3, 0, 2]
[1, 3, 2, 0]
[0, 3, 1, 2]
[3, 0, 1, 2]
[3, 1, 0, 2]
[3, 1, 2, 0]
[0, 3, 2, 1]
[3, 0, 2, 1]
[3, 2, 0, 1]
[3, 2, 1, 0]
```

Figure 10.9-A: Interleaving process to obtain all permutations by right moves.

	permutation	ffact	inv. perm.
0:	[. 1 2 3]	[. . .]	[. 1 2 3]
1:	[1 . 2 3]	[1 . .]	[1 . 2 3]
2:	[1 2 . 3]	[2 . .]	[2 . 1 3]
3:	[1 2 3 .]	[3 . .]	[3 . 1 2]
4:	[. 2 1 3]	[. 1 .]	[. 2 1 3]
5:	[2 . 1 3]	[1 1 .]	[1 2 . 3]
6:	[2 1 . 3]	[2 1 .]	[2 1 . 3]
7:	[2 1 3 .]	[3 1 .]	[3 1 . 2]
8:	[. 2 3 1]	[. 2 .]	[. 3 1 2]
9:	[2 . 3 1]	[1 2 .]	[1 3 . 2]
10:	[2 3 . 1]	[2 2 .]	[2 3 . 1]
11:	[2 3 1 .]	[3 2 .]	[3 2 . 1]
12:	[. 1 3 2]	[. . 1]	[. 1 3 2]
13:	[1 . 3 2]	[1 . 1]	[1 . 3 2]
14:	[1 3 . 2]	[2 . 1]	[2 . 3 1]
15:	[1 3 2 .]	[3 . 1]	[3 . 2 1]
16:	[. 3 1 2]	[. 1 1]	[. 2 3 1]
17:	[3 . 1 2]	[1 1 1]	[1 2 3 .]
18:	[3 1 . 2]	[2 1 1]	[2 1 3 .]
19:	[3 1 2 .]	[3 1 1]	[3 1 2 .]
20:	[. 3 2 1]	[. 2 1]	[. 3 2 1]
21:	[3 . 2 1]	[1 2 1]	[1 3 2 .]
22:	[3 2 . 1]	[2 2 1]	[2 3 1 .]
23:	[3 2 1 .]	[3 2 1]	[3 2 1 .]

Figure 10.9-B: All permutations of 4 elements and falling factorial numbers used to update the permutations. Dots denote zeros.

```
    [ 0 1 4 3 2 5 ]    [ . . 2 1 . ]
```

is done in three steps:

```
    [ 4 2 3 5 1 0 ]    [ 5 4 1 1 . ]
    [ 4 3 2 5 1 0 ]    [ 5 4 2 1 . ]   move element at position d=1 to the right
    [ * * 4 3 2 5 ]    [ * * 2 1 . ]   move all but j=2 elements to end
    [ 0 1 4 3 2 5 ]    [ . . 2 1 . ]   insert identical permutation at start
```

We treat the first digit separately as it changes most often (easy case):

```
    bool next()
    {
        if ( ++ect_ < n_ ) // easy case
        {
            swap2(x_[ect_], x_[ect_-1]);
            return true;
        }
        else
        {
            ect_ = 0;
            ulong j = 1;
            ulong m1 = n_ - 2;   // nine in falling factorial base
            while ( d_[j]==m1 )  // find digit to increment
            {
                d_[j] = 0;
                --m1;
                ++j;
            }

            if ( j==n_-1 )  return false;  // current permutation is last

            const ulong dj = d_[j];
            d_[j] = dj + 1;

            // element at d[j] moves one position to the right:
            swap2( x_[dj], x_[dj+1] );

            { // move n-j elements to end:
                ulong s = n_-j,  d = n_;
                do
                {
                    --s;
                    --d;
                    x_[d] = x_[s];
                }
                while ( s );
            }

            // fill in 0,1,2,..,j-1 at start:
            for (ulong k=0; k<j; ++k)  x_[k] = k;

            return true;
        }
    }
}
```

The routine generates about 160 million permutations per second [FXT: comb/perm-mv0-demo.cc].

10.9.2 Ives' algorithm

An ordering where most of the moves are a move by one to the right of the smallest element is shown in figure 10.9-C. With n elements only one in $n\,(n-1)$ moves is more than a transposition (only the update from 12 to 13 in figure 10.9-C). The second half of the list of permutations is the reversed list of the reversed permutations in the first half. The algorithm, given by Ives [139], is implemented in [FXT: `class perm_ives` in comb/perm-ives.h]:

```
class perm_ives
{
public:
    ulong *p_;   // permutation
    ulong *ip_;  // inverse permutation
    ulong n_;    // permutations of n elements

public:
    perm_ives(ulong n)
```

```
             permutation    inv. perm.
        1:  [ . 1 2 3 ]    [ . 1 2 3 ]
        2:  [ 1 . 2 3 ]    [ 1 . 2 3 ]
        3:  [ 1 2 . 3 ]    [ 2 . 1 3 ]
        4:  [ 1 2 3 . ]    [ 3 . 1 2 ]
        5:  [ . 2 3 1 ]    [ . 3 1 2 ]
        6:  [ 2 . 3 1 ]    [ 1 3 . 2 ]
        7:  [ 2 3 . 1 ]    [ 2 3 . 1 ]
        8:  [ 2 3 1 . ]    [ 3 2 . 1 ]
        9:  [ . 3 1 2 ]    [ . 2 3 1 ]
       10:  [ 3 . 1 2 ]    [ 1 2 3 . ]
       11:  [ 3 1 . 2 ]    [ 2 1 3 . ]
       12:  [ 3 1 2 . ]    [ 3 1 2 . ]  << only update with more
       13:  [ . 2 1 3 ]    [ . 2 1 3 ]  << than one transposition
       14:  [ 2 . 1 3 ]    [ 1 2 . 3 ]
       15:  [ 2 1 . 3 ]    [ 2 1 . 3 ]
       16:  [ 2 1 3 . ]    [ 3 1 . 2 ]
       17:  [ . 1 3 2 ]    [ . 1 3 2 ]
       18:  [ 1 . 3 2 ]    [ 1 . 3 2 ]
       19:  [ 1 3 . 2 ]    [ 2 . 3 1 ]
       20:  [ 1 3 2 . ]    [ 3 . 2 1 ]
       21:  [ . 3 2 1 ]    [ . 3 2 1 ]
       22:  [ 3 . 2 1 ]    [ 1 3 2 . ]
       23:  [ 3 2 . 1 ]    [ 2 3 1 . ]
       24:  [ 3 2 1 . ]    [ 3 2 1 . ]
```

Figure 10.9-C: All permutations of 4 elements in an order by Ives.

```
    // Must have: n >= 2
    {
        n_ = n;
        p_ = new ulong[n_];
        ip_ = new ulong[n_];
        first();
    }
  [--snip--]
```

The computation of the successor is

```
    bool next()
    {
        ulong e1 = 0,  u = n_ - 1;
        do
        {
            const ulong i1 = ip_[e1];
            const ulong i2 = (i1==u ? e1 : i1+1 );
            const ulong e2 = p_[i2];
            p_[i1] = e2;  p_[i2] = e1;
            ip_[e1] = i2;  ip_[e2] = i1;
            if ( (p_[e1]!=e1) || (p_[u]!=u) )  return true;
            ++e1;
            --u;
        }
        while ( u > e1 );
        return false;
    }
  [--snip--]
```

The rate of generation is about 180 M/s [FXT: comb/perm-ives-demo.cc]. Using arrays instead of pointers increases the rate to about 190 M/s.

As the easy case with the update (when just the first element is moved) occurs so often it is natural to create an extra branch for it. When the define for `PERM_IVES_OPT` is made before the class definition then a counter is created:

```
class perm_ives
{
  [--snip--]
    ulong ctm_;   // aux: counter for easy case
    ulong ctm0_;  // aux: start value of ctm == n*(n-1)-1
  [--snip--]
```

When the counter is nonzero the following update can be used:

```
bool next()
{
    if ( ctm_-- )  // easy case
    {
        const ulong i1 = ip_[0];  // e1 == 0
        const ulong i2 = (i1==n_-1 ? 0 : i1+1);
        const ulong e2 = p_[i2];
        p_[i1] = e2;  p_[i2] = 0;
        ip_[0] = i2;  ip_[e2] = i1;
        return true;
    }
    ctm_ = ctm0_;
[--snip--]   // rest as before
}
```

When arrays are used a minimal speedup is obtained (rate 192 M/s), when pointers are used, the effect is a notable slowdown (rate 163 M/s).

The greatest speedup is obtained by a modified condition in the loop:

```
if ( (p_[e1]^e1) | (p_[u]^u) )  return true;
// same as:   if ( (p_[e1]!=e1) || (p_[u]!=u) )  return true;
```

The rate is increased to almost 194 M/s. This optimization is activated by defining `PERM_IVES_OPT2`.

10.10 Single track orders

	permutation		inv. perm.
0:	[. 2 3 1]	[. . .]	[. 3 1 2]
1:	[. 3 2 1]	[1 . .]	[. 3 2 1]
2:	[. 3 1 2]	[. 1 .]	[. 2 3 1]
3:	[. 2 1 3]	[1 1 .]	[. 2 1 3]
4:	[. 1 2 3]	[. 2 .]	[. 1 2 3]
5:	[. 1 3 2]	[1 2 .]	[. 1 3 2]
6:	[1 . 2 3]	[. . 1]	[1 . 2 3]
7:	[1 . 3 2]	[1 . 1]	[1 . 3 2]
8:	[2 . 3 1]	[. 1 1]	[1 3 . 2]
9:	[3 . 2 1]	[1 1 1]	[1 3 2 .]
10:	[3 . 1 2]	[. 2 1]	[1 2 3 .]
11:	[2 . 1 3]	[1 2 1]	[1 2 . 3]
12:	[3 1 . 2]	[. . 2]	[2 1 3 .]
13:	[2 1 . 3]	[1 . 2]	[2 1 . 3]
14:	[1 2 . 3]	[. 1 2]	[2 . 1 3]
15:	[1 3 . 2]	[1 1 2]	[2 . 3 1]
16:	[2 3 . 1]	[. 2 2]	[2 3 . 1]
17:	[3 2 . 1]	[1 2 2]	[2 3 1 .]
18:	[2 3 1 .]	[. . 3]	[3 2 . 1]
19:	[3 2 1 .]	[1 . 3]	[3 2 1 .]
20:	[3 1 2 .]	[. 1 3]	[3 1 2 .]
21:	[2 1 3 .]	[1 1 3]	[3 1 . 2]
22:	[1 2 3 .]	[. 2 3]	[3 . 1 2]
23:	[1 3 2 .]	[1 2 3]	[3 . 2 1]

Figure 10.10-A: Permutations of 4 elements in single track order. Dots denote zeros.

Figure 10.10-A shows a *single track order* for the permutations of four elements. Each column in the list of permutations is a cyclic shift of the first column. A recursive construction for the ordering is shown in figure 10.10-B. The figure was created with the program [FXT: comb/perm-st-demo.cc] which uses [FXT: `class perm_st` in comb/perm-st.h]:

```
class perm_st
{
public:
    ulong *d_;  // mixed radix digits with radix = [2, 3, 4, ..., n-1, (sentinel=-1)]
    ulong *p_;  // permutation
```

```
        23  <--=  permutations of 2 elements
        32

        11 23 32  <--= concatenate rows and prepend new element

        112332   <--= shift  0
        321123   <--= shift  2
        233211   <--= shift  4


        000000 112332 321123 233211  <--= concatenate rows and prepend new element

        000000 112332 321123 233211  <--= shift  0
        233211 000000 112332 321123  <--= shift  6
        321123 233211 000000 112332  <--= shift 12
        112332 321123 233211 000000  <--= shift 18
```

Figure 10.10-B: Construction of the single track order for permutations of 4 elements.

```
    ulong *pi_; // inverse permutation
    ulong n_;   // permutations of n elements
public:
    perm_st(ulong n)
    {
        n_ = n;
        d_ = new ulong[n_];
        p_ = new ulong[n_];
        pi_ = new ulong[n_];
        d_[n-1] = -1UL;  // sentinel
        first();
    }
  [--snip--]
```

The first permutation is in enup order (see section 6.5.1 on page 175):

```
    const ulong *data()  const { return p_; }
    const ulong *invdata()  const  { return pi_; }

    void first()
    {
        for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
        for (ulong k=0, e=0;  k<n_;  ++k)
        {
            p_[k] = e;
            pi_[e] = k;
            e = next_enup(e, n_-1);
        }
    }
  [--snip--]
```

The swap with the inverse permutations are determined by the rightmost position j changing with mixed radix counting with rising factorial base. We write -1 for the last element, -2 for the second last, and so on:

```
    j   swaps
    0:  (-2,-1)
    1:  (-3,-2)
    2:  (-4,-3) (-2,-1)
    3:  (-5,-4) (-3,-2)
    4:  (-6,-5) (-4,-3) (-2,-1)
    5:  (-7,-6) (-5,-4) (-3,-2)
    j:  (-j-2, -j-1) ... (-2-(j%1), -1-(j%1))
```

The computation of the successor is CAT:

```
    bool next()
    {
        // increment mixed radix number:
        ulong j = 0;
        while ( d_[j]==j+1 )  { d_[j]=0;  ++j; }

        if ( j==n_-1 )  return false; // current permutation is last
        ++d_[j];

        for (ulong e1=n_-2-j, e2=e1+1;  e2<n_;  e1+=2, e2+=2)
        {
```

```
            const ulong i1 = pi_[e1];  // position of element e1
            const ulong i2 = pi_[e2];  // position of element e2
            pi_[e1] = i2;
            pi_[e2] = i1;
            p_[i1] = e2;
            p_[i2] = e1;
        }
        return true;
    }
```

All swaps with the inverse permutations are of adjacent pairs. The reversals of the first half of all permutations lie in the second half, the reversal of the k-the permutation lies at position $n! - 1 - k$

```
                permutation                 inv. perm.
 0:     [ . 1 2 3 ]     [ . . . ]     [ . 1 2 3 ]
 1:     [ . 1 3 2 ]     [ 1 . . ]     [ . 1 3 2 ]
 2:     [ . 2 3 1 ]     [ . 1 . ]     [ . 3 1 2 ]
 3:     [ . 3 2 1 ]     [ 1 1 . ]     [ . 3 2 1 ]
 4:     [ . 3 1 2 ]     [ . 2 . ]     [ . 2 3 1 ]
 5:     [ . 2 1 3 ]     [ 1 2 . ]     [ . 2 1 3 ]
 6:     [ 1 3 . 2 ]     [ . . 1 ]     [ 2 . 3 1 ]
 7:     [ 1 2 . 3 ]     [ 1 . 1 ]     [ 2 . 1 3 ]
 8:     [ 2 1 . 3 ]     [ . 1 1 ]     [ 2 1 . 3 ]
 9:     [ 3 1 . 2 ]     [ 1 1 1 ]     [ 2 1 3 . ]
10:     [ 3 2 . 1 ]     [ . 2 1 ]     [ 2 3 1 . ]
11:     [ 2 3 . 1 ]     [ 1 2 1 ]     [ 2 3 . 1 ]
12:     [ 3 2 1 . ]     [ . . 2 ]     [ 3 2 1 . ]
13:     [ 2 3 1 . ]     [ 1 . 2 ]     [ 3 2 . 1 ]
14:     [ 1 3 2 . ]     [ . 1 2 ]     [ 3 . 2 1 ]
15:     [ 1 2 3 . ]     [ 1 1 2 ]     [ 3 . 1 2 ]
16:     [ 2 1 3 . ]     [ . 2 2 ]     [ 3 1 . 2 ]
17:     [ 3 1 2 . ]     [ 1 2 2 ]     [ 3 1 2 . ]
18:     [ 2 . 3 1 ]     [ . . 3 ]     [ 1 3 . 2 ]
19:     [ 3 . 2 1 ]     [ 1 . 3 ]     [ 1 3 2 . ]
20:     [ 3 . 1 2 ]     [ . 1 3 ]     [ 1 2 3 . ]
21:     [ 2 . 1 3 ]     [ 1 1 3 ]     [ 1 2 . 3 ]
22:     [ 1 . 2 3 ]     [ . 2 3 ]     [ 1 . 2 3 ]
23:     [ 1 . 3 2 ]     [ 1 2 3 ]     [ 1 . 3 2 ]
```

Figure 10.10-C: Permutations of 4 elements in single track order starting with the identical permutation.

The single track property is independent of the first permutation, one can start with the trivial permutation using

```
void first_id()  // start with identical permutation
{
    for (ulong k=0; k<n_-1; ++k)  d_[k] = 0;
    for (ulong k=0; k<n_; ++k)  p_[k] = pi_[k] = k;
}
```

The ordering obtained is shown in figure 10.10-C, The reversal of the k-the permutation lies at position $(n!)/2 + k$. About 85 million permutations per second can be generated.

10.10.1 Construction of all single track orders

A construction for a single track order of $n+1$ elements from an arbitrary ordering of n elements is shown in figure 10.10-D (for $n = 3$ and lexicographic oder). Thereby we obtain as many single track orders for the permutations of n elements as there are orders of the permutations of $n-1$ elements, namely $((n-1)!)!$. One can apply cyclic shifts in each blocks as shown in figure 10.10-E. The shifts in the first $(n-1)!$ positions (first blocks in the figure) determine the shifts for the remaining permutations. Now there are n different cyclic shifts in each position. Indeed all single track orderings are of this form, so their number is

$$N_s(n) \quad = \quad ((n-1)!)!\; n^{(n-1)!} \tag{10.10-1}$$

```
112233
231312  <--= permutations of 3 elements in lex order (columns)
323121

000000 112233 231312 323121  <--= concatenate rows and prepend new element

000000 112233 231312 323121  <--= shift  0
323121 000000 112233 231312  <--= shift  6
231312 323121 000000 112233  <--= shift 12
112233 231312 323121 000000  <--= shift 18
```

Figure 10.10-D: Construction of a single track order for permutations of 4 elements from an arbitrary ordering of the permutations of 3 elements.

```
  single track ordering          modified single track ordering
..... 112233 231312 323121       21.113 1.333. .212.1 332.22
323121 ..... 112233 231312       1.333. .212.1 332.22 21.113
231312 323121 ..... 112233       .212.1 332.22 21.113 1.333.
112233 231312 323121 .....       332.22 21.113 1.333. .212.1

^^^^^^                           ^^^^^^
000000                           210321 <--= cyclic shifts
```

Figure 10.10-E: In each of the first $(n-1)!$ permutations in a single track ordering (first block left) an arbitrary rotation can be applied (first block right), leading to a different single track ordering.

The number of single track orders that start with the identical permutation, and where the k-th run of $(n-1)!$ elements starts with k (and thereby all shifts between successive tracks are left shifts by $(n-1)!$ positions) is

$$N_s(n)/n! \quad = \quad ((n-1)!-1)!\, n^{(n-1)!-1} \tag{10.10-2}$$

10.10.2 A single track Gray code

```
[ . 1 2 3 4 ]    [ 1 2 3 4 . ]    [ 2 3 4 . 1 ]    [ 3 4 . 1 2 ]    [ 4 . 1 2 3 ]
[ 1 . 2 3 4 ]    [ . 2 3 4 1 ]    [ 2 3 4 1 . ]    [ 3 4 1 . 2 ]    [ 4 1 . 2 3 ]
[ 2 . 1 3 4 ]    [ . 1 3 4 2 ]    [ 1 3 4 2 . ]    [ 3 4 2 . 1 ]    [ 4 2 . 1 3 ]
[ . 2 1 3 4 ]    [ 2 1 3 4 . ]    [ 1 3 4 . 2 ]    [ 3 4 . 2 1 ]    [ 4 . 2 1 3 ]
[ 1 2 . 3 4 ]    [ 2 . 3 4 1 ]    [ . 3 4 1 2 ]    [ 3 4 1 2 . ]    [ 4 1 2 . 3 ]
[ 2 1 . 3 4 ]    [ 1 . 3 4 2 ]    [ . 3 4 2 1 ]    [ 3 4 2 1 . ]    [ 4 2 1 . 3 ]
[ 3 1 . 2 4 ]    [ 1 . 2 4 3 ]    [ . 2 4 3 1 ]    [ 2 4 3 1 . ]    [ 4 3 1 . 2 ]
[ 1 3 . 2 4 ]    [ 3 . 2 4 1 ]    [ . 2 4 1 3 ]    [ 2 4 1 3 . ]    [ 4 1 3 . 2 ]
[ . 3 1 2 4 ]    [ 3 1 2 4 . ]    [ 1 2 4 . 3 ]    [ 2 4 . 3 1 ]    [ 4 . 3 1 2 ]
[ 3 . 1 2 4 ]    [ . 1 2 4 3 ]    [ 1 2 4 3 . ]    [ 2 4 3 . 1 ]    [ 4 3 . 1 2 ]
[ 1 . 3 2 4 ]    [ . 3 2 4 1 ]    [ 3 2 4 1 . ]    [ 2 4 1 . 3 ]    [ 4 1 . 3 2 ]
[ . 1 3 2 4 ]    [ 1 3 2 4 . ]    [ 3 2 4 . 1 ]    [ 2 4 . 1 3 ]    [ 4 . 1 3 2 ]
[ . 2 3 1 4 ]    [ 2 3 1 4 . ]    [ 3 1 4 . 2 ]    [ 1 4 . 2 3 ]    [ 4 . 2 3 1 ]
[ 2 . 3 1 4 ]    [ . 3 1 4 2 ]    [ 3 1 4 2 . ]    [ 1 4 2 . 3 ]    [ 4 2 . 3 1 ]
[ 3 . 2 1 4 ]    [ . 2 1 4 3 ]    [ 2 1 4 3 . ]    [ 1 4 3 . 2 ]    [ 4 3 . 2 1 ]
[ . 3 2 1 4 ]    [ 3 2 1 4 . ]    [ 2 1 4 . 3 ]    [ 1 4 . 3 2 ]    [ 4 . 3 2 1 ]
[ 2 3 . 1 4 ]    [ 3 . 1 4 2 ]    [ . 1 4 2 3 ]    [ 1 4 2 3 . ]    [ 4 2 3 . 1 ]
[ 3 2 . 1 4 ]    [ 2 . 1 4 3 ]    [ . 1 4 3 2 ]    [ 1 4 3 2 . ]    [ 4 3 2 . 1 ]
[ 3 2 1 . 4 ]    [ 2 1 . 4 3 ]    [ 1 . 4 3 2 ]    [ . 4 3 2 1 ]    [ 4 3 2 1 . ]
[ 2 3 1 . 4 ]    [ 3 1 . 4 2 ]    [ 1 . 4 2 3 ]    [ . 4 2 3 1 ]    [ 4 2 3 1 . ]
[ 1 3 2 . 4 ]    [ 3 2 . 4 1 ]    [ 2 . 4 1 3 ]    [ . 4 1 3 2 ]    [ 4 1 3 2 . ]
[ 3 1 2 . 4 ]    [ 1 2 . 4 3 ]    [ 2 . 4 3 1 ]    [ . 4 3 1 2 ]    [ 4 3 1 2 . ]
[ 2 1 3 . 4 ]    [ 1 3 . 4 2 ]    [ 3 . 4 2 1 ]    [ . 4 2 1 3 ]    [ 4 2 1 3 . ]
[ 1 2 3 . 4 ]    [ 2 3 . 4 1 ]    [ 3 . 4 1 2 ]    [ . 4 1 2 3 ]    [ 4 1 2 3 . ]
```

Figure 10.10-F: A cyclic Gray code for the permutations of 5 elements with the single track property.

A Gray code for permutations that has the single track property can be constructed by using a Gray code for the permutations of $n-1$ elements if the first and last permutation are cyclic shifts by one position of each other. Such Gray codes exist for even lengths only. Figure 10.10-F shows a single track Gray code for $n = 5$. For even n we use a Gray code where all but the last element are cyclically shifted

```
        1:    [ 0 1 2 3 4 5 ]
        2:    [ 1 0 2 3 4 5 ]
        3:    [ 2 0 1 3 4 5 ]
        4:    [ 0 2 1 3 4 5 ]
        5:    [ 1 2 0 3 4 5 ]
          [--Gray transitions only--]
      116:    [ 2 3 1 0 4 5 ]
      117:    [ 1 3 2 0 4 5 ]
      118:    [ 3 1 2 0 4 5 ]
      119:    [ 2 1 3 0 4 5 ]
      120:    [ 1 2 3 0 4 5 ]
      121:    [ 1 2 3 4 5 0 ]  << (0, 4, 5)
         [--Gray transitions only--]
      240:    [ 2 3 0 4 5 1 ]
      241:    [ 2 3 4 5 0 1 ]  << (0, 4, 5)
         [--Gray transitions only--]
      360:    [ 3 0 4 5 1 2 ]
      361:    [ 3 4 5 0 1 2 ]  << (0, 4, 5)
         [--Gray transitions only--]
      480:    [ 0 4 5 1 2 3 ]
      481:    [ 4 5 0 1 2 3 ]  << (0, 4, 5)
         [--Gray transitions only--]
      600:    [ 4 5 1 2 3 0 ]
      601:    [ 5 0 1 2 3 4 ]  << (0, 4, 5)
         [--Gray transitions only--]
      720:    [ 5 1 2 3 0 4 ]
        1:    [ 0 1 2 3 4 5 ]  << (0, 4, 5)
```

Figure 10.10-G: The single track ordering for odd n with the least number of transpositions contains $n-1$ extra transpositions. Here the non-Gray transitions are cycles between the elements 0, 4, and 5.

between the first and last permutation. Such a Gray code is given in section 10.8.3 on page 248. The resulting single track order is as close to a Gray code as possible, just $n-1$ extra transpositions occur for all permutation of n elements, see figure 10.10-G. The listings were created with the program [FXT: comb/perm-st-gray-demo.cc] which uses [FXT: `class perm_st_gray` in comb/perm-st-gray.h]:

```
class perm_st_gray
{
public:
    perm_gray_rot1 *G;  // underlying permutations
    ulong *x_;   // permutation
    ulong *ix_;  // inverse permutation
    ulong n_;    // number of elements
    ulong sct_;  // count cyclic shifts
public:
    perm_st_gray(ulong n)
    // Must have n>=2
    {
        n_ = (n>=2 ? n : 2);
        G = new perm_gray_rot1(n-1);
        x_ = new ulong[n_];
        ix_ = new ulong[n_];
        first();
    }
  [--snip--]
    void first()
    {
        G->first();
        for (ulong j=0; j<n_; ++j)  ix_[j] = x_[j] = j;
        sct_ = n_;
    }
```

We define two auxiliary routine to swap elements by their value and by their positions:

```
private:
    void swap_elements(ulong x1, ulong x2)
    {
        const ulong i1 = ix_[x1],  i2 = ix_[x2];
        x_[i1] = x2;   x_[i2] = x1;   // swap2(x_[i1], x_[i2]);
```

```
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
    }
    void swap_positions(ulong i1, ulong i2)
    {
        const ulong x1 = x_[i1],  x2 = x_[i2];
        x_[i1] = x2;   x_[i2] = x1;   // swap2(x_[i1], x_[i2]);
        ix_[x1] = i2;  ix_[x2] = i1;  // swap2(ix_[x1], ix_[x2]);
    }
```

The update routine consists of two cases. The frequent case is the update via the underlying permutation:

```
public:
    bool next()
    {
        bool q = G->next();
        if ( q )  // normal update (in underlying permutation of n-1 elements)
        {
            ulong i1, i2;  // positions of swaps
            G->get_swap(i1, i2);

            // rotate positions according to sct:
            i1 += sct_;  if ( i1>=n_ )  i1-=n_;
            i2 += sct_;  if ( i2>=n_ )  i2-=n_;

            swap_positions(i1, i2);

            return true;
        }
```

The infrequent case happens when the last underlying permutation is encountered:

```
        else  // goto next cyclic shift (once in (n-1)! updates, n-1 times in total)
        {
            G->first();  // restart underlying permutations
            --sct_;      // adjust cyclic shift

            swap_elements(0, n_-1);

            if ( 0==(n_&1) )  // n even
                if ( n_>=4 )  swap_elements(n_-2, n_-1);  // one extra transposition

            return ( 0!=sct_ );
        }
    }
```

10.11 Star-transposition order

The permutations can be ordered so that successive permutations differ by a swap of the element at the first position with some other element (star transposition), figure 10.11-A. An algorithm for the generation of such an ordering is given in [157]. The implementation of the algorithm, ascribed to Gideon Ehrlich, is given in [FXT: `class perm_star` in comb/perm-star.h]. The generation of the inverse permutations is an option that is activated by the `#define PERM_STAR_WITH_INVERSE`. If the successive permutations differ by a transposition

```
swap2(a_[0], a_[swp_]);
```

then the inverse can be updated as

```
swap2(ia_[a_[0]], ia_[a_[swp_]]);
```

Note that in the sequence of the inverse permutations the zero is always moved. In the list of the inverse permutations the reversed permutations of the first half are in the second half.

The listing shown in figure 10.11-A can be obtained with [FXT: comb/perm-star-demo.cc]. About 77 million permutations per second are generated and about 115 million when the inverse permutation is not computed. If the only the swaps are of interest then use [FXT: `class perm_star_swaps` in comb/perm-star-swaps.h] whose update routine works at a rate about 158 million per second [FXT: comb/perm-star-swaps-demo.cc].

```
             permutation     swap      inverse p.
      0:    [ . 1 2 3 ]                [ . 1 2 3 ]
      1:    [ 1 . 2 3 ]    (0, 1)      [ 1 . 2 3 ]
      2:    [ 2 . 1 3 ]    (0, 2)      [ 1 2 . 3 ]
      3:    [ . 2 1 3 ]    (0, 1)      [ . 2 1 3 ]
      4:    [ 1 2 . 3 ]    (0, 2)      [ 2 . 1 3 ]
      5:    [ 2 1 . 3 ]    (0, 1)      [ 2 1 . 3 ]
      6:    [ 3 1 . 2 ]    (0, 3)      [ 2 1 3 . ]
      7:    [ . 1 3 2 ]    (0, 2)      [ . 1 3 2 ]
      8:    [ 1 . 3 2 ]    (0, 1)      [ 1 . 3 2 ]
      9:    [ 3 . 1 2 ]    (0, 2)      [ 1 2 3 . ]
     10:    [ . 3 1 2 ]    (0, 1)      [ . 2 3 1 ]
     11:    [ 1 3 . 2 ]    (0, 2)      [ 2 . 3 1 ]
     12:    [ 2 3 . 1 ]    (0, 3)      [ 2 3 . 1 ]
     13:    [ 3 2 . 1 ]    (0, 1)      [ 2 3 1 . ]
     14:    [ . 2 3 1 ]    (0, 2)      [ . 3 1 2 ]
     15:    [ 2 . 3 1 ]    (0, 1)      [ 1 3 . 2 ]
     16:    [ 3 . 2 1 ]    (0, 2)      [ 1 3 2 . ]
     17:    [ . 3 2 1 ]    (0, 1)      [ . 3 2 1 ]
     18:    [ 1 3 2 . ]    (0, 3)      [ 3 . 2 1 ]
     19:    [ 2 3 1 . ]    (0, 2)      [ 3 2 . 1 ]
     20:    [ 3 2 1 . ]    (0, 1)      [ 3 2 1 . ]
     21:    [ 1 2 3 . ]    (0, 2)      [ 3 . 1 2 ]
     22:    [ 2 1 3 . ]    (0, 1)      [ 3 1 . 2 ]
     23:    [ 3 1 2 . ]    (0, 2)      [ 3 1 2 . ]
```

Figure 10.11-A: The permutations of 4 elements in star-transposition order. Dots denote zeros.

10.12 Derangement order

The *derangement order* for permutations is characterized by the fact that two successive permutations have no element at the same position, as shown in figure 10.12-A. The listing was created with the program [FXT: comb/perm-derange-demo.cc]. There is no such sequence for $n = 3$. The implementation of the underlying algorithm is [FXT: `class perm_derange` in comb/perm-derange.h]:

```
class perm_derange
{
public:
    ulong n_;   // number of elements
    ulong *x_;  // current permutation
    ulong ctm_; // counter modulo n
    perm_trotter* T_;
public:
    perm_derange(ulong n)
    // Must have:  n>=4
    // n=2: trivial, n=3: no solution exists,  n>=4: ok
    {
        n_ = n;
        x_ = new ulong[n_];
        T_ = new perm_trotter(n_-1);
        first();
    }
  [--snip--]
```

The routine to update the permutation is

```
    bool next()
    {
        ++ctm_;
        if ( ctm_>=n_ ) // every n steps: need next perm_trotter
        {
            ctm_ = 0;
            if ( ! T_->next() )  return false;  // current permutation is last
            const ulong *t = T_->data();
            for (ulong k=0; k<n_-1; ++k)  x_[k] = t[k];
            x_[n_-1] = n_-1;  // last element
        }
        else  // rotate
        {
```

```
              permutation        inverse perm.
      0:      [ . 1 2 3 ]        [ . 1 2 3 ]
      1:      [ 3 . 1 2 ]        [ 1 2 3 . ]
      2:      [ 1 2 3 . ]        [ 3 . 1 2 ]
      3:      [ 2 3 . 1 ]        [ 2 3 . 1 ]
      4:      [ 1 . 2 3 ]        [ 1 . 2 3 ]
      5:      [ 3 1 . 2 ]        [ 2 1 3 . ]
      6:      [ . 2 3 1 ]        [ . 3 1 2 ]
      7:      [ 2 3 1 . ]        [ 3 2 . 1 ]
      8:      [ 1 2 . 3 ]        [ 2 . 1 3 ]
      9:      [ 3 1 2 . ]        [ 3 1 2 . ]
     10:      [ 2 . 3 1 ]        [ 1 3 . 2 ]
     11:      [ . 3 1 2 ]        [ . 2 3 1 ]
     12:      [ 2 1 . 3 ]        [ 2 1 . 3 ]
     13:      [ 3 2 1 . ]        [ 3 2 1 . ]
     14:      [ 1 . 3 2 ]        [ 1 . 3 2 ]
     15:      [ . 3 2 1 ]        [ . 3 2 1 ]
     16:      [ 2 . 1 3 ]        [ 1 2 . 3 ]
     17:      [ 3 2 . 1 ]        [ 2 3 1 . ]
     18:      [ . 1 3 2 ]        [ . 1 3 2 ]
     19:      [ 1 3 2 . ]        [ 3 . 2 1 ]
     20:      [ . 2 1 3 ]        [ . 2 1 3 ]
     21:      [ 3 . 2 1 ]        [ 1 3 2 . ]
     22:      [ 2 1 3 . ]        [ 3 1 . 2 ]
     23:      [ 1 3 . 2 ]        [ 2 . 3 1 ]
```

Figure 10.12-A: The permutations of 4 elements in derangement order.

```
            if ( ctm_==n_-1 )  rotate_left1(x_, n_);
            else // last two elements swapped
            {
                rotate_right1(x_, n_);
                if ( ctm_==n_-2 )  rotate_right1(x_, n_);
            }
        }
        return true;
    }
```

The routines `rotate_right1()` and `rotate_last()` rotate the array `x_[]` by one position [FXT: perm/rotate.h]. These rotations are the performance bottleneck, the cost of one update of a length-n permutation is proportional to n. Still, about 35 million permutations per second are generated for $n = 12$.

Derangement order for even n

An algorithm for the generation of permutations via cyclic shifts suggested in [163] generates a derangement order if the number n of elements is even, see figure 10.12-B. An implementation of the algorithm, following [157], is [FXT: `class perm_rot` in comb/perm-rot.h]. For odd n the number of times that the successor is not a derangement of the predecessor equals $((n + 1)/2)! - 1$. The program [FXT: comb/perm-rot-demo.cc] generates the permutations and counts those transitions.

An alternative ordering with the same number of transitions that are no derangements is obtained via mixed radix counting in falling factorial basis and the routine [FXT: comb/perm-rot-unrank-demo.cc]

```
void ffact2perm_rot(const ulong *fc, ulong n, ulong *x)
// Convert falling factorial number fc[0, ..., n-2] into
// permutation of x[0, ... ,n-1].
{
    for (ulong k=0; k<n; ++k)  x[k] = k;
    for (ulong k=n-1, j=2;  k!=0;  --k, ++j)  rotate_right(x+k-1, j, fc[k-1]);
}
```

Figure 10.12-C shows the generated ordering for $n = 4$ and $n = 3$. The observation that the permutations in second ordering are the complemented reversals of the first leads to the unranking routine

```
class perm_rot
{
```

```
          permutation    inv. perm.
      0:  [ . 1 2 3 ]    [ . 1 2 3 ]
      1:  [ 1 2 3 . ]    [ 3 . 1 2 ]
      2:  [ 2 3 . 1 ]    [ 2 3 . 1 ]
      3:  [ 3 . 1 2 ]    [ 1 2 3 . ]
      4:  [ 1 2 . 3 ]    [ 2 . 1 3 ]
      5:  [ 2 . 3 1 ]    [ 1 3 . 2 ]
      6:  [ . 3 1 2 ]    [ . 2 3 1 ]
      7:  [ 3 1 2 . ]    [ 3 1 2 . ]
      8:  [ 2 . 1 3 ]    [ 1 2 . 3 ]
      9:  [ . 1 3 2 ]    [ . 1 3 2 ]
     10:  [ 1 3 2 . ]    [ 3 . 2 1 ]
     11:  [ 3 2 . 1 ]    [ 2 3 1 . ]
     12:  [ 1 . 2 3 ]    [ 1 . 2 3 ]
     13:  [ . 2 3 1 ]    [ . 3 1 2 ]
     14:  [ 2 3 1 . ]    [ 3 2 . 1 ]
     15:  [ 3 1 . 2 ]    [ 2 1 3 . ]
     16:  [ . 2 1 3 ]    [ . 2 1 3 ]
     17:  [ 2 1 3 . ]    [ 3 1 . 2 ]
     18:  [ 1 3 . 2 ]    [ 2 . 3 1 ]
     19:  [ 3 . 2 1 ]    [ 1 3 2 . ]
     20:  [ 2 1 . 3 ]    [ 2 1 . 3 ]
     21:  [ 1 . 3 2 ]    [ 1 . 3 2 ]
     22:  [ . 3 2 1 ]    [ . 3 2 1 ]
     23:  [ 3 2 1 . ]    [ 3 2 1 . ]
```

```
          permutation  inv. perm.
      0:  [ . 1 2 ]    [ . 1 2 ]
      1:  [ 1 2 . ]    [ 2 . 1 ]
      2:  [ 2 . 1 ]    [ 1 2 . ]  <<
      3:  [ 1 . 2 ]    [ 1 . 2 ]  <<
      4:  [ . 2 1 ]    [ . 2 1 ]
      5:  [ 2 1 . ]    [ 2 1 . ]
```

Figure 10.12-B: Permutations generated via cyclic shifts. The order is a derangement order for even n (left), but not for odd n (right). Dots denote zeros.

```
            ffact    permutation  inv. perm.
      0:  [ . . . ]  [ . 1 2 3 ]  [ . 1 2 3 ]
      1:  [ 1 . . ]  [ 3 . 1 2 ]  [ 1 2 3 . ]
      2:  [ 2 . . ]  [ 2 3 . 1 ]  [ 2 3 . 1 ]
      3:  [ 3 . . ]  [ 1 2 3 . ]  [ 3 . 1 2 ]
      4:  [ . 1 . ]  [ . 3 1 2 ]  [ . 2 3 1 ]
      5:  [ 1 1 . ]  [ 2 . 3 1 ]  [ 1 3 . 2 ]
      6:  [ 2 1 . ]  [ 1 2 . 3 ]  [ 2 . 1 3 ]
      7:  [ 3 1 . ]  [ 3 1 2 . ]  [ 3 1 2 . ]
      8:  [ . 2 . ]  [ . 2 3 1 ]  [ . 3 1 2 ]
      9:  [ 1 2 . ]  [ 1 . 2 3 ]  [ 1 . 2 3 ]
     10:  [ 2 2 . ]  [ 3 1 . 2 ]  [ 2 1 3 . ]
     11:  [ 3 2 . ]  [ 2 3 1 . ]  [ 3 2 . 1 ]
     12:  [ . . 1 ]  [ . 1 3 2 ]  [ . 1 3 2 ]
     13:  [ 1 . 1 ]  [ 2 . 1 3 ]  [ 1 2 . 3 ]
     14:  [ 2 . 1 ]  [ 3 2 . 1 ]  [ 2 3 1 . ]
     15:  [ 3 . 1 ]  [ 1 3 2 . ]  [ 3 . 2 1 ]
     16:  [ . 1 1 ]  [ . 2 1 3 ]  [ . 2 1 3 ]
     17:  [ 1 1 1 ]  [ 3 . 2 1 ]  [ 1 3 2 . ]
     18:  [ 2 1 1 ]  [ 1 3 . 2 ]  [ 2 . 3 1 ]
     19:  [ 3 1 1 ]  [ 2 1 3 . ]  [ 3 1 . 2 ]
     20:  [ . 2 1 ]  [ . 3 2 1 ]  [ . 3 2 1 ]
     21:  [ 1 2 1 ]  [ 1 . 3 2 ]  [ 1 . 3 2 ]
     22:  [ 2 2 1 ]  [ 2 1 . 3 ]  [ 2 1 . 3 ]
     23:  [ 3 2 1 ]  [ 3 2 1 . ]  [ 3 2 1 . ]
```

```
          ffact      perm.    inv. perm.
      0:  [ . . ]  [ . 1 2 ]  [ . 1 2 ]
      1:  [ 1 . ]  [ 2 . 1 ]  [ 1 2 . ]
      2:  [ 2 . ]  [ 1 2 . ]  [ 2 . 1 ]  <<
      3:  [ . 1 ]  [ . 2 1 ]  [ . 2 1 ]  <<
      4:  [ 1 1 ]  [ 1 . 2 ]  [ 1 . 2 ]
      5:  [ 2 1 ]  [ 2 1 . ]  [ 2 1 . ]
```

Figure 10.12-C: Alternative ordering for permutations generated via cyclic shifts. The order is a derangement order for even n (left), but not for odd n (right).

```
    ulong *a_;  // permutation of n elements
    ulong n_;
 [--snip--]
    void goto_ffact(const ulong *d)
    // Goto permutation corresponding to d[] (i.e. unrank d[]).
    // d[] must be a valid (falling) factorial mixed radix string.
    {
        for (ulong k=0; k<n_; ++k)  a_[k] = k;
        for (ulong k=n_-1, j=2;  k!=0;  --k, ++j)  rotate_right(a_+k-1, j, d[k-1]);
        reverse(a_, n_);
        make_complement(a_, a_, n_);
    }
 [--snip--]
}
```

Compare to the unranking for permutations by prefix reversals shown in section 10.4.2 on page 234.

10.13 Recursive algorithm for cyclic permutations

	permutation	inverse	ffact-swp
0:	[. 1 2 3]	[. 1 2 3]	[. . .]
1:	[. 1 3 2]	[. 1 3 2]	[. . 1]
2:	[. 2 1 3]	[. 2 1 3]	[. 1 .]
3:	[. 2 3 1]	[. 3 1 2]	[. 1 1]
4:	[. 3 2 1]	[. 3 2 1]	[. 2 .]
5:	[. 3 1 2]	[. 2 3 1]	[. 2 1]
6:	[1 . 2 3]	[1 . 2 3]	[1 . .]
7:	[1 . 3 2]	[1 . 3 2]	[1 . 1]
8:	[1 2 . 3]	[2 . 1 3]	[1 1 .]
9:	[1 2 3 .]	[3 . 1 2]	[1 1 1]
10:	[1 3 2 .]	[3 . 2 1]	[1 2 .]
11:	[1 3 . 2]	[2 . 3 1]	[1 2 1]
12:	[2 1 . 3]	[2 1 . 3]	[2 . .]
13:	[2 1 3 .]	[3 1 . 2]	[2 . 1]
14:	[2 . 1 3]	[1 2 . 3]	[2 1 .]
15:	[2 . 3 1]	[1 3 . 2]	[2 1 1]
16:	[2 3 . 1]	[2 3 . 1]	[2 2 .]
17:	[2 3 1 .]	[3 2 . 1]	[2 2 1]
18:	[3 1 2 .]	[3 1 2 .]	[3 . .]
19:	[3 1 . 2]	[2 1 3 .]	[3 . 1]
20:	[3 2 1 .]	[3 2 1 .]	[3 1 .]
21:	[3 2 . 1]	[2 3 1 .]	[3 1 1]
22:	[3 . 2 1]	[1 3 2 .]	[3 2 .]
23:	[3 . 1 2]	[1 2 3 .]	[3 2 1]

Figure 10.13-A: All permutations of 4 elements (left) and their inverses (middle), and their (swaps-) representations as mixed radix numbers with falling factorial basis. Permutations with common prefixes appear in succession. Dots denote zeros.

A simple recursive algorithm for the generation of the permutations of n elements can be described as follows: Put each of the n element of the array to the first position and generate all permutations of $n-1$ elements. If n equals one, print the permutation.

The order obtained is shown in figure 10.13-A, it corresponds to the alternative (swaps-) factorial representation with falling basis, given in section 10.3.3 on page 227.

The algorithm is implemented in [FXT: `class perm_rec` in comb/perm-rec.h]:

```
class perm_rec
{
public:
    ulong *x_;   // permutation
    ulong n_;    // number of elements
    void (*visit_)(const perm_lex_rec &);  // function to call with each permutation
public:
```

```
    perm_rec(ulong n)
    {
        n_ = n;
        x_ = new ulong[n_];
    }
    ~perm_rec()
    { delete [] x_; }
    void init()
    {
        for (ulong k=0; k<n_; ++k)  x_[k] = k;
    }
    void generate(void (*visit)(const perm_lex_rec &))
    {
        visit_ = visit;
        init();
        next_rec(0);
    }
```

The recursive function `next_rec()` is

```
    void next_rec(ulong d)
    {
        if ( d==n_-1 )  visit_(*this);
        else
        {
            const ulong pd = x_[d];
            for (ulong k=d; k<n_; ++k)
            {
                ulong px = x_[k];
                x_[k] = pd;  x_[d] = px;  // =^= swap2(x_[d], x_[k]);
                next_rec(d+1);
                x_[k] = px;  x_[d] = pd;  // =^= swap2(x_[d], x_[k]);
            }
        }
    }
```

The algorithm works because at each recursive call the elements `x[d],...,x[n-1]` are in a different order and when the function returns the elements are in the same order as they were initially. With the '`for`'-statement changed to

```
        for (ulong x=n_-1; (long)x>=(long)d; --x)
```

the permutations would appear in reversed order. Changing the loop in the function `next_rec()` to

```
            for (ulong k=d; k<n_; ++k)
            {
                swap2(x_[d], x_[k]);
                next_rec(d+1, qq);
            }
            rotate_left1(x_+d, n_-d);
```

produces lexicographic order.

A modified function generates the cyclic permutations (permutations consisting of exactly one cycle of full length, see section 108). The only change is to skip the case $x = d$ in the loop:

```
        for (ulong k=d+1; k<n_; ++k)  // omit k==d
```

The cyclic permutations of five elements are shown in figure 10.13-B. The program [FXT: comb/perm-rec-demo.cc] was used to create the figures 10.13-A and 10.13-B.

```
void visit(const perm_rec &P)  // function to call with each permutation
{
    // Print the permutation
}
int
main(int argc, char **argv)
{
    ulong n = 5;  // Number of elements to permute
    bool cq = 1;  // Whether to generate only cyclic permutations
    perm_rec P(n);
    if ( cq )  P.generate_cyclic(visit);
    else       P.generate(visit);
```

	permutation	cycle	inverse	ffact-swp
0:	[1 2 3 4 .]	(0, 1, 2, 3, 4)	[4 . 1 2 3]	[1 1 1 1]
1:	[1 2 4 . 3]	(0, 1, 2, 4, 3)	[3 . 1 4 2]	[1 1 2 1]
2:	[1 3 . 4 2]	(0, 1, 3, 4, 2)	[2 . 4 1 3]	[1 2 1 1]
3:	[1 3 4 2 .]	(0, 1, 3, 2, 4)	[4 . 3 1 2]	[1 2 2 1]
4:	[1 4 3 . 2]	(0, 1, 4, 2, 3)	[3 . 4 2 1]	[1 3 1 1]
5:	[1 4 . 2 3]	(0, 1, 4, 3, 2)	[2 . 3 4 1]	[1 3 2 1]
6:	[2 . 3 4 1]	(0, 2, 3, 4, 1)	[1 4 . 2 3]	[2 1 1 1]
7:	[2 . 4 1 3]	(0, 2, 4, 3, 1)	[1 3 . 4 2]	[2 1 2 1]
8:	[2 3 1 4 .]	(0, 2, 1, 3, 4)	[4 2 . 1 3]	[2 2 1 1]
9:	[2 3 4 . 1]	(0, 2, 4, 1, 3)	[3 4 . 1 2]	[2 2 2 1]
10:	[2 4 3 1 .]	(0, 2, 3, 1, 4)	[4 3 . 2 1]	[2 3 1 1]
11:	[2 4 1 . 3]	(0, 2, 1, 4, 3)	[3 2 . 4 1]	[2 3 2 1]
12:	[3 2 . 4 1]	(0, 3, 4, 1, 2)	[2 4 1 . 3]	[3 1 1 1]
13:	[3 2 4 1 .]	(0, 3, 1, 2, 4)	[4 3 1 . 2]	[3 1 2 1]
14:	[3 . 1 4 2]	(0, 3, 4, 2, 1)	[1 2 4 . 3]	[3 2 1 1]
15:	[3 . 4 2 1]	(0, 3, 2, 4, 1)	[1 4 3 . 2]	[3 2 2 1]
16:	[3 4 . 1 2]	(0, 3, 1, 4, 2)	[2 3 4 . 1]	[3 3 1 1]
17:	[3 4 1 2 .]	(0, 3, 2, 1, 4)	[4 2 3 . 1]	[3 3 2 1]
18:	[4 2 3 . 1]	(0, 4, 1, 2, 3)	[3 4 1 2 .]	[4 1 1 1]
19:	[4 2 . 1 3]	(0, 4, 3, 1, 2)	[2 3 1 4 .]	[4 1 2 1]
20:	[4 3 1 . 2]	(0, 4, 2, 1, 3)	[3 2 4 1 .]	[4 2 1 1]
21:	[4 3 . 2 1]	(0, 4, 1, 3, 2)	[2 4 3 1 .]	[4 2 2 1]
22:	[4 . 3 1 2]	(0, 4, 2, 3, 1)	[1 3 4 2 .]	[4 3 1 1]
23:	[4 . 1 2 3]	(0, 4, 3, 2, 1)	[1 2 3 4 .]	[4 3 2 1]

Figure 10.13-B: All cyclic permutations of 5 elements and the permutations as cycles, their inverses, and their (swaps-) representations as mixed radix numbers with falling factorial basis (from left to right).

```
    return 0;
}
```

The routines generate about 57 million permutations and about 37 million cyclic permutations per second.

10.14 Minimal-change order for cyclic permutations

All cyclic permutations can be generated from a mixed radix Gray code with falling factorial base (see section 9.2 on page 210). Two successive permutations differ at three positions as shown in figure 10.14-A. An constant amortized time (CAT) implementation is [FXT: `class cyclic_perm` in comb/cyclic-perm.h]:

```
class cyclic_perm
{
public:
    mixedradix_gray *M_;
    ulong n_;   // number of elements to permute
    ulong *ix_;  // current permutation (of {0, 1, ..., n-1})
    ulong *x_;  // inverse permutation
public:
    cyclic_perm(ulong n)
        : n_(n)
    {
        ix_ = new ulong[n_];
        x_ = new ulong[n_];
        M_ = new mixedradix_gray(n_-2, 0);  // falling factorial base
        first();
    }
 [--snip--]
```

The computation of the successor uses the position and direction of the mixed radix digit changed with the last increment:

```
private:
    void setup()
    {
```

```
              permutation     fact.num.          cycle
      0:    [ 4 0 1 2 3 ]    [ . . . ]    ( 4, 3, 2, 1, 0, )
      1:    [ 3 4 1 2 0 ]    [ 1 . . ]    ( 4, 0, 3, 2, 1, )
      2:    [ 3 0 4 2 1 ]    [ 2 . . ]    ( 4, 1, 0, 3, 2, )
      3:    [ 3 0 1 4 2 ]    [ 3 . . ]    ( 4, 2, 1, 0, 3, )
      4:    [ 2 3 1 4 0 ]    [ 3 1 . ]    ( 4, 0, 2, 1, 3, )
      5:    [ 2 3 4 0 1 ]    [ 2 1 . ]    ( 4, 1, 3, 0, 2, )
      6:    [ 2 4 1 0 3 ]    [ 1 1 . ]    ( 4, 3, 0, 2, 1, )
      7:    [ 4 3 1 0 2 ]    [ . 1 . ]    ( 4, 2, 1, 3, 0, )
      8:    [ 4 0 3 1 2 ]    [ . 2 . ]    ( 4, 2, 3, 1, 0, )
      9:    [ 2 4 3 1 0 ]    [ 1 2 . ]    ( 4, 0, 2, 3, 1, )
     10:    [ 2 0 4 1 3 ]    [ 2 2 . ]    ( 4, 3, 1, 0, 2, )
     11:    [ 2 0 3 4 1 ]    [ 3 2 . ]    ( 4, 1, 0, 2, 3, )
     12:    [ 1 2 3 4 0 ]    [ 3 2 1 ]    ( 4, 0, 1, 2, 3, )
     13:    [ 1 2 4 0 3 ]    [ 2 2 1 ]    ( 4, 3, 0, 1, 2, )
     14:    [ 1 4 3 0 2 ]    [ 1 2 1 ]    ( 4, 2, 3, 0, 1, )
     15:    [ 4 2 3 0 1 ]    [ . 2 1 ]    ( 4, 1, 2, 3, 0, )
     16:    [ 4 3 0 2 1 ]    [ . 1 1 ]    ( 4, 1, 3, 2, 0, )
     17:    [ 1 4 0 2 3 ]    [ 1 1 1 ]    ( 4, 3, 2, 0, 1, )
     18:    [ 1 3 4 2 0 ]    [ 2 1 1 ]    ( 4, 0, 1, 3, 2, )
     19:    [ 1 3 0 4 2 ]    [ 3 1 1 ]    ( 4, 2, 0, 1, 3, )
     20:    [ 3 2 0 4 1 ]    [ 3 . 1 ]    ( 4, 1, 2, 0, 3, )
     21:    [ 3 2 4 1 0 ]    [ 2 . 1 ]    ( 4, 0, 3, 1, 2, )
     22:    [ 3 4 0 1 2 ]    [ 1 . 1 ]    ( 4, 2, 0, 3, 1, )
     23:    [ 4 2 0 1 3 ]    [ . . 1 ]    ( 4, 3, 1, 2, 0, )
```

Figure 10.14-A: All cyclic permutations of 5 elements in a minimal-change order.

```
        const ulong *fc = M_->data();
        for (ulong k=0; k<n_; ++k)  ix_[k] = k;

        for (ulong k=n_-1; k>1; --k)
        {
            ulong z = n_-3-(k-2); // 0, ..., n-3
            ulong i = fc[z];
            swap2(ix_[k], ix_[i]);
        }
        if ( n_>1 )  swap2(ix_[0], ix_[1]);

        make_inverse(ix_, x_, n_);
    }
public:
    void first()
    {
        M_->first();
        setup();
    }

    bool next()
    {
        if ( false == M_->next() )  { first();  return false; }
        ulong j = M_->pos();

        if ( j && (x_[0]==n_-1) )  // once in 2*n cases
        {
            setup();  // work proportional n
            // only 3 elements are interchanged
        }
        else  // easy case
        {
            int d = M_->dir();
            ulong x2 = (M_->data())[j];
            ulong x1 = x2 - d,  x3 = n_-1;
            ulong i1 = x_[x1], i2 = x_[x2], i3 = x_[x3];

            swap2(x_[x1], x_[x2]);
            swap2(x_[x1], x_[x3]);
            swap2(ix_[i1], ix_[i2]);
            swap2(ix_[i2], ix_[i3]);
        }
        return true;
```

```
   }
[--snip--]
```

The order so that the permutation is the same as if one would compute it via the function [FXT: `ffact2cyclic()` in comb/fact2cyclic.cc] which is given in section 10.3.4 on page 229. The listing in figure 10.14-A was created with the program [FXT: comb/cyclic-perm-demo.cc]. About 40 million permutations per second are generated.

10.15 Permutations with special properties

10.15.1 The number of certain permutations

We discuss permutations with special properties, such as involutions, derangements, and permutations with prescribed cycle types.

Permutations with m cycles: Stirling cycle numbers

n:	total	m= 1	2	3	4	4	6	7	8	9
1:	1	1								
2:	2	1	1							
3:	6	2	3	1						
4:	24	6	11	6	1					
5:	120	24	50	35	10	1				
6:	720	120	274	225	85	15	1			
7:	5040	720	1764	1624	735	175	21	1		
8:	40320	5040	13068	13132	6769	1960	322	28	1	
9:	362880	40320	109584	118124	67284	22449	4536	546	36	1

Figure 10.15-A: Stirling numbers of the first kind $s(n,m)$ (Stirling cycle numbers).

The number of permutations of n elements into m cycles is given by the (unsigned) *Stirling numbers of the first kind* (or *Stirling cycle numbers*) $s(n,m)$. The first few are shown in figure 10.15-A which was created with the program [FXT: comb/stirling1-demo.cc]. One has $s(1,1)=1$ and

$$s(n,m) \;=\; s(n-1,m-1) + (n-1)\,s(n-1,m) \tag{10.15-1}$$

$$\sum_{m=0}^{n} s(n,m)\,e^m \;=\; \prod_{m=0}^{n} e+m \;=\; e^{\overline{n}} \tag{10.15-2}$$

A generating function is given as relation 35.2-71a on page 673, see also entry A008275 of [214]. The Stirling numbers of the second kind (Stirling set numbers) are treated in section 15.1 on page 320. Many identities involving the Stirling numbers are given in [124, pp.243-253]. We note just a few, writing $S(n,k)$ for the Stirling set numbers:

$$x^n \;=\; \sum_{k=0}^{n} S(n,k)\,x^{\underline{k}} \;=\; \sum_{k=0}^{n} S(n,k)\,(-1)^{n-k}\,x^{\overline{k}} \tag{10.15-3a}$$

where $x^{\underline{k}} = x\,(x-1)\,(x-2)\cdots(x-k+1)$ and $x^{\overline{k}} = x\,(x+1)\,(x+2)\cdots(x+k-1)$.

$$x^{\underline{k}} \;=\; \sum_{k=0}^{n} s(n,k)\,(-1)^{n-k}\,x^k \tag{10.15-3b}$$

$$x^{\overline{k}} \;=\; \sum_{k=0}^{n} s(n,k)\,x^k \tag{10.15-3c}$$

Further [124, p.296], with $D := \frac{d}{dz}$ and $\vartheta = z\frac{d}{dz}$, we have the operator identities

$$\vartheta^n = \sum_{k=0}^{n} S(n,k)\, z^k\, D^k \tag{10.15-4a}$$

$$z^n\, D^n = \sum_{k=0}^{n} s(n,k)\, (-1)^{n-k}\, \vartheta^k \tag{10.15-4b}$$

Permutations with prescribed cycle type

A permutation of n elements is of *type* $C = [c_1, c_2, c_3, \ldots, c_n]$ if it has c_1 fixed points, c_2 cycles of length 2, c_3 cycles of length 3, and so on. The number $Z_{n,C}$ of permutations of n elements with type C equals

$$Z_{n,C} = n!\,/\,(c_1!\, c_2!\, c_3!\, \ldots\, c_n!\, 1^{c_1}\, 2^{c_2}\, 3^{c_3}\, \ldots\, n^{c_n}) \tag{10.15-5}$$

This relation is given in [87, p.233] which is a good source for identities of generating functions. We necessarily have $n = 1\,c_1 + 2\,c_2 + \ldots + n\,c_n$, that is, the c_j correspond to a integer partition of n.

Prefix conditions

```
   involutions          up-down          indecomposable       derangements
 1:  1 2 3 4       1:  1 3 2 4       1:  2 3 4 1       1:  2 1 4 3
 2:  1 2 4 3       2:  1 4 2 3       2:  2 4 1 3       2:  2 3 4 1
 3:  1 3 2 4       3:  2 3 1 4       3:  2 4 3 1       3:  2 4 1 3
 4:  1 4 3 2       4:  2 4 1 3       4:  3 1 4 2       4:  3 1 4 2
 5:  2 1 3 4       5:  3 4 1 2       5:  3 2 4 1       5:  3 4 1 2
 6:  2 1 4 3       #perm = 5         6:  3 4 1 2       6:  3 4 2 1
 7:  3 2 1 4                         7:  3 4 2 1       7:  4 1 2 3
 8:  3 4 1 2                         8:  4 1 2 3       8:  4 3 1 2
 9:  4 2 3 1                         9:  4 1 3 2       9:  4 3 2 1
10:  4 3 2 1                        10:  4 2 1 3       #perm = 9
#perm = 10                          11:  4 2 3 1
                                    12:  4 3 1 2
                                    13:  4 3 2 1
                                    #perm = 13
```

Figure 10.15-B: Examples of permutations subject to conditions on the prefixes. From left to right: involutions, up-down permutations, indecomposable permutations and derangements.

Some types of permutations can be generated efficiently by a routine that produces the lexicographically ordered list of permutations subject to conditions for all prefixes. The implementation (following [157]) is [FXT: `class perm_restrpref` in comb/perm-restrpref.h]. The condition (as a function pointer) has to be supplied upon creation of an instance of the class. The program [FXT: comb/perm-restrpref-demo.cc] demonstrates the usage, it can be used to generate all involutions, up-down, indecomposable, or derangement permutations, see figure 10.15-B..

Involutions

The sequence $I(n)$ of the number of involutions (self-inverse permutations) starts as ($n \geq 1$)

`1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152, 568504, 2390480, ...`

This is sequence A000085 of [214]. Compute $I(n)$ using the relation

$$I(n) = I(n-1) + (n-1)\, I(n-2) \tag{10.15-6}$$

```
N=20; v=vector(N);
v[1]=1; v[2]=2;
for(n=3,N,v[n]=v[n-1]+(n-1)*v[n-2]);
  [1, 2, 4, 10, 26, 76, ... ]
```

The exponential generating function is

$$\sum_{k=0}^{\infty} \frac{I(k)\,x^k}{k!} \quad = \quad \exp\left(x + x^2/2\right) \tag{10.15-7}$$

The relation is given in [247, p.85], as the special case $m = 2$ of the exponential generating function

$$\exp\left(\sum_{d\backslash m} x^d/d\right) \tag{10.15-8}$$

for the number permutations whose m-th power is identity.

The corresponding condition function is

```
bool cond_inv(const ulong *a, ulong k)
{
    ulong ak = a[k];
    if ( (ak<=k) && (a[ak]!=k) )  return false;
    return true;
}
```

Alternating permutations

The *alternating permutations* (or *up-down permutations*) satisfy $a_0 < a_1 > a_2 < a_3 > \ldots$. The condition function is

```
bool cond_updown(const ulong *a, ulong k)
// up-down condition: a1 < a2 > a3 < a4 > ...
{
    if ( k<2 ) return true;
    if ( (k%2) )  return ( a[k]<a[k-1] );
    else          return ( a[k]>a[k-1] );
}
```

Note that the routine is for the permutations of the elements $1, 2, \ldots, n$ in a one-based array.

The sequence $A(n)$ of the number of alternating permutations starts as ($n \geq 1$)

```
1, 1, 2, 5, 16, 61, 272, 1385, 7936, 50521, 353792, 2702765, 22368256, ...
```

It is sequence A000111 of [214], the sequence of the *Euler numbers*. The list can be computed using the relation

$$A(n) \quad = \quad \frac{1}{2} \sum_{k=0}^{n-1} \binom{n-1}{k} A(k)\,A(n-1-k) \tag{10.15-9}$$

```
N=20; v=vector(N+1);
v[0+1]=1;  v[1+1]=1;  v[2+1]=1;  \\ start with zero: v[x] == A(x-1)
for(n=3,N,v[n+1]=1/2*sum(k=0,n-1,binomial(n-1,k)*v[k+1]*v[n-1-k+1]));  v
  [1, 1, 1, 2, 5, 16, 61, 272, ... ]
```

Indecomposable permutations

The *indecomposable* (or *connected*) permutations satisfy, for $k = 0, 1, \ldots, n-1$, the inequality of sets

$$\{a_0,\, a_1,\, \ldots,\, a_k\} \quad \neq \quad \{0,\, 1,\, \ldots,\, k\} \tag{10.15-10}$$

The condition function is

```
ulong N;  // set to n in main()
bool cond_indecomp(const ulong *a, ulong k)
// indecomposable condition: {a1,...,ak} != {1,...,k} for all k<n
{
```

```
    if ( k==N )  return true;
    for (ulong i=1; i<=k; ++i)  if ( a[i]>k )  return true;
    return false;
}
```

The sequence $C(n)$ of the number of indecomposable permutations starts as ($n \geq 1$)

```
1, 1, 3, 13, 71, 461, 3447, 29093, 273343, 2829325, 31998903, 392743957, ...
```

This is sequence A003319 of [214]. Compute $C(n)$ using

$$C(n) \quad = \quad n! - \sum_{k=1}^{n-1} k!\, C(n-k) \tag{10.15-11}$$

```
N=20; v=vector(N);
for(n=1,N,v[n]=n!-sum(k=1,n-1,k!*v[n-k]));  v
  [1, 1, 3, 13, 71, 461, 3447, ... ]
```

Derangements

A permutation is a *derangement* if $a_k \neq k$ for all k:

```
bool cond_derange(const ulong *a, ulong k)
// derangement condition: f[k]!=k for all k
{
    return ( a[k]!=k );
}
```

The sequence $D(n)$ of the number of derangements starts as ($n \geq 1$)

```
0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570, 176214841, ...
```

This is sequence A000166 of [214], the *subfactorial numbers*. Compute $D(n)$ using either of

$$D(n) \quad = \quad (n-1)\,[D(n-1) + D(n-2)] \tag{10.15-12a}$$

$$D(n) \quad = \quad n\,D(n-1) + (-1)^n \tag{10.15-12b}$$

$$D(n) \quad = \quad \sum_{k=0}^{n} (-1)^{n-k}\, \frac{n!}{(n-k)!} \tag{10.15-12c}$$

$$D(n) \quad = \quad \lfloor (n! + 1/2)/e \rfloor \tag{10.15-12d}$$

where $e = \exp(1)$. We use the recursion 10.15-12a:

```
N=20;  v=vector(N);  v[1]=0;  v[2]=1;
for(n=3,N,v[n]=(n-1)*(v[n-1]+v[n-2]));  v
  [0, 1, 2, 9, 44, 265, 1854, 14833, ... ]
```

10.15.2 Permutations with distance restrictions

We present constructions for Gray codes for permutations with certain restrictions. These are computed from Gray codes of mixed radix numbers with factorial basis. We write $p(k)$ for the position of the element k in a given permutation.

Permutations where $p(k) \leq k+1$

Let $M(n)$ the number of permutations of n elements where no element can move more than one place to the right. We have $M(n) = 2^{n-1}$. A Gray code for these permutation is shown in figure 10.15-C which was created with the program [FXT: comb/perm-right1-gray-demo.cc]. $M(n)$ also counts the permutations that start as a rising sequence (ending in the maximal element) and end as a falling sequence. The recursion for the array in the leftmost column of figure 10.15-C can be obtained by the recursion

```
              ffact        perm            inv. perm       ffact(inv)
       1:     . 3 . .    [ 0 4 1 2 3 ]    [ 0 2 3 4 1 ]     . 1 1 1
       2:     . 2 . .    [ 0 3 1 2 4 ]    [ 0 2 3 1 4 ]     . 1 1 .
       3:     . 1 . .    [ 0 2 1 3 4 ]    [ 0 2 1 3 4 ]     . 1 . .
       4:     . 1 . 1    [ 0 2 1 4 3 ]    [ 0 2 1 4 3 ]     . 1 . 1
       5:     . . . 1    [ 0 1 2 4 3 ]    [ 0 1 2 4 3 ]     . . . 1
       6:     . . . .    [ 0 1 2 3 4 ]    [ 0 1 2 3 4 ]     . . . .
       7:     . . 1 .    [ 0 1 3 2 4 ]    [ 0 1 3 2 4 ]     . . 1 .
       8:     . . 2 .    [ 0 1 4 2 3 ]    [ 0 1 3 4 2 ]     . . 1 1
       9:     1 . 2 .    [ 1 0 4 2 3 ]    [ 1 0 3 4 2 ]     1 . 1 1
      10:     1 . 1 .    [ 1 0 3 2 4 ]    [ 1 0 3 2 4 ]     1 . 1 .
      11:     1 . . .    [ 1 0 2 3 4 ]    [ 1 0 2 3 4 ]     1 . . .
      12:     1 . . 1    [ 1 0 2 4 3 ]    [ 1 0 2 4 3 ]     1 . . 1
      13:     2 . . 1    [ 2 0 1 4 3 ]    [ 1 2 0 4 3 ]     1 1 . 1
      14:     2 . . .    [ 2 0 1 3 4 ]    [ 1 2 0 3 4 ]     1 1 . .
      15:     3 . . .    [ 3 0 1 2 4 ]    [ 1 2 3 0 4 ]     1 1 1 .
      16:     4 . . .    [ 4 0 1 2 3 ]    [ 1 2 3 4 0 ]     1 1 1 1
```

Figure 10.15-C: Gray code for the permutations of 5 elements where no element lies more than one place to the right of its position in the identical permutation.

```
void Y_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( z )  // forward:
        {
            // words 0, 10, 200, 3000, 40000, ...
            ulong k = 0;
            do
            {
                ff[d] = k;
                Y_rec(d+k+1, !z);
            }
            while ( ++k <= (n-d) );
        }
        else      // backward:
        {
            // words ..., 40000, 3000, 200, 10, 0
            ulong k = n-d+1;
            do
            {
                --k;
                ff[d] = k;
                Y_rec(d+k+1, !z);
            }
            while ( k != 0 );
        }
    }
}
```

The array `ff` (of length `n`) must be initialized with zeros and the initial call is `Y_rec(0, true);`. About 85 million words per second are generated. In the inverse permutations (where no element is more than one place left of its original position) the swaps are adjacent end their position is determined by the ruler function. Thereby the inverse permutations can be generated using [FXT: `class ruler_func` in comb/ruler-func.h], described in section 8.2.3 on page 196.

Permutations where $k-1 \le p(k) \le k+1$

Let $F(n)$ the number of permutations of n elements where no element can move more than one place to the left. Then $F(n)$ is the $(n+1)$-st Fibonacci number. A Gray code for these permutation is shown in figure 10.15-D which was created with the program [FXT: comb/perm-dist1-gray-demo.cc].

	ffact	perm
1:	1 . . 1 . .	[1 0 2 4 3 5 6]
2:	1 . . 1 . 1	[1 0 2 4 3 6 5]
3:	1 1	[1 0 2 3 4 6 5]
4:	1	[1 0 2 3 4 5 6]
5:	1 . . . 1 .	[1 0 2 3 5 4 6]
6:	1 . 1 . 1 .	[1 0 3 2 5 4 6]
7:	1 . 1 . . .	[1 0 3 2 4 5 6]
8:	1 . 1 . . 1	[1 0 3 2 4 6 5]
9:	. . 1 . . 1	[0 1 3 2 4 6 5]
10:	. . 1 . . .	[0 1 3 2 4 5 6]
11:	. . 1 . 1 .	[0 1 3 2 5 4 6]
12:	. . . . 1 .	[0 1 2 3 5 4 6]
13:	. . . . . .	[0 1 2 3 4 5 6]
14:	 1	[0 1 2 3 4 6 5]
15:	. . . 1 . 1	[0 1 2 4 3 6 5]
16:	. . . 1 . .	[0 1 2 4 3 5 6]
17:	. 1 . 1 . .	[0 2 1 4 3 5 6]
18:	. 1 . 1 . 1	[0 2 1 4 3 6 5]
19:	. 1 . . . 1	[0 2 1 3 4 6 5]
20:	. 1	[0 2 1 3 4 5 6]
21:	. 1 . . 1 .	[0 2 1 3 5 4 6]

Figure 10.15-D: Gray code for the permutations of 7 elements where no element lies more than one place away from its position in the identical permutation. The permutations are self-inverse.

	ffact	perm	inv. perm	ffact(inv)
1:	1 1 . . 1	[1 2 0 3 5 4]	[2 0 1 3 5 4]	2 . . . 1
2:	1 1 . . .	[1 2 0 3 4 5]	[2 0 1 3 4 5]	2 . . . .
3:	1 1 . 1 .	[1 2 0 4 3 5]	[2 0 1 4 3 5]	2 . . 1 .
4:	1 1 . 1 1	[1 2 0 4 5 3]	[2 0 1 5 3 4]	2 . . 2 .
5:	1 . . 1 1	[1 0 2 4 5 3]	[1 0 2 5 3 4]	1 . . 2 .
6:	1 . . 1 .	[1 0 2 4 3 5]	[1 0 2 4 3 5]	1 . . 1 .
7:	1	[1 0 2 3 4 5]	[1 0 2 3 4 5]	1 . . . .
8:	1 . . . 1	[1 0 2 3 5 4]	[1 0 2 3 5 4]	1 . . . 1
9:	1 . 1 . 1	[1 0 3 2 5 4]	[1 0 3 2 5 4]	1 . 1 . 1
10:	1 . 1 . .	[1 0 3 2 4 5]	[1 0 3 2 4 5]	1 . 1 . .
11:	1 . 1 1 .	[1 0 3 4 2 5]	[1 0 4 2 3 5]	1 . 2 . .
12:	. . 1 1 .	[0 1 3 4 2 5]	[0 1 4 2 3 5]	. . 2 . .
13:	. . 1 . .	[0 1 3 2 4 5]	[0 1 3 2 4 5]	. . 1 . .
14:	. . 1 . 1	[0 1 3 2 5 4]	[0 1 3 2 5 4]	. . 1 . 1
15:	 1	[0 1 2 3 5 4]	[0 1 2 3 5 4]	 1
16:	. . . . .	[0 1 2 3 4 5]	[0 1 2 3 4 5]	. . . . .
17:	. . . 1 .	[0 1 2 4 3 5]	[0 1 2 4 3 5]	. . . 1 .
18:	. . . 1 1	[0 1 2 4 5 3]	[0 1 2 5 3 4]	. . . 2 .
19:	. 1 . 1 1	[0 2 1 4 5 3]	[0 2 1 5 3 4]	. 1 . 2 .
20:	. 1 . 1 .	[0 2 1 4 3 5]	[0 2 1 4 3 5]	. 1 . 1 .
21:	. 1 . . .	[0 2 1 3 4 5]	[0 2 1 3 4 5]	. 1 . . .
22:	. 1 . . 1	[0 2 1 3 5 4]	[0 2 1 3 5 4]	. 1 . . 1
23:	. 1 1 . 1	[0 2 3 1 5 4]	[0 3 1 2 5 4]	. 2 . . 1
24:	. 1 1 . .	[0 2 3 1 4 5]	[0 3 1 2 4 5]	. 2 . . .

Figure 10.15-E: Gray code for the permutations of 6 elements where no element lies more than one place to the left or two places to the right of its position in the identical permutation.

Permutations where $k-1 \le p(k) \le k+d$

A Gray code for the permutations where no element lies more than one place to the left or d places to the right of its original position can be obtained via the Gray codes for binary words with at most d successive ones given in section 12.2 on page 284. Figure 10.15-E shows the permutations of 6 elements with $d = 2$, it was created with the program [FXT: comb/perm-l1r2-gray-demo.cc]. The array shown leftmost in figure 10.15-E can be generated via the recursion

```
void Y_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        const ulong w = n-d;
        if ( z )
        {
            if ( w>1 )  { ff[d]=1; ff[d+1]=1; ff[d+2]=0;  Y_rec(d+3, !z); }
            ff[d]=1; ff[d+1]=0;  Y_rec(d+2, !z);
            ff[d]=0;  Y_rec(d+1, !z);
        }
        else
        {
            ff[d]=0;  Y_rec(d+1, !z);
            ff[d]=1; ff[d+1]=0;  Y_rec(d+2, !z);
            if ( w>1 )  { ff[d]=1; ff[d+1]=1; ff[d+2]=0; Y_rec(d+3, !z); }
        }
    }
}
```

If the two lines starting `if ( w>1 )` are omitted then the Fibonacci words are obtained. About 110 million words per second are generated.

Chapter 11

Subsets and permutations of a multiset

11.1 Subsets of a multiset

```
n == 630
primes    = [   2   3   5   7 ]
exponents = [   1   2   1   1 ]

           d      auxiliary products          exponents    change @
 1:        1     [   1   1   1   1   1 ]     [ . . . . ]       4
 2:        2     [   2   1   1   1   1 ]     [ 1 . . . ]       0
 3:        3     [   3   3   1   1   1 ]     [ . 1 . . ]       1
 4:        6     [   6   3   1   1   1 ]     [ 1 1 . . ]       0
 5:        9     [   9   9   1   1   1 ]     [ . 2 . . ]       1
 6:       18     [  18   9   1   1   1 ]     [ 1 2 . . ]       0
 7:        5     [   5   5   5   1   1 ]     [ . . 1 . ]       2
 8:       10     [  10   5   5   1   1 ]     [ 1 . 1 . ]       0
 9:       15     [  15  15   5   1   1 ]     [ . 1 1 . ]       1
10:       30     [  30  15   5   1   1 ]     [ 1 1 1 . ]       0
11:       45     [  45  45   5   1   1 ]     [ . 2 1 . ]       1
12:       90     [  90  45   5   1   1 ]     [ 1 2 1 . ]       0
13:        7     [   7   7   7   7   1 ]     [ . . . 1 ]       3
14:       14     [  14   7   7   7   1 ]     [ 1 . . 1 ]       0
15:       21     [  21  21   7   7   1 ]     [ . 1 . 1 ]       1
16:       42     [  42  21   7   7   1 ]     [ 1 1 . 1 ]       0
17:       63     [  63  63   7   7   1 ]     [ . 2 . 1 ]       1
18:      126     [ 126  63   7   7   1 ]     [ 1 2 . 1 ]       0
19:       35     [  35  35  35   7   1 ]     [ . . 1 1 ]       2
20:       70     [  70  35  35   7   1 ]     [ 1 . 1 1 ]       0
21:      105     [ 105 105  35   7   1 ]     [ . 1 1 1 ]       1
22:      210     [ 210 105  35   7   1 ]     [ 1 1 1 1 ]       0
23:      315     [ 315 315  35   7   1 ]     [ . 2 1 1 ]       1
24:      630     [ 630 315  35   7   1 ]     [ 1 2 1 1 ]       0
```

Figure 11.1-A: Divisors of $630 = 2^1 \cdot 3^2 \cdot 5^1 \cdot 7^1$ generated as subsets of the multiset of exponents.

A *multiset* is set where elements can be repeated. A subset of a set of n elements can be identified with the bits of all n-bit binary words. The subsets of a multiset can be obtained as mixed radix numbers: if the j-th element is repeated r_j times then the radix of digit j has to be $r_j + 1$. Thereby all methods of chapter 9 on page 207 can be applied.

As an example, all divisors of a number x whose factorization $x = p_0^{e_0} \cdot p_1^{e_1} \cdots p_{n-1}^{e_{n-1}}$ is known can be obtained via the length-n mixed radix numbers with radices $[e_0 + 1, e_1 + 1, \ldots, e_{n-1} + 1]$. The implementation [FXT: `class divisors` in mod/divisors.h] generates the subsets of the exponent-multiset

in counting order (figure 11.1-A shows the data for $x = 630$). An auxiliary array T of products is updated with each step: if the changed digit (at position j) became 1 then set $t := T_{j+1} \cdot p_j$, else set $t := T_j \cdot p_j$. Set $T_i = t$ for all $0 \le i \le j$. A sentinel element $T_n = 1$ avoids unnecessary code. Figure 11.1-A was created with the program [FXT: mod/divisors-demo.cc]. The computation of all products of k out of n given factors is described in section 6.1.2 on page 168.

11.2 Permutations of a multiset

```
        (2, 2, 1)                (6, 2)                (1, 1, 1, 1)
 1:  [ . . 1 1 2 ]       1:  [ . . . . . . 1 1 ]       1:  [ . 1 2 3 ]
 2:  [ . . 1 2 1 ]       2:  [ . . . . . 1 . 1 ]       2:  [ . 1 3 2 ]
 3:  [ . . 2 1 1 ]       3:  [ . . . . . 1 1 . ]       3:  [ . 2 1 3 ]
 4:  [ . 1 . 1 2 ]       4:  [ . . . . 1 . . 1 ]       4:  [ . 2 3 1 ]
 5:  [ . 1 . 2 1 ]       5:  [ . . . . 1 . 1 . ]       5:  [ . 3 1 2 ]
 6:  [ . 1 1 . 2 ]       6:  [ . . . . 1 1 . . ]       6:  [ . 3 2 1 ]
 7:  [ . 1 1 2 . ]       7:  [ . . . 1 . . . 1 ]       7:  [ 1 . 2 3 ]
 8:  [ . 1 2 . 1 ]       8:  [ . . . 1 . . 1 . ]       8:  [ 1 . 3 2 ]
 9:  [ . 1 2 1 . ]       9:  [ . . . 1 . 1 . . ]       9:  [ 1 2 . 3 ]
10:  [ . 2 . 1 1 ]      10:  [ . . . 1 1 . . . ]      10:  [ 1 2 3 . ]
11:  [ . 2 1 . 1 ]      11:  [ . . 1 . . . . 1 ]      11:  [ 1 3 . 2 ]
12:  [ . 2 1 1 . ]      12:  [ . . 1 . . . 1 . ]      12:  [ 1 3 2 . ]
13:  [ 1 . . 1 2 ]      13:  [ . . 1 . . 1 . . ]      13:  [ 2 . 1 3 ]
14:  [ 1 . . 2 1 ]      14:  [ . . 1 . 1 . . . ]      14:  [ 2 . 3 1 ]
15:  [ 1 . 1 . 2 ]      15:  [ . . 1 1 . . . . ]      15:  [ 2 1 . 3 ]
16:  [ 1 . 1 2 . ]      16:  [ . 1 . . . . . 1 ]      16:  [ 2 1 3 . ]
17:  [ 1 . 2 . 1 ]      17:  [ . 1 . . . . 1 . ]      17:  [ 2 3 . 1 ]
18:  [ 1 . 2 1 . ]      18:  [ . 1 . . . 1 . . ]      18:  [ 2 3 1 . ]
19:  [ 1 1 . . 2 ]      19:  [ . 1 . . 1 . . . ]      19:  [ 3 . 1 2 ]
20:  [ 1 1 . 2 . ]      20:  [ . 1 . 1 . . . . ]      20:  [ 3 . 2 1 ]
21:  [ 1 1 2 . . ]      21:  [ . 1 1 . . . . . ]      21:  [ 3 1 . 2 ]
22:  [ 1 2 . . 1 ]      22:  [ 1 . . . . . . 1 ]      22:  [ 3 1 2 . ]
23:  [ 1 2 . 1 . ]      23:  [ 1 . . . . . 1 . ]      23:  [ 3 2 . 1 ]
24:  [ 1 2 1 . . ]      24:  [ 1 . . . . 1 . . ]      24:  [ 3 2 1 . ]
25:  [ 2 . . 1 1 ]      25:  [ 1 . . . 1 . . . ]
26:  [ 2 . 1 . 1 ]      26:  [ 1 . . 1 . . . . ]
27:  [ 2 . 1 1 . ]      27:  [ 1 . 1 . . . . . ]
28:  [ 2 1 . . 1 ]      28:  [ 1 1 . . . . . . ]
29:  [ 2 1 . 1 . ]
30:  [ 2 1 1 . . ]
```

Figure 11.2-A: Permutations of multisets in lexicographic order: the multiset $(2, 2, 1)$ (left), $(6, 2)$ (combinations $\binom{6+2}{2}$), middle), and $(1, 1, 1, 1)$ (permutations of four elements, right). Dots denote zeros.

We write $(r_0, r_1, \ldots, r_{k-1})$ for a multiset with r_0 elements of the first sort, r_1 of the second sort, ..., r_{k-1} elements of the k-th sort. The total number of elements is $n = \sum_{j=0}^{k-1} r_k$. For the elements of the j-th sort we always use the number j. The number of permutations $P(r_0, r_1, \ldots, r_{k-1})$ of the multiset $(r_0, r_1, \ldots, r_{k-1})$ is a *multinomial coefficient*:

$$P(r_0, r_1, \ldots, r_{k-1}) \;=\; \binom{n}{r_0,\, r_1,\, r_2, \ldots,\, r_{k-1}} \;=\; \frac{n!}{r_0! \cdots r_{k-1}!} \tag{11.2-1a}$$

$$= \binom{n}{r_0} \binom{n-r_0}{r_1} \binom{n-r_0-r_1}{r_2} \cdots \binom{r_{k-3}+r_{k-2}+r_{k-1}}{r_{k-3}} \binom{r_{k-2}+r_{k-1}}{r_{k-2}} \tag{11.2-1b}$$

Relation 11.2-1a is obtained by observing that among the $n!$ ways to arrange all n elements $r_0!$ permutations of the first sort of elements, $r_1!$ of the second, and so on, lead to identical permutations.

11.2.1 Recursive generation

Let $[r_0, r_1, r_2, \ldots, r_{k-1}]$ denote the list of all permutations of the multiset $(r_0, r_1, r_2, \ldots, r_{k-1})$. The recursion

$$[r_0, r_1, r_2, \ldots, r_{k-1}] \;=\; \begin{matrix} r_0 \,.\, [r_0 - 1, r_1, r_2, \ldots, r_{k-1}] \\ r_1 \,.\, [r_0, r_1 - 1, r_2, \ldots, r_{k-1}] \\ r_2 \,.\, [r_0, r_1, r_2 - 1, \ldots, r_{k-1}] \\ \vdots \\ r_{k-1} \,.\, [r_0, r_1, r_2, \ldots, r_{k-1} - 1] \end{matrix} \qquad (11.2\text{-}2)$$

is used to obtain the following procedure [FXT: comb/mset-lex-rec-demo.cc]:

```
ulong n;    // number of objects
ulong *ms;  // multiset data in ms[0], ..., ms[n-1]
ulong k;    // number of different sorts of objects
ulong *r;   // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]
```

With the recursion

```
void mset_rec(ulong d)
{
    if ( d>=n )  visit();
    else
    {
        for (ulong j=0; j<k; ++j)  // for all buckets
        {
            ++wct;
            if ( r[j] )    // bucket has elements left
            {
                ++rct;
                --r[j];     // take element from bucket
                ms[d] = j;  // put element in place
                mset_rec(d+1);  // recursion
                ++r[j];     // put element back
            }
        }
    }
}
```

and the initial call `mset_rec(0)` we generate all multiset permutations in lexicographic order. As given the routine is inefficient when used with (many) small numbers r_j. An extreme case is $r_j = 1$ for all j, corresponding to the (regular) permutations: we have $n = k$ and for the $n!$ permutations the work is proportional to n^n. The method can be made efficient by maintaining a list of pointer to the next nonzero 'bucket' `nk[]` [FXT: `class mset_lex_rec` in comb/mset-lex-rec.h]:

```
class mset_lex_rec
{
public:
    ulong k_;    // number of different sorts of objects
    ulong *r_;   // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]
    ulong n_;    // number of objects
    ulong *ms_;  // multiset data in ms[0], ..., ms[n-1]
    ulong *nn_;  // position of next nonempty bucket
    void (*visit_)(const mset_lex_rec &);  // function to call with each permutation
    ulong ct_;  // count objects
    ulong rct_; // count recursions (==work)
  [--snip--]
```

The initializer takes as arguments an array of multiplicities and its length:

```
public:
    mset_lex_rec(ulong *r, ulong k)
    {
        k_ = k;
        r_ = new ulong[k];
        for (ulong j=0; j<k_; ++j)  r_[j] = r[j];  // get buckets
        n_ = 0;
        for (ulong j=0; j<k_; ++j)  n_ += r_[j];
        ms_ = new ulong[n_];
```

```
        nn_ = new ulong[k_+1];  // incl sentinel
        for (ulong j=0; j<k_; ++j)  nn_[j] = j+1;
        nn_[k] = 0;  // pointer to first nonempty bucket
    }
  [--snip--]
```

The method to generate all permutations takes a 'visit' function as argument:

```
    void generate(void (*visit)(const mset_lex_rec &))
    {
        visit_ = visit;
        ct_ = 0;
        rct_ = 0;
        mset_rec(0);
    }
private:
    void mset_rec(ulong d);
};
```

The recursion itself is [FXT: comb/mset-lex-rec.cc]:

```
void mset_lex_rec::mset_rec(ulong d)
{
    if ( d>=n_ )
    {
        ++ct_;
        visit_( *this );
    }
    else
    {
        for (ulong jf=k_, j=nn_[jf];  j<k_;  jf=j, j=nn_[j])  // for all nonempty buckets
        {
            ++rct_;  // work == number of recursions
            --r_[j];     // take element from bucket
            ms_[d] = j;  // put element in place
            if ( r_[j]==0 )  // bucket now empty?
            {
                ulong f = nn_[jf]; // where we come from
                nn_[jf] = nn_[j];  // let recursions skip over j
                mset_rec(d+1);     // recursion
                nn_[jf] = f;       // remove skip
            }
            else  mset_rec(d+1);   // recursion
            ++r_[j];      // put element back
        }
    }
}
```

Note that the test whether the current bucket is nonempty is omitted as empty buckets are skipped. Now the work involved with (regular) permutations is (less than) $e = 2.71828\ldots$ times the number of the generated permutations. Usage of the class is shown in [FXT: comb/mset-lex-rec2-demo.cc]. The permutations of 12 elements are generated at a rate of about 25 million per second, the combinations $\binom{30}{15}$ at about 40 million per second, and the permutations of $(2, 2, 2, 3, 3, 3)$ at about 20 million per second.

11.2.2 Iterative generation

The algorithm to generate the next permutation in lexicographic order given in section 10.1 on page 220 can be adapted for an iterative method for multiset permutations [FXT: `class mset_lex` in comb/mset-lex.h]:

```
class mset_lex
{
public:
    ulong k_;    // number of different sorts of objects
    ulong *r_;   // number of elements '0' in r[0], '1' in r[1], ..., 'k-1' in r[k-1]
    ulong n_;    // number of objects
    ulong *ms_;  // multiset data in ms[0], ..., ms[n-1], sentinel at [-1]
public:
    mset_lex(const ulong *r, ulong k)
```

```
    {
        k_ = k;
        r_ = new ulong[k];
        for (ulong j=0; j<k_; ++j)  r_[j] = r[j];  // get buckets
        n_ = 0;
        for (ulong j=0; j<k_; ++j)  n_ += r_[j];
        ms_ = new ulong[n_+1];
        ms_[0] = 0;  // sentinel
        ++ms_;  // nota bene
        first();
    }
    void first()
    {
        for (ulong j=0, i=0;  j<k_;  ++j)
            for (ulong h=r_[j];  h!=0;  --h, ++i)  ms_[i] = j;
    }
  [--snip--]
```

The only change in the update routine is to replace the operators `>` by `>=` in the scanning loops:

```
    bool next()
    {
        // find for rightmost pair with p_[i] < p_[i+1]:
        const ulong n1 = n_ - 1;
        ulong i = n1;
        do  { --i; }  while ( ms_[i] >= ms_[i+1] );  // can touch sentinel
        if ( (long)i<0 )  return false;  // last sequence is falling seq.
        // find rightmost element p[j] smaller than p[i]:
        ulong j = n1;
        while ( ms_[i] >= ms_[j] )  { --j; }
        swap2(ms_[i], ms_[j]);
        // Here the elements p[i+1], ..., p[n-1] are a falling sequence.
        // Reverse order to the right:
        ulong r = n1;
        ulong s = i + 1;
        while ( r > s )  { swap2(ms_[r], ms_[s]);  --r;  ++s; }
        return true;
    }
}
```

Usage of the class is shown in [FXT: comb/mset-lex-demo.cc]:

```
    ulong ct = 0;
    do
    {
        // visit
    }
    while ( P.next() );
```

The permutations of 12 elements are generated at a rate of about 110 million per second, the combinations $\binom{30}{15}$ at about 60 million per second, and the permutations of $(2,2,2,3,3,3)$ at about 82 million per second.

Chapter 12

Gray codes for strings with restrictions

We give constructions for Gray codes for strings with certain restrictions, such as forbidding two successive zeros or nonzero digits. The constraints considered are so that the number of strings of a given type satisfies a linear recursion with constant coefficients.

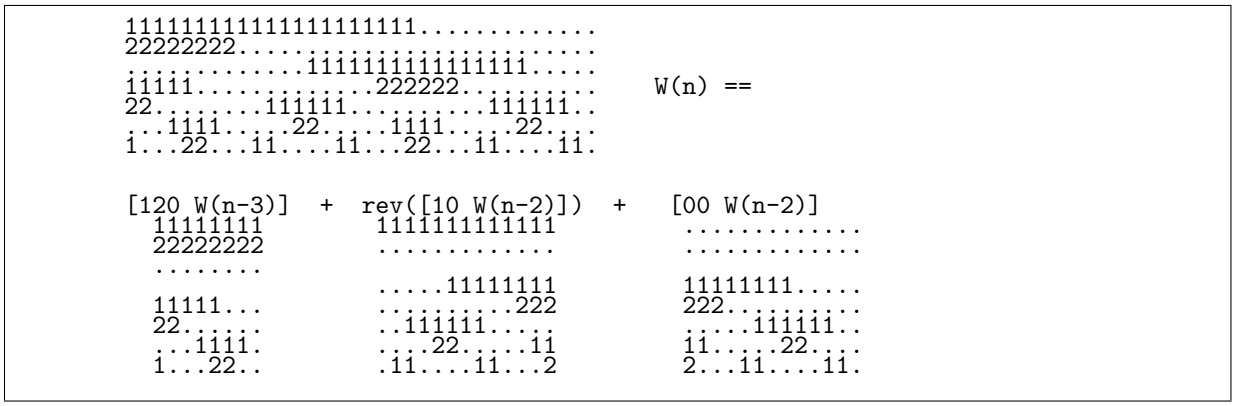


Figure 12.0-A: Obtaining a Gray code by a sublist recursion.

The algorithms are given as *list recursions*. For example, write $W(n)$ for the list of n-digit words (of a certain type), write $W^{\mathbf{R}}(n)$ for the reversed list, and $[x\,.\,W(n)]$ for the list with the word x prepended at each word. The recursion for a Gray code is

$$W(n) \;=\; \begin{matrix}[0\,0\,.\,W(n-2)\;\;] \\ [1\,0\,.\,W^{\mathbf{R}}(n-2)] \\ [1\,2\,0\,.\,W(n-3)]\end{matrix} \tag{12.0-1}$$

Such a relation always implies a backward version which is obtained by reversing the order of the sublists on the right hand side and additionally reversing each sublist

$$W^{\mathbf{R}}(n) \;=\; \begin{matrix}[1\,2\,0\,.\,W^{\mathbf{R}}(n-3)] \\ [1\,0\,.\,W(n-2)\;\;\;] \\ [0\,0\,.\,W^{\mathbf{R}}(n-2)\;\;] \end{matrix} \tag{12.0-2}$$

The construction is illustrated in figure 12.0-A. An implementation of the algorithm is [FXT: comb/fib-alt-gray-demo.cc]:

```
void X_rec(ulong d, bool z)
{
    if ( d>=n )
    {
        if ( d<=n+1 )  // avoid duplicates
        {
            visit();
        }
    }
    else
    {
        if ( z )
        {
            rv[d]=0;  rv[d+1]=0;  X_rec(d+2, z);
            rv[d]=1;  rv[d+1]=0;  X_rec(d+2,  ! z);
            rv[d]=1;  rv[d+1]=2;  rv[d+2]=0;  X_rec(d+3, z);
        }
        else
        {
            rv[d]=1;  rv[d+1]=2;  rv[d+2]=0;  X_rec(d+3, z);
            rv[d]=1;  rv[d+1]=0;  X_rec(d+2,  ! z);
            rv[d]=0;  rv[d+1]=0;  X_rec(d+2, z);
        }
    }
}
```

The initial call is `X_rec(0, 0);`. The parameter `z` determines whether the list is generated in forward or backward order. No optimizations are made as these tend to obscure the idea. Here we could omit one statement `rv[d]=1;`, replace the arguments `z` and `!z` in the recursive calls by constants, and of course create an iterative version.

The number $w(n)$ of words $W(n)$ is determined by a recursion (and some initial values $w(n)$) that can be obtained by counting the size of the list on both sides of the recursion relation 12.0-1 on the preceding page:

$$w(n) \quad = \quad 2\,w(n-2) + w(n-3) \tag{12.0-3}$$

One can typically set $w(0) = 1$, there is one empty list and this satisfies all conditions. The numbers $w(n)$ are in fact the Fibonacci numbers.

12.1 Fibonacci words

A recursive routine to generate the Fibonacci words (binary words not containing two consecutive ones) can be given as follows:

```
ulong n;    // number of bits in words
ulong *rv;  // bits of the word

void fib_rec(ulong d)
{
    if ( d>=n )  visit();
    else
    {
         rv[d]=0;  fib_rec(d+1);
         rv[d]=1;  rv[d+1]=0;  fib_rec(d+2);
    }
}
```

We allocate one extra element to avoid if-statements in the code:

```
int main()
{
    n = 7;
    rv = new ulong[n+1];
    fib_rec(0);
    return 0;
}
```

The output (assuming `visit()` simply prints the array) is given in the left of figure 12.1-A. Note that with the n-bit Fibonacci Gray code the number of ones in the first and last, second and second-last, etc. tracks are equal. Thereby the sequence of reversed words is also a Fibonacci Gray code. It turns

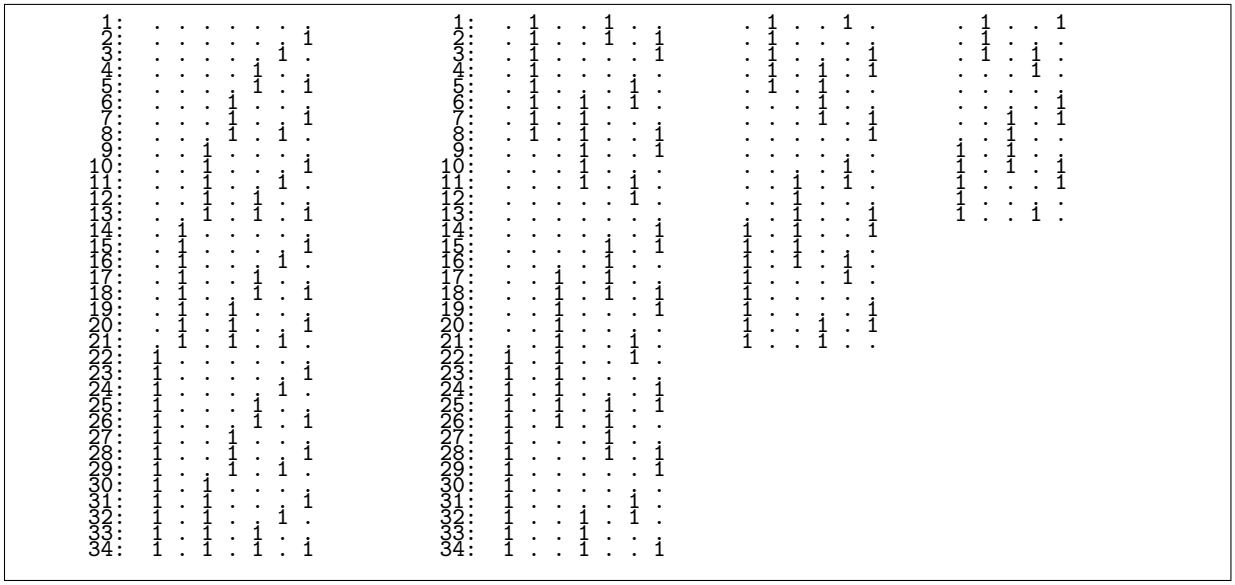


Figure 12.1-A: The first 34 Fibonacci words in counting order (left), and Gray codes through the first 34, 21, and 13 Fibonacci words (right). Dots are used for zeros.

out that a simple modification of the routine generates a Gray code through the Fibonacci words [FXT: comb/fibgray-rec-demo.cc]:

```
void fib_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        z = !z;  // change direction for Gray code
        if ( z )
        {
            rv[d]=0;  fib_rec(d+1, z);
            rv[d]=1;  rv[d+1]=0;  fib_rec(d+2, z);
        }
        else
        {
            rv[d]=1;  rv[d+1]=0;  fib_rec(d+2, z);
            rv[d]=0;  fib_rec(d+1, z);
        }
    }
}
```

The variable z controls the direction in the recursion, it is changed unconditionally with each step. The `if-else` blocks can be merged into

```
        rv[d]=!z;  rv[d+1]= z;  fib_rec(d+1+!z, z);
        rv[d]= z;  rv[d+1]=!z;  fib_rec(d+1+ z, z);
```

The algorithm is CAT (constant amortized time) and fast in practice, about 80 million objects are generated per second. A bit-level algorithm is given in section 1.29.2 on page 73.

The algorithm for the list of the length-n Fibonacci words $F(n)$ can be given as a recursion:

$$F(n) \;=\; \begin{matrix}[1\,0\,.\,F^{\mathbf{R}}(n-2)] \\ [0\,.\,F^{\mathbf{R}}(n-1)\;\;]\end{matrix} \tag{12.1-1}$$

Merging two steps we find

$$F(n) \;=\; \begin{matrix}[1\,0\,0\,.\,F(n-3)\;\;] \\ [1\,0\,1\,0\,.\,F(n-4)] \\ [0\,0\,.\,F(n-2)\;\;\;] \\ [0\,1\,0\,.\,F(n-3)\;\;]\end{matrix} \tag{12.1-2}$$

12.2 Generalized Fibonacci words

```
............................................1111111111111111111111111111111111111
........................11111111111111111111........................1111111111111
.............11111111111.............1111111.............11111111111.............
.......111111.......1111.......111111..............111111.......1111.......111111
....111....11....111........111....11....111....111....11....111........111....11
..11..1..11....11..1..11..11..1..11....11..1..11..1..11....11..1..11..11..1..11..
.1.1.1..1.1.1.1.1.1..1.1.1.1.1..1.1.1.1.1.1..1.1.1..1.1.1.1.1.1..1.1.1.1.1..1.1.1

1111111111111111111111111111111111111............................................
1111111111111................................................11111111111111111111
..........................1111111111111111111111..........................1111111
.......111111111111..............11111111..............111111111111..............
111........1111........111111................111111........1111........111111....
1....1111........1111....11....1111....1111....11....1111........1111....11....11
..11..11..11..11..11..11....11..11..11..11..11....11..11..11..11..11..11....11..1
```

Figure 12.2-A: The 7-bit binary words with maximal 2 successive ones in lexicographic (top) and minimal-change (bottom) order. Dots denote zeros.

```
1111111111111 111111111111111111111111 ............................................
1111111111111 ........................
.............                          ........................11111111111111111111
              .............11111111111 11111111111..........................1111111
.......111111 111111..............1111 1111..............111111111111..............
111........11 11........111111........ ........111111........1111........111111....
1....1111.... ....1111....11....1111.. ..1111....11....1111........1111....11....11
..11..11..11. .11..11..11....11..11..1 1..11..11....11..11..11..11..11..11....11..1
```

Figure 12.2-B: Recursive structure for the 7-bit binary words with maximal 2 successive ones.

We generalize the Fibonacci words by allowing a fixed maximum value r of successive ones in a binary word. The Fibonacci words correspond to $r = 1$. Figure 12.2-A shows the 7-bit words with $r = 2$. The method to generate a Gray code for these words is a straightforward generalization of the recursion for the Fibonacci words. Write $L_r(n)$ for the list of n-bit words with at most r successive ones, then the recursive structure for the Gray code is

$$L_r(n) \quad = \quad \begin{array}{ll} [0\,.\,L_r^{\mathbf{R}}(n-1) &] \\ [1\,0\,.\,L_r^{\mathbf{R}}(n-2) &] \\ [1\,1\,0\,.\,L_r^{\mathbf{R}}(n-3) &] \\ [\quad\vdots &] \\ [1^{r-2}\,0\,.\,L_r^{\mathbf{R}}(n-1-r+2) &] \\ [1^{r-1}\,0\,.\,L_r^{\mathbf{R}}(n-1-r+1) &] \\ [1^{r}\,0\,.\,L_r^{\mathbf{R}}(n-1-r) &] \end{array} \qquad (12.2\text{-}1)$$

Figure 12.2-B shows the structure for $L_2(7)$, corresponding to the three lowest sublists on the right side of the equation. An implementation is [FXT: comb/maxrep-gray-demo.cc]:

```
ulong n;    // number of bits in words
ulong *rv;  // bits of the word
long mr;    // maximum number of successive ones

void maxrep_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        z = !z;

        long km = mr;
        if ( d+km > n )  km = n - d;

        if ( z )
        {
            // words: 0, 10, 110, 1110, ...
            for (long k=0; k<=km; ++k)
```

```
            {
                rv[d+k] = 0;
                maxrep_rec(d+1+k, z);
                rv[d+k] = 1;
            }
        }
        else
        {
            // words: ... 1110, 110, 10, 0
            for (long k=0; k<km; ++k)  rv[d+k] = 1;
            for (long k=km; k>=0; --k)
            {
                rv[d+k] = 0;
                maxrep_rec(d+1+k, z);
            }
        }
    }
}
```

Figure 12.2-C shows the 5-bit Gray codes for $r \in \{1, 2, 3, 4, 5\}$. Observe that all sequences are subsequences of the leftmost column.

```
          r = 5       r = 4       r = 3       r = 2       r = 1
 1:   1 1 1 1 1   1 1 1 1 .   1 1 1 . .   1 1 . . 1   1 . . 1 .
 2:   1 1 1 1 .   1 1 1 . .   1 1 1 . 1   1 1 . . .   1 . . . .
 3:   1 1 1 . .   1 1 1 . 1   1 1 . . 1   1 1 . 1 .   1 . . . 1
 4:   1 1 1 . 1   1 1 . . 1   1 1 . . .   1 1 . 1 1   1 . 1 . 1
 5:   1 1 . . 1   1 1 . . .   1 1 . 1 .   1 . . 1 1   1 . 1 . .
 6:   1 1 . . .   1 1 . 1 .   1 1 . 1 1   1 . . 1 .   . . 1 . .
 7:   1 1 . 1 .   1 1 . 1 1   1 . . 1 1   1 . . . .   . . 1 . 1
 8:   1 1 . 1 1   1 . . 1 1   1 . . 1 .   1 . . . 1   . . . . 1
 9:   1 . . 1 1   1 . . 1 .   1 . . . .   1 . 1 . 1   . . . . .
10:   1 . . 1 .   1 . . . .   1 . . . 1   1 . 1 . .   . . . 1 .
11:   1 . . . .   1 . . . 1   1 . 1 . 1   1 . 1 1 .   . 1 . 1 .
12:   1 . . . 1   1 . 1 . 1   1 . 1 . .   . . 1 1 .   . 1 . . .
13:   1 . 1 . 1   1 . 1 . .   1 . 1 1 .   . . 1 . .   . 1 . . 1
14:   1 . 1 . .   1 . 1 1 .   1 . 1 1 1   . . 1 . 1
15:   1 . 1 1 .   1 . 1 1 1   . . 1 1 1   . . . . 1
16:   1 . 1 1 1   . . 1 1 1   . . 1 1 .   . . . . .
17:   . . 1 1 1   . . 1 1 .   . . 1 . .   . . . 1 .
18:   . . 1 1 .   . . 1 . .   . . 1 . 1   . . . 1 1
19:   . . 1 . .   . . 1 . 1   . . . . 1   . 1 . 1 1
20:   . . 1 . 1   . . . . 1   . . . . .   . 1 . 1 .
21:   . . . . 1   . . . . .   . . . 1 .   . 1 . . .
22:   . . . . .   . . . 1 .   . . . 1 1   . 1 . . 1
23:   . . . 1 .   . . . 1 1   . 1 . 1 1   . 1 1 . 1
24:   . . . 1 1   . 1 . 1 1   . 1 . 1 .   . 1 1 . .
25:   . 1 . 1 1   . 1 . 1 .   . 1 . . .
26:   . 1 . 1 .   . 1 . . .   . 1 . . 1
27:   . 1 . . .   . 1 . . 1   . 1 1 . 1
28:   . 1 . . 1   . 1 1 . 1   . 1 1 . .
29:   . 1 1 . 1   . 1 1 . .   . 1 1 1 .
30:   . 1 1 . .   . 1 1 1 .
31:   . 1 1 1 .   . 1 1 1 1
32:   . 1 1 1 1
```

Figure 12.2-C: Gray codes of the 5-bit binary words with maximal r successive ones. The leftmost column is the complement of the Gray code of all binary words, the rightmost column is the Gray code for the Fibonacci words.

Let $w_r(n)$ be the number of n-bit words $W_r(n)$ with $\leq r$ successive ones. Taking the length of the lists on both sides of relation 12.2-1 we obtain the recursion

$$w_r(n) \quad = \quad \sum_{j=0}^{r} w_r(n-1-j) \tag{12.2-2}$$

where we set $w_r(n) = 2^k$ for $0 \leq n \leq r$. The sequences for $r \leq 5$ start as

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r=1:	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597
r=2:	1	2	4	7	13	24	44	81	149	274	504	927	1705	3136	5768	10609
r=3:	1	2	4	8	15	29	56	108	208	401	773	1490	2872	5536	10671	20569
r=4:	1	2	4	8	16	31	61	120	236	464	912	1793	3525	6930	13624	26784
r=5:	1	2	4	8	16	32	63	125	248	492	976	1936	3840	7617	15109	29970

For $r = 1$ we obtain the *Fibonacci numbers*, entry A000045 of [214]; For $r = 2$ the *tribonacci numbers*, entry A000073; for $r = 3$ the *tetranacci numbers*, entry A000078; for $r = 4$ the *pentanacci numbers*, entry A001591; for $r = 5$ the *hexanacci numbers*, entry A001592. The variant of the Fibonacci sequence where each number is the sum of its k predecessors is also called *Fibonacci k-step sequence*. The generating function for $w_r(n)$ is

$$\sum_{n=0}^{\infty} w_r(n)\, x^n \quad = \quad \frac{\sum_{k=0}^{r} x^k}{1 - \sum_{k=1}^{r+1} x^k} \tag{12.2-3}$$

Alternative Gray code for words without substrings 111 ($r = 2$)

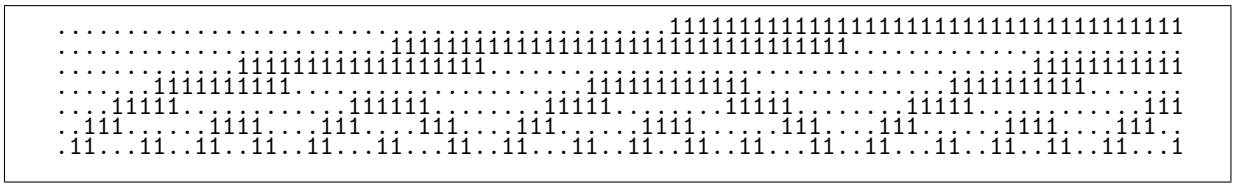


Figure 12.2-D: The 7-bit binary words with maximal 2 successive ones in a minimal-change order.

The list recursion for the Gray code for binary words without substrings 111 is the special case $r = 2$ of relation 12.2-1 on page 284:

$$L_2(n) \quad = \quad \begin{array}{l} [1\,1\,0\,.\,L_2^{\mathbf{R}}(n-3)] \\ [1\,0\,.\,L_2^{\mathbf{R}}(n-2)\ \ ] \\ [0\,.\,L_2^{\mathbf{R}}(n-1)\ \ \ ] \end{array} \tag{12.2-4}$$

A different Gray code is obtained via

$$L'_2(n) \quad = \quad \begin{array}{l} [1\,0\,.\,L'_2(n-2)\ \ \ ] \\ [1\,1\,0\,.\,L'^{\mathbf{R}}_2(n-3)] \\ [0\,.\,L'_2(n-1)\ \ \ \ ] \end{array} \tag{12.2-5}$$

The ordering is shown in figure 12.2-D, it was created with the program [FXT: comb/no111-gray-demo.cc].

Alternative Gray code for words without substrings 1111 ($r = 3$)

A list recursion for an alternative Gray code for binary words without substrings 1111 ($r = 3$) is

$$L'_3(n) \quad = \quad \begin{array}{l} [1\,1\,0\,.\,L'^{\mathbf{R}}_3(n-3)\ \ ] \\ [0\,.\,L'^{\mathbf{R}}_3(n-1)\ \ \ \ ] \\ [1\,1\,1\,0\,.\,L'^{\mathbf{R}}_3(n-4)] \\ [1\,0\,.\,L'^{\mathbf{R}}_3(n-2)\ \ \ ] \end{array} \tag{12.2-6}$$

The ordering is shown in figure 12.2-E, it was created with the program [FXT: comb/no1111-gray-demo.cc]. For all odd $r \geq 3$ a Gray code can be obtained by a list recursion where the prefixes with an even number of ones are followed by those with an odd number of ones. For example, with $r = 5$ the recursion is

$$L'_5(n) \quad = \quad \begin{array}{l} [1\,1\,1\,1\,0\,.\,L'^{\mathbf{R}}_5(n-7)\ \ ] \\ [1\,1\,0\,.\,L'^{\mathbf{R}}_5(n-3)\ \ \ \ ] \\ [0\,.\,L'^{\mathbf{R}}_5(n-1)\ \ \ \ ] \\ [1\,1\,1\,1\,1\,0\,.\,L'^{\mathbf{R}}_5(n-6)] \\ [1\,1\,1\,0\,.\,L'^{\mathbf{R}}_5(n-4)\ \ ] \\ [1\,0\,.\,L'^{\mathbf{R}}_5(n-2)\ \ \ \ ] \end{array} \tag{12.2-7}$$

```
  1:  1.11.1.      28:  1.1..1.      55:  ..111.1      82:  .1.1.1.
  2:  1.11...      29:  1.1.11.      56:  ...11.1      83:  .1.111.
  3:  1.11..1      30:  111.11.      57:  ...11..      84:  .1..11.
  4:  1.11.11      31:  111..1.      58:  ....1..      85:  .1...1.
  5:  1..1.11      32:  111....      59:  ....1.1      86:  .1.....
  6:  1..1..1      33:  111...1      60:  ....111      87:  .1....1
  7:  1..1...      34:  111..11      61:  .....11      88:  .1...11
  8:  1..1.1.      35:  111.111      62:  ......1      89:  .1..111
  9:  1..111.      36:  111.1.1      63:  .......      90:  .1..1.1
 10:  1...11.      37:  111.1..      64:  .....1.      91:  .1..1..
 11:  1....1.      38:  .11.1..      65:  ....11.      92:  .1.11..
 12:  1......      39:  .11.1.1      66:  ...111.      93:  .1.11.1
 13:  1.....1      40:  .11.111      67:  ...1.1.      94:  11.11.1
 14:  1....11      41:  .11..11      68:  ...1...      95:  11.11..
 15:  1...111      42:  .11...1      69:  ...1..1      96:  11..1..
 16:  1...1.1      43:  .11....      70:  ...1.11      97:  11..1.1
 17:  1...1..      44:  .11..1.      71:  ..11.11      98:  11..111
 18:  1..11..      45:  .11.11.      72:  ..11..1      99:  11...11
 19:  1..11.1      46:  ..1.11.      73:  ..11...     100:  11....1
 20:  1.111.1      47:  ..1..1.      74:  ..11.1.     101:  11.....
 21:  1.111..      48:  ..1....      75:  .111.1.     102:  11...1.
 22:  1.1.1..      49:  ..1...1      76:  .111...     103:  11..11.
 23:  1.1.1.1      50:  ..1..11      77:  .111..1     104:  11.111.
 24:  1.1.111      51:  ..1.111      78:  .111.11     105:  11.1.1.
 25:  1.1..11      52:  ..1.1.1      79:  .1.1.11     106:  11.1...
 26:  1.1...1      53:  ..1.1..      80:  .1.1..1     107:  11.1..1
 27:  1.1....      54:  ..111..      81:  .1.1...     108:  11.1.11
```

Figure 12.2-E: The 7-bit binary words with maximal 3 successive ones in a minimal-change order.

12.3 Digit x followed by at least x zeros

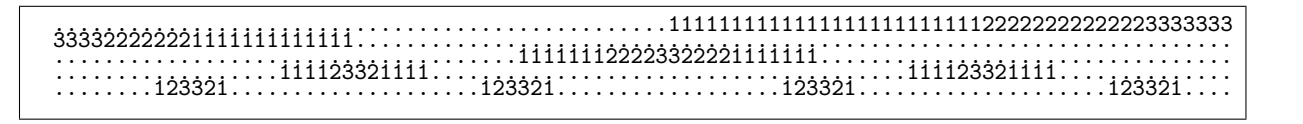


Figure 12.3-A: Gray code for the length-5 words with maximal digit 3 where a digit x is followed by at least x zeros. Dots denote zeros.

Figure 12.3-A shows a Gray code for the length-5 words with maximal digit 3 where a digit x is followed by at least x zeros. For the Gray code list $Z_r(n)$ of the length-n words with maximal digit r we have

$$Z_r(n) \;=\; \begin{bmatrix} [0\,.\,Z_r^{\mathbf{R}}(n-1) &] \\ [1\,0\,.\,Z_r^{\mathbf{R}}(n-2) &] \\ [2\,0\,0\,.\,Z_r^{\mathbf{R}}(n-3) &] \\ [3\,0\,0\,0\,.\,Z_r^{\mathbf{R}}(n-4) &] \\ [\quad\vdots &] \\ [r\,0^r\,.\,Z_r^{\mathbf{R}}(n-r-1) &] \end{bmatrix} \tag{12.3-1}$$

A implementation is [FXT: comb/gexz-gray-demo.cc]:

```
ulong n;    // number of digits in words
ulong *rv;  // digits of the word
ulong mr;   // radix== mr+1

void gexz_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( z )
        {
            // words 0, 10, 200, 3000, 40000, ...
            ulong k = 0;
            do
            {
                rv[d]=k;
                for (ulong j=1; j<=k; ++j)   rv[d+j] = 0;
                gexz_rec(d+k+1, !z);
```

```
            }
            while ( ++k <= mr );
        }
        else
        {
            // words ..., 40000, 3000, 200, 10, 0
            ulong k = mr + 1;
            do
            {
                --k;
                rv[d]=k;
                for (ulong j=1; j<=k; ++j)   rv[d+j] = 0;
                gexz_rec(d+k+1, !z);
            }
            while ( k != 0 );
        }
    }
}
```

Let $z_r(n)$ be the number of n-bit words $Z_r(n)$, then

$$z_r(n) \quad = \quad \sum_{j=1}^{r+1} z_r(n-j) \tag{12.3-2}$$

where we set $z_r(n) = 1$ for $n \leq 0$. The sequences for $r \leq 5$ start as

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r=1:	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597
r=2:	1	3	5	9	17	31	57	105	193	355	653	1201	2209	4063	7473	13745
r=3:	1	4	7	13	25	49	94	181	349	673	1297	2500	4819	9289	17905	34513
r=4:	1	5	9	17	33	65	129	253	497	977	1921	3777	7425	14597	28697	56417
r=5:	1	6	11	21	41	81	161	321	636	1261	2501	4961	9841	19521	38721	76806

For $r = 1$ we obtain the *Fibonacci numbers*, entry A000045 of [214]; For $r = 2$ the *tribonacci numbers*, entry A000213; for $r = 3$ the *tetranacci numbers*, entry A000288; for $r = 4$ the *pentanacci numbers*, entry A000322; for $r = 5$ the *hexanacci numbers*, entry A000383. Note that the sequences for $r \geq 2$ are different from those obtained via relation 12.2-2 on page 285.

12.4 Generalized Pell words

Gray code for Pell words

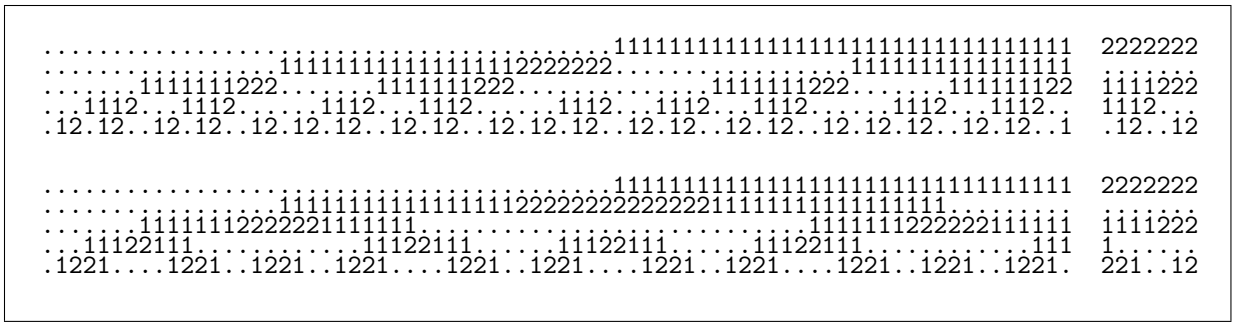


Figure 12.4-A: Start and end of the lists of five digit Pell words in counting order (top) and Gray code order (bottom). The lowest row is the least significant digit, dots denote zeros.

A Gray code of the Pell words (ternary words without the substrings `"21"` and `"22"`) can be computed as follows:

```
ulong n;    // number of digits in words
ulong *rv;  // digits of the word
bool zq;    // order: 0==>Lex,  1==>Gray
```

```
void pell_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  pell_rec(d+1, z);
            rv[d]=1;  pell_rec(d+1, zq^z);
            rv[d]=2;  rv[d+1]=0;  pell_rec(d+2, z);
        }
        else
        {
            rv[d]=2;  rv[d+1]=0;  pell_rec(d+2, z);
            rv[d]=1;  pell_rec(d+1, zq^z);
            rv[d]=0;  pell_rec(d+1, z);
        }
    }
}
```

The global boolean variable `zq` controls whether the counting order or the Gray code is generated. The code is given in [FXT: comb/pellgray-rec-demo.cc]. Both orderings are shown in figure 12.4-A. About 110 million words per second are generated. The computation of a function whose power series coefficients are related to the Pell Gray code is described in section 36.12.3 on page 724.

Gray code for generalized Pell words

```
.....................................................11111111111111111111111111111111
33332222222222221111111111111.....................................1111111111111122222
........11112222332222111111.........1111222233222211111.........11112222332222
.123321..123321....123321..123321..123321....123321..123321..123321....123

  111111111112222222222222222222222222222222222222222223333333333333
  22222222333333332222222222222111111111111.....................
  1111.................11112222333222211111.........11112222333222211111....
  321..123321..123321..123321....123321..123321..123321....123321..123
```

Figure 12.4-B: Gray code for 4-digit radix-4 strings with no substring $3x$ with $x \neq 0$.

A generalization of the Pell words are the radix-$(r+1)$ strings where the substring rx with $x \neq 0$ is forbidden. Let $P_r(n)$ the length-n words, a gray code for these strings can be generated by the recursion

$$P_r(n) = \begin{bmatrix} [0\,.\,P_r(n-1) &] \\ [1\,.\,P_r^{\mathbf{R}}(n-1) &] \\ [2\,.\,P_r(n-1) &] \\ [3\,.\,P_r^{\mathbf{R}}(n-1) &] \\ [\quad\vdots &] \\ [(r-1)\,.\,P_r^{\mathbf{R}}(n-1) &] \\ [(r)\,0\,.\,P_r(n-2) &] \end{bmatrix} \tag{12.4-1a}$$

if r is even, and

$$P_r(n) = \begin{bmatrix} [0\,.\,P_r^{\mathbf{R}}(n-1) &] \\ [1\,.\,P_r(n-1) &] \\ [2\,.\,P_r^{\mathbf{R}}(n-1) &] \\ [3\,.\,P_r(n-1) &] \\ [\quad\vdots &] \\ [(r-1)\,.\,P_r(n-1) &] \\ [(r)\,0\,.\,P_r^{\mathbf{R}}(n-2) &] \end{bmatrix} \tag{12.4-1b}$$

if r is odd. Figure 12.4-B shows a Gray code for the 4-digit strings with $r = 3$. An implementation of the algorithm is [FXT: comb/pellgen-gray-demo.cc]:

```
ulong n;    // number of digits in words
ulong *rv;  // digits of the word (radix r+1)
long r;     // Forbidden substrings are [r, x] where x!=0

void pellgen_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        const bool p = r & 1;  // parity of r
        rv[d] = 0;
        if ( z )
        {
            for (long k=0; k<r; ++k)  { rv[d] = k;  pellgen_rec(d+1, z ^ p ^ (k&1)); }
            { rv[d] = r;  rv[d+1] = 0;  pellgen_rec(d+2, p ^ z); }
        }
        else
        {
            { rv[d] = r;  rv[d+1] = 0;  pellgen_rec(d+2, p ^ z); }
            for (long k=r-1; k>=0; --k)  { rv[d] = k;  pellgen_rec(d+1, z ^ p ^ (k&1)); }
        }
    }
}
```

With $r = 1$ we again obtain the Gray code for Fibonacci words. Taking the number $p_r(n)$ of words $P_r(n)$ on both sides of relations 12.4-1a and 12.4-1b we find

$$p_r(n) \quad = \quad r\,p_r(n) + p_r(n-2) \tag{12.4-2}$$

where $p_r(0) = 1$ and $p_r(1) = r + 1$. For $r \le 5$ the sequences start as

n:	0	1	2	3	4	5	6	7	8	9	10	11
r=1:	1	2	3	5	8	13	21	34	55	89	144	233
r=2:	1	3	7	17	41	99	239	577	1393	3363	8119	19601
r=3:	1	4	13	43	142	469	1549	5116	16897	55807	184318	608761
r=4:	1	5	21	89	377	1597	6765	28657	121393	514229	2178309	9227465
r=5:	1	6	31	161	836	4341	22541	117046	607771	3155901	16387276	85092281

The sequences are the following entries of [214]: $r = 1$: A000045; $r = 2$: A001333; $r = 3$: A003688; $r = 4$: A015448; $r = 5$: A015449. The generating function for $p_r(n)$ is

$$\sum_{n=0}^{\infty} p_r(n)\,x^n \quad = \quad \frac{1+x}{1-r\,x-x^2} \tag{12.4-3}$$

12.5 Sparse signed binary words

```
                                          .MMMMMMMMMMMMMMMMMMMMPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPMMMMMMMMMMM..........................................................
                          .MMMMMPPPPPPPPPPPMMMMM.....................MMMMMPPPPP
PPPMMM.........MMMPPPPPPMMM.....................MMMPPPPPPMMM.....................
.......MPPM.....................MPPM......MPPM.....MPPM.....................MPPM......MP
PM..MPPM......MPPM..MPPM..MPPM......MPPM......MPPM......MPPM..MPPM..MPPM......MPPM...
```

Figure 12.5-A: A Gray code through the 85 sparse 6-bit signed binary words. Dots are used for zeros, the symbols 'P' and 'M' denote +1 and −1, respectively.

Figure 12.5-A shows a minimal-change order (Gray code) for the sparse signed binary words (nonadjacent form (NAF), see section 1.24 on page 59). Note that we allow a digit to switch between +1 and −1. If all words with any positive digit ('P') are omitted then we obtain the Gray code for Fibonacci words given in section 12.1 on page 282.

A recursive routine for the generation of the Gray code is given in [FXT: comb/naf-gray-rec-demo.cc]:

```
ulong n;   // number of digits of the string
int *rv;   // the string
```

```
void sb_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  sb_rec(d+1, 1);
            rv[d]=-1;  rv[d+1]=0;  sb_rec(d+2, 1);
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);
        }
        else
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);
            rv[d]=-1;  rv[d+1]=0;  sb_rec(d+2, 0);
            rv[d]=0;  sb_rec(d+1, 0);
        }
    }
}
```

About 120 million words per second are generated.

Let $S(n)$ be the number of n-digit sparse signed binary numbers (of both signs), and $P(n)$ be the number of positive n-digit sparse signed binary numbers, then

```
   n: 0 1 2  3  4  5  6  7   8   9   10   11   12   13    14    15    16
S(n): 1 3 5 11 21 43 85 171 341 683 1365 2731 5461 10923 21845 43691 87381
P(n): 1 2 3  6 11 22 43 86  171 342  683 1366 2731  5462 10923 21846 43691
```

The sequence $S(n)$ is entry A001045 of [214], the sequence $P(n)$ is entry A005578. We have (with $e := n \bmod 2$)

$$\begin{aligned}
S(n) &= \frac{2^{n+2}-1+2\,e}{3} = 2\,S(n-1)-1+2\,e && \text{(12.5-1a)}\\
&= S(n-1)+2\,S(n-2) = 3\,S(n-2)+2\,S(n-3) = 2\,P(n)-1 && \text{(12.5-1b)}\\
P(n) &= \frac{2^{n+1}+1+e}{3} = 2\,P(n-1)-1-e = S(n-1)+e && \text{(12.5-1c)}\\
&= P(n-1)+S(n-2) = P(n-2)+S(n-2)+S(n-3) && \text{(12.5-1d)}\\
&= S(n-2)+S(n-3)+S(n-4)+\ldots+S(2)+S(1)+3 && \text{(12.5-1e)}\\
&= 2\,P(n-1)+P(n-2)-2\,P(n-3) && \text{(12.5-1f)}
\end{aligned}$$

Almost Gray code for positive words *

```
                     ><          ><
.....................................PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPP.....................................................................
.............................PPPPPPPPPPPPPPPPPPPPPPPPPMMMMMMMMMMM.....................
PPPPPMMMMM.........PPPPP.....................................................MMMMMPPPPP
.............MMMPPP.....PPP.....MMMPPPPPPMMM.........MMMPPPPPPMMM.............
PM.....MPPM............MPP.....PM.................MPPM.................MPPM.....MP
...MPPM.....MPPM..MPPM.....PPM....MPPM..MPPM..MPPM.....MPPM..MPPM..MPPM.....MPPM...
                     ><          ><
```

Figure 12.5-B: An ordering of the 86 sparse 7-bit positive signed binary words that is almost a Gray code. The transitions that are not minimal are marked with '><'. Dots denote zeros.

If we start with the following routine that calls `sb_rec()` only after a one has been inserted, we obtain an ordering of the positive numbers:

```
void pos_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  pos_rec(d+1, 1);
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);
```

```
        }
        else
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);
            rv[d]=0;  pos_rec(d+1, 0);
        }
    }
}
```

The ordering obtained with n-digit words is a Gray code, except for $n-4$ transitions. An ordering with only about $n/2$ non-Gray transitions is obtained by the more complicated recursion [FXT: comb/naf-pos-rec-demo.cc]:

```
void pos_AAA(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);  // 0
            rv[d]=0;  pos_AAA(d+1, 1);  // 1
        }
        else
        {
            rv[d]=0;  pos_BBB(d+1, 0);  // 0
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);  // 1
        }
    }
}

void pos_BBB(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 1);  // 1
            rv[d]=0;  pos_BBB(d+1, 1);  // 1
        }
        else
        {
            rv[d]=0;  pos_AAA(d+1, 0);  // 0
            rv[d]=+1;  rv[d+1]=0;  sb_rec(d+2, 0);  // 0
        }
    }
}
```

The initial call is `pos_AAA(0,0)`. The result for $n = 7$ is shown in figure 12.5-B. We list the number N of non-Gray transitions and the number of digit changes X in excess of a Gray code for $n \le 30$:

```
n: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
N: 0 0 0 0 1 2 2 2 3  4  4  4  5  6  6  6  7  8  8  8  9 10 10 10 11 12 12 12 13 14
X: 0 0 0 0 1 3 4 4 5  7  8  8  9 11 12 12 13 15 16 16 17 19 20 20 21 23 24 24 25 27
```

12.6 Strings with no two successive nonzero digits

A Gray code for the length-n strings with radix $(r+1)$ and no two successive nonzero digits is obtained by the recursion for the list $D_r(n)$:

$$D_r(n) \;=\; \begin{matrix} [\;0\,.\,D_r^{\mathbf{R}}(n-1)] \\ [1\,0\,.\,D_r^{\mathbf{R}}(n-1)] \\ [2\,0\,.\,D_r(n-1)\;] \\ [3\,0\,.\,D_r^{\mathbf{R}}(n-1)] \\ [4\,0\,.\,D_r(n-1)\;] \\ [5\,0\,.\,D_r^{\mathbf{R}}(n-1)] \\ [\quad\vdots\qquad\quad] \end{matrix} \tag{12.6-1}$$

An implementation is [FXT: comb/ntnz-gray-demo.cc]:

```
  1:  .3..3      26:  ....1      51:  1.1.2      76:  2.3.2
  2:  .3..2      27:  ....2      52:  1.1.3      77:  2.3.1
  3:  .3..1      28:  ....3      53:  1...3      78:  2.3..
  4:  .3...      29:  ..1.3      54:  1...2      79:  3.3..
  5:  .3.1.      30:  ..1.2      55:  1...1      80:  3.3.1
  6:  .3.2.      31:  ..1.1      56:  1....      81:  3.3.2
  7:  .3.3.      32:  ..1..      57:  1..1.      82:  3.3.3
  8:  .2.3.      33:  ..2..      58:  1..2.      83:  3.2.3
  9:  .2.2.      34:  ..2.1      59:  1..3.      84:  3.2.2
 10:  .2.1.      35:  ..2.2      60:  2..3.      85:  3.2.1
 11:  .2...      36:  ..2.3      61:  2..2.      86:  3.2..
 12:  .2..1      37:  ..3.3      62:  2..1.      87:  3.1..
 13:  .2..2      38:  ..3.2      63:  2....      88:  3.1.1
 14:  .2..3      39:  ..3.1      64:  2...1      89:  3.1.2
 15:  .1..3      40:  ..3..      65:  2...2      90:  3.1.3
 16:  .1..2      41:  1.3..      66:  2...3      91:  3...3
 17:  .1..1      42:  1.3.1      67:  2.1.3      92:  3...2
 18:  .1...      43:  1.3.2      68:  2.1.2      93:  3...1
 19:  .1.1.      44:  1.3.3      69:  2.1.1      94:  3....
 20:  .1.2.      45:  1.2.3      70:  2.1..      95:  3..1.
 21:  .1.3.      46:  1.2.2      71:  2.2..      96:  3..2.
 22:  ...3.      47:  1.2.1      72:  2.2.1      97:  3..3.
 23:  ...2.      48:  1.2..      73:  2.2.2
 24:  ...1.      49:  1.1..      74:  2.2.3
 25:  .....      50:  1.1.1      75:  2.3.3
```

Figure 12.6-A: Gray code for the length-4 radix-4 strings with no two successive nonzero digits.

```
ulong n;     // length of strings
ulong *rv;   // digits of strings
ulong mr;    // max digit

void ntnz_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        if ( 0==z )
        {
            rv[d]=0;  ntnz_rec(d+1, 1);
            for (ulong t=1; t<=mr; ++t)  { rv[d]=t;  rv[d+1]=0;  ntnz_rec(d+2, t&1); }
        }
        else
        {
            for (ulong t=mr; t>0; --t)  { rv[d]=t;  rv[d+1]=0;  ntnz_rec(d+2, !(t&1)); }
            rv[d]=0;  ntnz_rec(d+1, 0);
        }
    }
}
```

Figure 12.6-A shows the Gray code for length-4, radix-4 ($r = 3$) strings. With $r = 2$ and upon replacing 1 with -1 and 2 with $+1$ we obtain the Gray code for the sparse binary words (figure 12.5-A on page 290). With $r = 1$ we again obtain the Gray code for the Fibonacci words.

Counting the elements on both sides of relation 12.6-1 on the preceding page we find that for the number $d_r(n)$ of strings in the list $D_r(n)$ we have

$$d_r(n) \quad = \quad d_r(n-1) + r\, d_r(n-2) \tag{12.6-2}$$

where $d_r(0) = 1$ and $d_r(1) = r + 1$. The sequences of these numbers start as

```
    n: 0 1  2  3  4   5   6    7    8     9     10     11     12      13      14
 r=1: 1 2  3  5  8  13  21   34   55    89    144    233    377     610     987
 r=2: 1 3  5 11 21  43  85  171  341   683  1365   2731   5461   10923   21845
 r=3: 1 4  7 19 40  97 217  508 1159  2683  6160  14209  32689   75316  173383
 r=4: 1 5  9 29 65 181 441 1165 2929  7589 19305  49661 126881  325525  833049
 r=5: 1 6 11 41 96 301 781 2286 6191 17621 48576 136681 379561 1062966 2960771
```

These are the following entries of [214]: $r = 1$: A000045; $r = 2$: A001045; $r = 3$: A006130; $r = 4$: A006131; $r = 5$: A015440; $r = 6$: A015441; $r = 7$: A015442; $r = 8$: A015443. The generating function for $d_r(n)$ is

$$\sum_{n=0}^{\infty} d_r(n)\, x^n \quad = \quad \frac{1 + r\, x}{1 - x - r\, x^2} \tag{12.6-3}$$

12.7 Strings with no two successive zeros

```
  1: . 1 3    21: 1 1 .    41: 2 3 1
  2: . 1 2    22: 1 1 1    42: 2 3 .
  3: . 1 1    23: 1 1 2    43: 3 3 .
  4: . 1 .    24: 1 1 3    44: 3 3 1
  5: . 2 .    25: 1 . 3    45: 3 3 2
  6: . 2 1    26: 1 . 2    46: 3 3 3
  7: . 2 2    27: 1 . 1    47: 3 2 3
  8: . 2 3    28: 2 . 1    48: 3 2 2
  9: . 3 3    29: 2 . 2    49: 3 2 1
 10: . 3 2    30: 2 . 3    50: 3 2 .
 11: . 3 1    31: 2 1 3    51: 3 1 .
 12: . 3 .    32: 2 1 2    52: 3 1 1
 13: 1 3 .    33: 2 1 1    53: 3 1 2
 14: 1 3 1    34: 2 1 .    54: 3 1 3
 15: 1 3 2    35: 2 2 .    55: 3 . 3
 16: 1 3 3    36: 2 2 1    56: 3 . 2
 17: 1 2 3    37: 2 2 2    57: 3 . 1
 18: 1 2 2    38: 2 2 3
 19: 1 2 1    39: 2 3 3
 20: 1 2 .    40: 2 3 2
```

```
  1: . 1 2 2    21: 1 2 1 1    41: 2 . 1 .
  2: . 1 2 1    22: 1 2 1 2    42: 2 . 2 .
  3: . 1 2 .    23: 1 2 . 2    43: 2 . 2 1
  4: . 1 1 .    24: 1 2 . 1    44: 2 . 2 2
  5: . 1 1 1    25: 1 1 . 1    45: 2 1 2 2
  6: . 1 1 2    26: 1 1 . 2    46: 2 1 2 1
  7: . 1 . 2    27: 1 1 1 2    47: 2 1 2 .
  8: . 1 . 1    28: 1 1 1 1    48: 2 1 1 .
  9: . 2 . 1    29: 1 1 1 .    49: 2 1 1 1
 10: . 2 . 2    30: 1 1 2 .    50: 2 1 1 2
 11: . 2 1 2    31: 1 1 2 1    51: 2 1 . 2
 12: . 2 1 1    32: 1 1 2 2    52: 2 1 . 1
 13: . 2 1 .    33: 1 . 2 2    53: 2 2 . 1
 14: . 2 2 .    34: 1 . 2 1    54: 2 2 . 2
 15: . 2 2 1    35: 1 . 2 .    55: 2 2 1 2
 16: . 2 2 2    36: 1 . 1 .    56: 2 2 1 1
 17: 1 2 2 2    37: 1 . 1 1    57: 2 2 1 .
 18: 1 2 2 1    38: 1 . 1 2    58: 2 2 2 .
 19: 1 2 2 .    39: 2 . 1 2    59: 2 2 2 1
 20: 1 2 1 .    40: 2 . 1 1    60: 2 2 2 2
```

Figure 12.7-A: Gray codes for strings with no two successive zeros: length-3 radix-4 (left), and length-4 radix-3 (right). Dots denote zeros.

A Gray code for the length-n strings with radix $(r+1)$ and no two successive zeros (see figure 12.7-A) is obtained by the recursion for the list $Z_r(n)$ as follows:

$$Z_r(n) = \begin{array}{l} [0\,1\,.\,Z_r(n-2)\,] \\ [0\,2\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [0\,3\,.\,Z_r(n-2)\,] \\ [0\,4\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [0\,5\,.\,Z_r(n-2)\,] \\ [\quad\vdots\qquad\quad] \\ [0\,r\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [1\,.\,Z_r^{\mathbf{R}}(n-1)\;] \\ [2\,.\,Z_r(n-1)\;\;] \\ [3\,.\,Z_r^{\mathbf{R}}(n-1)\;] \\ [4\,.\,Z_r(n-1)\;\;] \\ [\quad\vdots\qquad\quad] \\ [r\,.\,Z_r^{\mathbf{R}}(n-1)\;] \end{array} \quad \text{for } r \text{ even}, \qquad Z_r(n) = \begin{array}{l} [0\,1\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [0\,2\,.\,Z_r(n-2)\,] \\ [0\,3\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [0\,4\,.\,Z_r(n-2)\,] \\ [0\,5\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [\quad\vdots\qquad\quad] \\ [0\,r\,.\,Z_r^{\mathbf{R}}(n-2)] \\ [1\,.\,Z_r(n-1)\;\;] \\ [2\,.\,Z_r^{\mathbf{R}}(n-1)\;] \\ [3\,.\,Z_r(n-1)\;\;] \\ [4\,.\,Z_r^{\mathbf{R}}(n-1)\;] \\ [\quad\vdots\qquad\quad] \\ [r\,.\,Z_r(n-1)\;\;] \end{array} \quad \text{for } r \text{ odd}. \qquad (12.7\text{-}1)$$

An implementation is given in [FXT: comb/ntz-gray-demo.cc]:

```
ulong n;    // number of digits in words
ulong *rv;  // digits of the word (radix r+1)
long r;     // Forbidden substrings are [r, x] where x!=0

void ntz_rec(ulong d, bool z)
{
    if ( d>=n )  visit();
    else
    {
        bool w = 0;  // r-parity:  w depends on z ...
        if ( r&1 )  w = !z;   // ... if r odd

        if ( z )
        {
            // words 0X:
            rv[d] = 0;
            if ( d+2<=n )
            {
                for (long k=1; k<=r; ++k, w=!w)  { rv[d+1]=k;  ntz_rec(d+2, w); }
            }
            else
            {
                ntz_rec(d+1, w);
```

```
            w = !w;
        }

        w ^= (r&1);  //  r-parity:  change direction if r odd

        // words X:
        for (long k=1; k<=r; ++k, w=!w)  { rv[d]=k;  ntz_rec(d+1, w); }
    }
    else
    {
        // words X:
        for (long k=r; k>=1; --k, w=!w)  { rv[d]=k;  ntz_rec(d+1, w); }

        w ^= (r&1);  //  r-parity:  change direction if r odd

        // words 0X:
        rv[d] = 0;
        if ( d+2<=n )
        {
            for (long k=r; k>=1; --k, w=!w)  { rv[d+1]=k;  ntz_rec(d+2, w); }
        }
        else
        {
            ntz_rec(d+1, w);
            w = !w;
        }
    }
  }
}
```

With $r = 1$ we obtain the complement of the minimal-change list of Fibonacci words. Let $z_r(n)$ be the number of words $W_r(n)$, we find

$$z_r(n) \quad = \quad r\, z_r(n-1) + r\, z_r(n-1) \tag{12.7-2}$$

where $z_r(0) = 1$ and $z_r(1) = r + 1$. The sequences for $r \le 5$ start

n:	0	1	2	3	4	5	6	7	8	9	10	11
r=1:	1	2	3	5	8	13	21	34	55	89	144	233
r=2:	1	3	8	22	60	164	448	1224	3344	9136	24960	68192
r=3:	1	4	15	57	216	819	3105	11772	44631	169209	641520	2432187
r=4:	1	5	24	116	560	2704	13056	63040	304384	1469696	7096320	34264064
r=5:	1	6	35	205	1200	7025	41125	240750	1409375	8250625	48300000	282753125

These (for $r \le 4$) are the following entries of [214]: $r = 1$: A000045; $r = 2$: A028859; $r = 3$: A125145; $r = 4$: A086347. The generating function for $z_r(n)$ is

$$\sum_{n=0}^{\infty} z_r(n)\, x^n \quad = \quad \frac{1+x}{1 - r\,x - r\,x^2} \tag{12.7-3}$$

12.8 Binary strings without substrings 1x1

A Gray code for binary strings with no substring 1x1 is shown in figure 12.8-A. The recursive structure for the list $V(n)$ of the n-bit words is

$$V(n) \quad = \quad \begin{matrix} [1\,0\,0\,.\,V(n-3) \quad\quad\;\;] \\ [1\,1\,0\,0\,.\,V^{\mathbf{R}}(n-4)] \\ [0\,.\,V(n-1) \quad\quad\;\;] \end{matrix} \tag{12.8-1}$$

The implied algorithm can be implemented as [FXT: comb/no1x1-gray-demo.cc]:

```
ulong n;    // number of bits in words
ulong *rv;  // bits of the word

void no1x1_rec(ulong d, bool z)
{
    if ( d>=n )  { if ( d<=n+2 )  visit(); }
    else
    {
        if ( z )
```

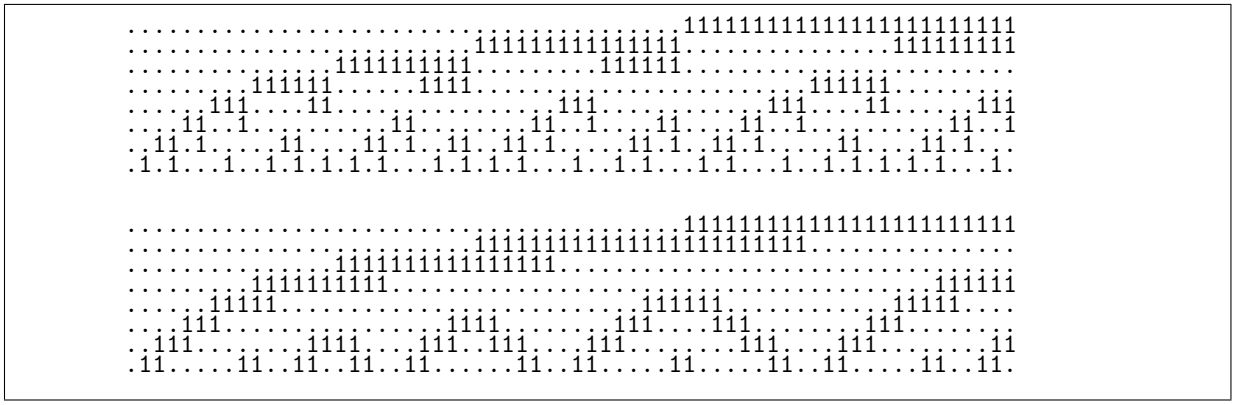


Figure 12.8-A: The length-8 binary strings with no substring 1x1 (where x is either 0 or 1): lex order (top) and minimal-change order (bottom). Dots denote zeros.

```
        {
            rv[d]=1;  rv[d+1]=0;  rv[d+2]=0;  no1x1_rec(d+3, z);
            rv[d]=1;  rv[d+1]=1;  rv[d+2]=0;  rv[d+3]=0;  no1x1_rec(d+4, !z);
            rv[d]=0;  no1x1_rec(d+1, z);
        }
        else
        {
            rv[d]=0;  no1x1_rec(d+1, z);
            rv[d]=1;  rv[d+1]=1;  rv[d+2]=0;  rv[d+3]=0;  no1x1_rec(d+4, !z);
            rv[d]=1;  rv[d+1]=0;  rv[d+2]=0;  no1x1_rec(d+3, z);
        }
    }
}
```

The sequence of the numbers $v(n)$ of length-n strings starts as

```
n:     0 1 2 3 4  5  6  7  8   9  10  11  12  13   14   15   16   17
v(n):  1 2 4 6 9 15 25 40 64 104 169 273 441 714 1156 1870 3025 4895
```

This is entry A006498 of [214]. The recurrence relation is

$$v(n) \quad = \quad v(n-1) + v(n-3) + v(n-4) \tag{12.8-2}$$

The generating function is

$$\sum_{n=0}^{\infty} v(n)\, x^n \quad = \quad \frac{1 + x + 2\,x^2 + x^3}{1 - x - x^3 - x^4} \tag{12.8-3}$$

12.9 Binary strings without substrings 1xy1

Figure 12.9-A shows a Gray code for binary words with no substring 1xy1. The recursion for the list of n-bit words $Y(n)$ is

$$Y(n) \quad = \quad \begin{array}{l} [1\,0\,0\,0\,.\,Y(n-4) \quad\quad\;\;] \\ [1\,0\,1\,0\,0\,0\,.\,Y^{\mathbf{R}}(n-6)] \\ [1\,1\,1\,0\,0\,0\,.\,Y(n-6) \;\;] \\ [1\,1\,0\,0\,0\,.\,Y^{\mathbf{R}}(n-5) \;\;] \\ [0\,.\,Y(n-1) \quad\quad\;\;] \end{array} \tag{12.9-1}$$

An implementation is given in [FXT: comb/no1xy1-gray-demo.cc]:

```
void Y_rec(long p1, long p2, bool z)
{
    if ( p1>p2 )  { visit();  return; }
```

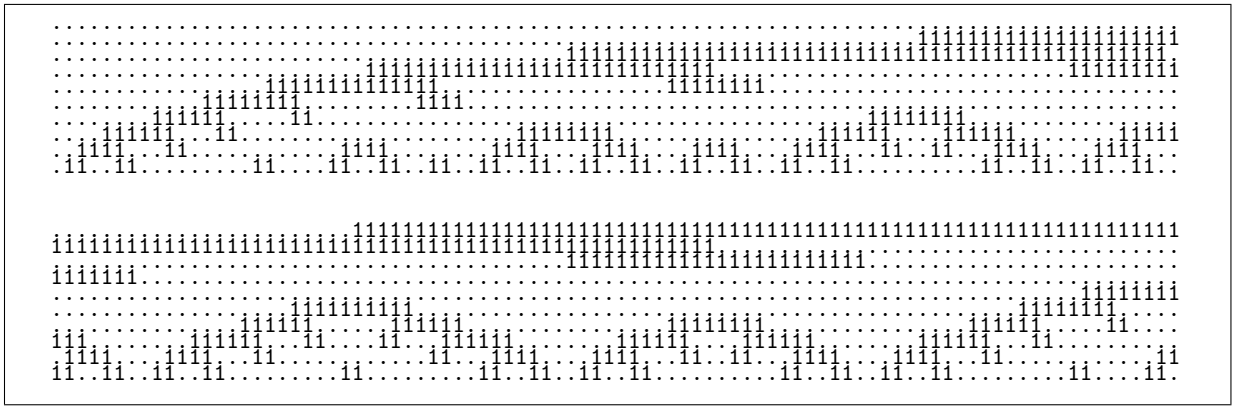


Figure 12.9-A: The length-10 binary strings with no substring 1xy1 (where x and y are either 0 or 1) in minimal-change order. Dots denote zeros.

```
#define S1(a)  rv[p1+0]=a
#define S2(a,b)  S1(a); rv[p1+1]=b;
#define S3(a,b,c)  S2(a,b); rv[p1+2]=c;
#define S4(a,b,c,d)  S3(a,b,c); rv[p1+3]=d;
#define S5(a,b,c,d,e)  S4(a,b,c,d); rv[p1+4]=e;
#define S6(a,b,c,d,e,f)  S5(a,b,c,d,e); rv[p1+5]=f;
    long d = p2 - p1;
    if ( z )
    {
        if ( d >= 0 )  { S4(1,0,0,0);      Y_rec(p1+4, p2,  z); }   // 1 0 0 0
        if ( d >= 2 )  { S6(1,0,1,0,0,0);  Y_rec(p1+6, p2, !z); }   // 1 0 1 0 0 0
        if ( d >= 2 )  { S6(1,1,1,0,0,0);  Y_rec(p1+6, p2,  z); }   // 1 1 1 0 0 0
        if ( d >= 1 )  { S5(1,1,0,0,0);    Y_rec(p1+5, p2, !z); }   // 1 1 0 0 0
        if ( d >= 0 )  { S1(0);            Y_rec(p1+1, p2,  z); }   // 0
    }
    else
    {
        if ( d >= 0 )  { S1(0);            Y_rec(p1+1, p2,  z); }   // 0
        if ( d >= 1 )  { S5(1,1,0,0,0);    Y_rec(p1+5, p2, !z); }   // 1 1 0 0 0
        if ( d >= 2 )  { S6(1,1,1,0,0,0);  Y_rec(p1+6, p2,  z); }   // 1 1 1 0 0 0
        if ( d >= 2 )  { S6(1,0,1,0,0,0);  Y_rec(p1+6, p2, !z); }   // 1 0 1 0 0 0
        if ( d >= 0 )  { S4(1,0,0,0);      Y_rec(p1+4, p2,  z); }   // 1 0 0 0
    }
}
```

Note the conditions `if ( d>= ? )` that make sure that no string appears repeated. The initial call is `Y_rec(0, n-1, 0)`. The sequence of the numbers $y(n)$ of length-n strings starts as

```
n:     0 1 2 3  4  5  6  7  8   9  10  11  12  13   14   15   16   17
y(n):  1 2 4 8 12 17 25 41 69 114 180 280 440 705 1137 1825 2905 4610
```

The generating function is

$$\sum_{n=0}^{\infty} y(n)\,x^n \quad = \quad \frac{1 + x + 2\,x^2 + 4\,x^3 + 3\,x^4 + 2\,x^5}{1 - x - x^4 - x^5 - 2\,x^6} \tag{12.9-2}$$

No substrings 1x1 or 1xy1

A recursion for a Gray code of the of n-bit binary words $Z(n)$ with no substrings 1x1 or 1xy1 (shown in figure 12.9-B) is

$$Z(n) \quad = \quad \begin{matrix} [1\,0\,0\,0\,.\,Z(n-4) \quad\;\;] \\ [1\,1\,0\,0\,0\,.\,Z^{\mathbf{R}}(n-5)] \\ [0\,.\,Z(n-1) \quad\;\;] \end{matrix} \tag{12.9-3}$$

The sequence of the numbers $z(n)$ of length-n strings starts as

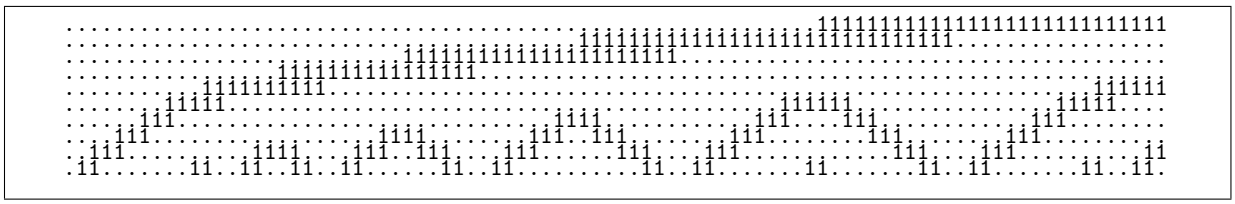


Figure 12.9-B: A Gray code for the length-10 binary strings with no substring 1x1 or 1xy1.

```
n:     0 1 2 3 4  5  6  7  8  9 10  11  12  13  14  15   16   17
z(n):  1 2 4 6 8 11 17 27 41 60 88 132 200 301 449 669 1001 1502
```

The sequence is (apart from three leading ones) entry A079972 of [214] where two combinatorial interpretations are given:

```
Number of permutations satisfying -k<=p(i)-i<=r and p(i)-i not in I, i=1..n,
    with k=1, r=4, I={1,2}.
Number of compositions (ordered partitions) of n into elements of the set {1,4,5}.
```

The generating function is

$$\sum_{n=0}^{\infty} z(n)\,x^n \quad = \quad \frac{1+x+2\,x^2+2\,x^3+x^4}{1-x-x^4-x^5} \tag{12.9-4}$$

Chapter 13

Parenthesis strings

13.1 Co-lexicographic order

```
 1:  ((((()))))  11111.....      22:  (())(()())  11..11.1..
 2:  (((()())))  1111.1....      23:  ()()(()())  1.1.11.1..
 3:  (((())()))  111.11....      24:  ((()))(())  111...11..
 4:  ((()(())))  11.111....      25:  (()())(())  11.1..11..
 5:  ()((()()))  1.1111....      26:  ()(())(())  1.11..11..
 6:  (((())()))  1111..1...      27:  (())()(())  11..1.11..
 7:  (((()()()))  111.1.1...     28:  ()()()(())  1.1.1.11..
 8:  ((()(()()))  11.11.1...     29:  (((()))())  1111....1.
 9:  ()(((()())))  1.111.1...    30:  ((()))()()  111.1...1.
10:  (((())(())))  111..11...    31:  (()(()))()  11.11...1.
11:  ((()()(())))  11.1.11...    32:  ()((()))()  1.111...1.
12:  ()(()(()()))  1.11.11...    33:  ((())())()  111..1..1.
13:  (())((()()))  11..111...    34:  (()()())()  11.1.1..1.
14:  ()()(((())))  1.1.111...    35:  ()(()())()  1.11.1..1.
15:  ((((()))()))  1111...1..    36:  (())(())()  11..11..1.
16:  (((()())()))  111.1..1..    37:  ()()(())()  1.1.11..1.
17:  ((()(()))())  11.11..1..    38:  ((()))()()  111...1.1.
18:  ()(((()))())  1.111..1..    39:  (()())()()  11.1..1.1.
19:  (((())()())  111..1.1..     40:  ()(())()()  1.11..1.1.
20:  ((()()()()))  11.1.1.1..    41:  (())()()()  11..1.1.1.
21:  ()(()()()))  1.11.1.1..     42:  ()()()()()  1.1.1.1.1.
```

Figure 13.1-A: All (42) valid strings of 5 pairs of parenthesis in colex order.

An iterative scheme to generate all valid ways to group parenthesis can be obtained from a modified version of the combinations in co-lexicographic order (see section 6.1.2 on page 167). For $n = 5$ pairs the possible combinations are shown in figure 13.1-A. This is the output of [FXT: comb/paren-demo.cc].

Consider the sequences to the right of the paren strings as binary words. Whenever the leftmost block has more than one bit then its rightmost bit is moved one position to the right. If the leftmost block consists of a single bit then the bits of the longest run of the repeated pattern '`1.`' at the left are gathered at the left end and further, the rightmost bit in next block of ones (which contains at least two ones) is moved by one position to the right and the rest of the block is gathered at the left end (see the transitions from `#13` to `#14` or `#36` to `#37`).

A sentinel `x[k]` is used to save one branch with the generation of the next string [FXT: `class paren` in comb/paren.h]:

```
class paren
{
public:
    ulong k_;  // Number of paren pairs
    ulong n_;  // ==2*k
```

```
    ulong *x_; // Positions where a opening paren occurs
    char *str_;  // String representation,  e.g. "((())())()"
public:
    paren(ulong k)
    {
        k_ = (k>1 ? k : 2);  // not zero (empty) or one (trivial: "()")
        n_ = 2 * k_;
        x_ = new ulong[k_ + 1];
        x_[k_] = 999;  // sentinel
        str_ = new char[n_ + 1];
        str_[n_] = 0;
        first();
    }
    ~paren()
    {
        delete [] x_;
        delete [] str_;
    }
    void first()  { for (ulong i=0; i<k_; ++i)  x_[i] = i; }
    void last()  { for (ulong i=0; i<k_; ++i)  x_[i] = 2*i; }
    [--snip--]
```

The code for the computation of the successor and predecessor is quite concise:

```
    ulong next()  // return zero if current paren is the last
    {
        // if ( k_==1 )  return 0;  // uncomment to make algorithm work for k_==1
        ulong j = 0;
        if ( x_[1] == 2 )
        {
            // scan for low end == 010101:
            j = 2;
            while ( (j<=k_) && (x_[j]==2*j) )  ++j;  // can touch sentinel
            if ( j==k_ )  {  first();  return 0; }
        }
        // scan block:
        while ( 1 == (x_[j+1] - x_[j]) )  { ++j; }
        ++x_[j];  // move edge element up
        for (ulong i=0; i<j; ++i)  x_[i] = i; // attach block at low end
        return 1;
    }
    ulong prev()  // return zero if current paren is the first
    {
        // if ( k_==1 )  return 0;  // uncomment to make algorithm work for k_==1
        ulong j = 0;
        // scan for first gap:
        while ( x_[j]==j )  ++j;
        if ( j==k_ )  {  last();  return 0; }
        if ( x_[j]-x_[j-1] == 2 )  --x_[j];  // gap of length one
        else
        {
            ulong i = --x_[j];
            --j;
            --i;
            // j items to go, distribute as 1.1.1.11111
            for (  ;  2*i>j;  --i,--j)  x_[j] = i;
            for (  ; i; --i)  x_[i] = 2*i;
            x_[0] = 0;
        }
        return 1;
    }
    const ulong * data()  { return x_; }
    [--snip--]
```

The strings are set up on demand only:

```
    const char * string()  // generate on demand
```

```
    {
        for (ulong j=0; j<n_; ++j)  str_[j] = ')';
        for (ulong j=0; j<k_; ++j)  str_[x_[j]] = '(';
        return str_;
    }
};
```

The 477,638,700 paren words for $n = 18$ are generated at a rate of about 65 million objects per second. Section 1.30 on page 74 gives a bit-level algorithm for the generation of the paren words in colex order.

13.2 Gray code via restricted growth strings

```
 1:    [ 0, 0, 0, 0, ]    ()()()()    1.1.1.1.
 2:    [ 0, 0, 0, 1, ]    ()()(())    1.1.11..
 3:    [ 0, 0, 1, 0, ]    ()(())()    1.11..1.
 4:    [ 0, 0, 1, 1, ]    ()(()())    1.11.1..
 5:    [ 0, 0, 1, 2, ]    ()((()))    1.111...
 6:    [ 0, 1, 0, 0, ]    (())()()    11..1.1.
 7:    [ 0, 1, 0, 1, ]    (())(())    11..11..
 8:    [ 0, 1, 1, 0, ]    (()())()    11.1..1.
 9:    [ 0, 1, 1, 1, ]    (()()())    11.1.1..
10:    [ 0, 1, 1, 2, ]    (()(()))    11.11...
11:    [ 0, 1, 2, 0, ]    ((()))()    111...1.
12:    [ 0, 1, 2, 1, ]    ((())())    111..1..
13:    [ 0, 1, 2, 2, ]    ((()()))    111.1...
14:    [ 0, 1, 2, 3, ]    ((((?)))    1111....
```

Figure 13.2-A: Length-4 restricted growth strings (left), and the corresponding paren strings (middle) and delta sets (right).

The valid paren strings can be represented by sequences a_0, a_1, ..., a_n where $a_0 = 0$ and $a_k \leq a_{k-1} + 1$. These sequences are examples of so-called *restricted growth strings* (RGS). Some sources use the term *restricted growth functions*. The RGSs for $n = 4$ are shown in figure 13.2-A. A RGS can be incremented by incrementing the highest (rightmost in figure 13.2-A) digit a_j where $a_j \leq a_{j-1}$ and setting $a_i = 0$ for all $i > j$. A decrement is obtained by decrementing the highest digit $a_j \neq 0$ and setting $a_i = a_{i-1} + 1$ for all $i > j$.

The RGSs for a given n can be generated as follows [FXT: `class catalan` in comb/catalan.h]:

```
class catalan
// Catalan restricted growth strings (RGS)
// By default in near-perfect minimal-change order, i.e.
//  exactly two symbols in paren string change with each step
{
public:
    int *as_;   // digits of the RGS: as_[k] <= as[k-1] + 1
    int *d_;    // direction with recursion (+1 or -1)
    ulong n_;   // Number of digits (paren pairs)
    char *str_; // paren string
    bool xdr_;  // whether to change direction in recursion (==> minimal-change order)
    int dr0_;   // dr0: starting direction in each recursive step:
    //   dr0=+1  ==> start with as[]=[0,0,0,...,0]   == "()()()...()"
    //   dr0=-1  ==> start with as[]=[0,1,2,...,n-1] == "((( ... )))"

public:
    catalan(ulong n, bool xdr=true, int dr0=+1)
        : n_(n)
    {
        as_ = new int[n_];
        d_ = new int[n_];
        str_ = new char[2*n_+1];  str_[2*n_] = 0;
        init(xdr, dr0);
    }

    ~catalan()
    {
```

```
    1:  [ 0 1 2 3 4 ]    [ - - - - - ]    (((((-----)))))    11111.....
    2:  [ 0 1 2 3 3 ]    [ - - - - - ]    ((((()----)))) 1    1111.1....    ((((XA))))
    3:  [ 0 1 2 3 2 ]    [ - - - - - ]    (((()()---))) 1    1111..1...    ((()()XA))
    4:  [ 0 1 2 3 1 ]    [ - - - - - ]    (((())()-)) 1 1    1111...1..    ((())()XA))
    5:  [ 0 1 2 3 0 ]    [ - - - - - ]    (((()))()) 1 1 1    1111....1.    (((()))()XA)
    6:  [ 0 1 2 2 0 ]    [ - - - - + ]    ((()()))() 1 1 1    111.1...1.    (((XA)))()
    7:  [ 0 1 2 2 1 ]    [ - - - - + ]    ((()())()) 1 1 1    111.1..1..    ((()())AX)
    8:  [ 0 1 2 2 2 ]    [ - - - - + ]    ((()()()-)) 1 1 1    111.1.1...    ((()()AX))
    9:  [ 0 1 2 2 3 ]    [ - - - - + ]    ((()(()-)) 1 1 1 1    111.11....    ((()(AX)))
   10:  [ 0 1 2 1 2 ]    [ - - - - - ]    ((())()()) 1 1 1 1    111..11...    ((()X(A)))  2
   11:  [ 0 1 2 1 1 ]    [ - - - - - ]    ((())()-)) 1 1 1 1    111..1.1..    ((()))(XA))
   12:  [ 0 1 2 1 0 ]    [ - - - - - ]    ((())())() 1 1 1 1    111..1..1.    ((()))()XA)
   13:  [ 0 1 2 0 0 ]    [ - - - - + ]    ((()))()() 1 1 1 1    111...1.1.    ((()))XA)()
   14:  [ 0 1 2 0 1 ]    [ - - - - + ]    ((()))()-) 1 1 1 1    111...11..    ((()))(AX)
   15:  [ 0 1 1 0 1 ]    [ - - - + - ]    (()())()-) 1 1 1 1    11.1..11..    ((XA))(())
   16:  [ 0 1 1 0 0 ]    [ - - - + - ]    (()())()() 1 1 1 1    11.1..1.1.    (()())(XA)
   17:  [ 0 1 1 1 0 ]    [ - - - + + ]    (()()())() 1 1 1 1    11.1.1..1.    (()()AX)()
   18:  [ 0 1 1 1 1 ]    [ - - - + + ]    (()()()-)) 1 1 1 1    11.1.1.1..    (()()()AX)
   19:  [ 0 1 1 1 2 ]    [ - - - + + ]    (()()(-))) 1 1 1 1    11.1.11...    (()()(AX))
   20:  [ 0 1 1 2 3 ]    [ - - - + - ]    (()((-))) 1 1 1 1    11.111....    (()(A(X)))  2
   21:  [ 0 1 1 2 2 ]    [ - - - + - ]    (()()(-))) 1 1 1 1    11.11.1...    (()((XA)))
   22:  [ 0 1 1 2 1 ]    [ - - - + - ]    (()(())-)) 1 1 1 1    11.11..1..    (()(()XA))
   23:  [ 0 1 1 2 0 ]    [ - - - + - ]    (()(()))() 1 1 1 1    11.11...1.    (()(()))XA)
   24:  [ 0 1 0 1 0 ]    [ - - - - + ]    (())(())() 1 1 1 1    11..11..1.    (()X(A))()  2
   25:  [ 0 1 0 1 1 ]    [ - - - - + ]    (())(()-)) 1 1 1 1    11..11.1..    (())(()AX)
   26:  [ 0 1 0 1 2 ]    [ - - - - + ]    (())((-))) 1 1 1 1    11..111...    (())((AX))
   27:  [ 0 1 0 0 1 ]    [ - - - - - ]    (())()(()) 1 1 1 1    11..1.11..    (())(X(A))  2
   28:  [ 0 1 0 0 0 ]    [ - - - - - ]    (())()()() 1 1 1 1    11..1.1.1.    (())()(XA)
   29:  [ 0 0 0 0 0 ]    [ - - + + + ]    ()()()()() 1 1 1 1 1    1.1.1.1.1.    (XA)()()()
   30:  [ 0 0 0 0 1 ]    [ - - + + + ]    ()()()(()) 1 1 1 1 1    1.1.1.11..    ()()()(AX)
   31:  [ 0 0 0 1 2 ]    [ - - + + - ]    ()()((-))) 1 1 1 1 1    1.1.111...    ()()(A(X))  2
   32:  [ 0 0 0 1 1 ]    [ - - + + - ]    ()()(()-)) 1 1 1 1 1    1.1.11.1..    ()()((XA))
   33:  [ 0 0 0 1 0 ]    [ - - + + - ]    ()()(())() 1 1 1 1 1    1.1.11..1.    ()()(()XA)
   34:  [ 0 0 1 2 0 ]    [ - - + - + ]    ()((()))() 1 1 1 1 1    1.111...1.    ()(A(X))()  2
   35:  [ 0 0 1 2 1 ]    [ - - + - + ]    ()((())()) 1 1 1 1 1    1.111..1..    ()((()))AX)
   36:  [ 0 0 1 2 2 ]    [ - - + - + ]    ()((()()-)) 1 1 1 1 1    1.111.1...    ()((()))AX))
   37:  [ 0 0 1 2 3 ]    [ - - + - + ]    ()((()(-))) 1 1 1 1 1    1.1111....    ()(((AX)))
   38:  [ 0 0 1 1 2 ]    [ - - + - - ]    ()(()(-))) 1 1 1 1 1    1.11.11...    ()((X(A)))  2
   39:  [ 0 0 1 1 1 ]    [ - - + - - ]    ()(()()-)) 1 1 1 1 1    1.11.1.1..    ()(()(XA))
   40:  [ 0 0 1 1 0 ]    [ - - + - - ]    ()(()())() 1 1 1 1 1    1.11.1..1.    ()(()()XA)
   41:  [ 0 0 1 0 0 ]    [ - - + - + ]    ()(())()() 1 1 1 1 1    1.11..1.1.    ()(())XA)()
   42:  [ 0 0 1 0 1 ]    [ - - + - + ]    ()(())(()) 1 1 1 1 1    1.11..11..    ()(())(AX)
```

Figure 13.2-B: Minimal-change order for the paren strings of 5 pairs. From left to right: restricted growth strings, arrays of directions, paren strings, delta sets, and difference strings. If the change is not adjacent, then the distance of changed positions is given at the right. The order corresponds to `dr0=-1`.

```
 1:  [ 0 0 0 0 0 0 ]    [ + + + + + + ]    ()()()()()()    1.1.1.1.1.1.
 2:  [ 0 0 0 0 0 1 ]    [ + + + + + + ]    ()()()()(())    1.1.1.1.11..    ()()()(AX)
 3:  [ 0 0 0 1 2 2 ]    [ + + + + - + ]    ()()((()))    1.1.111...    ()()(A(X))  2
 4:  [ 0 0 0 1 2 1 ]    [ + + + + - + ]    ()()(()())    1.1.11.1..    ()()((XA))
 5:  [ 0 0 0 1 2 0 ]    [ + + + + - + ]    ()()(())()    1.1.11..1.    ()()(()XA)
 6:  [ 0 0 1 2 3 0 ]    [ + + + - + + ]    ()((()))()    1.111...1.    ()(A(X))()  2
 7:  [ 0 0 1 2 2 1 ]    [ + + + - + + ]    ()((())())    1.111..1..    ()((()))AX)
 8:  [ 0 0 1 2 2 2 ]    [ + + + - + + ]    ()((()()))    1.111.1...    ()((()AX))
 9:  [ 0 0 1 2 3 3 ]    [ + + + - + + ]    ()(((())))    1.1111....    ()(((AX)))
10:  [ 0 0 1 1 2 2 ]    [ + + + - - + ]    ()(()(()))    1.11.11...    ()((X(A)))  2
11:  [ 0 0 1 1 1 1 ]    [ + + + - - + ]    ()(()()())    1.11.1.1..    ()(()(XA))
12:  [ 0 0 1 1 0 0 ]    [ + + + - - + ]    ()(()())()    1.11.1..1.    ()(()()XA)
13:  [ 0 0 1 0 0 0 ]    [ + + + - + + ]    ()(())()()    1.11..1.1.    ()(()XA)()
14:  [ 0 0 1 0 1 1 ]    [ + + + - + + ]    ()(())(())    1.11..11..    ()(())(AX)
15:  [ 0 1 2 0 1 1 ]    [ + + - + - + ]    ((()))(())    111...11..    (A(X))(())  2
16:  [ 0 1 2 0 0 0 ]    [ + + - + - + ]    ((()))()()    111...1.1.    ((()))(XA)
17:  [ 0 1 2 1 0 0 ]    [ + + - + + + ]    ((())())()    111..1..1.    ((()))AX)()
18:  [ 0 1 2 1 1 1 ]    [ + + - + + + ]    ((())()())    111..1.1..    ((()))()AX)
19:  [ 0 1 2 1 2 2 ]    [ + + - + + + ]    ((()))(()))    111..11...    ((()))(AX))
20:  [ 0 1 2 2 3 3 ]    [ + + - + - + ]    ((()(())))    111.11....    ((()A(X)))  2
21:  [ 0 1 2 2 2 2 ]    [ + + - + - + ]    ((()()()))    111.1.1...    ((()(XA)))
22:  [ 0 1 2 2 1 1 ]    [ + + - + - + ]    ((()())())    111.1..1..    ((()()XA))
23:  [ 0 1 2 2 0 0 ]    [ + + - + - + ]    ((()()))()    111.1...1.    ((()())XA)
24:  [ 0 1 2 3 0 0 ]    [ + + - + + + ]    ((())))()    1111....1.    (((AX)))()
25:  [ 0 1 2 3 1 1 ]    [ + + - + + + ]    ((()))(()))    1111...1..    ((()))(AX))
26:  [ 0 1 2 3 2 2 ]    [ + + - + + + ]    ((())(())))    1111..1...    ((()))AX))
27:  [ 0 1 2 3 3 3 ]    [ + + - + + + ]    ((()()()))    1111.1....    ((()(AX)))
28:  [ 0 1 2 3 4 4 ]    [ + + - + + + ]    ((((()))))    11111.....    ((((AX))))
29:  [ 0 1 1 2 3 3 ]    [ + + - - - + ]    (()((())))    11.111....    ((X((A))))  3
30:  [ 0 1 1 2 2 2 ]    [ + + - - - + ]    (()(()()))    11.11.1...    (()((XA)))
31:  [ 0 1 1 2 1 1 ]    [ + + - - - + ]    (()(())())    11.11..1..    (()(()XA))
32:  [ 0 1 1 2 0 0 ]    [ + + - - - + ]    (()(()))()    11.11...1.    (()(()))XA)
33:  [ 0 1 1 1 1 0 ]    [ + + - - + + ]    (()()()())    11.1.1.1.    (()(XA))()
34:  [ 0 1 1 1 1 1 ]    [ + + - - + + ]    (()()()())    11.1.1.1..    (()()()AX)
35:  [ 0 1 1 1 2 2 ]    [ + + - - + + ]    (()()(()))    11.1.11...    (()()(AX))
36:  [ 0 1 1 0 1 1 ]    [ + + - - - + ]    (()())(())    11.1..11..    (()()X(A))  2
37:  [ 0 1 1 0 0 0 ]    [ + + - - - + ]    (()())()()    11.1..1.1.    (()())(XA)
38:  [ 0 1 0 0 0 0 ]    [ + + - + + + ]    (())()()()    11..1.1.1.    (()XA)()()
39:  [ 0 1 0 0 1 1 ]    [ + + - + + + ]    (())()(())    11..1.11..    (())()(AX)
40:  [ 0 1 0 1 2 2 ]    [ + + - + + - ]    (())((()))    11..111...    (())(A(X))  2
41:  [ 0 1 0 1 1 1 ]    [ + + - + + - ]    (())(()())    11..11.1..    (())((XA))
42:  [ 0 1 0 1 0 1 ]    [ + + - + + - ]    (())(())()    11..11..1.    (())(()XA)
```

Figure 13.2-C: Minimal-change order for the paren strings of 5 pairs. From left to right: restricted growth strings, arrays of directions, paren strings, delta sets, and difference strings. If the change is not adjacent, then the distance of changed positions is given at the right. The order corresponds to `dr0=+1`.

```
        delete [] as_;
        delete [] d_;
        delete [] str_;
    }

    void init(bool xdr, int dr0)
    {
        dr0_ = ( (dr0>=0) ? +1 : -1 );
        xdr_ = xdr;

        ulong n = n_;
        if ( dr0_>0 )  for (ulong k=0; k<n; ++k)  as_[k] = 0;
        else           for (ulong k=0; k<n; ++k)  as_[k] = k;

        for (ulong k=0; k<n; ++k)  d_[k] = dr0_;
    }

    bool next()  { return next_rec(n_-1); }

    const int *get()  const  { return as_; }

    const char* str()  { make_str();  return (const char*)str_; }

   [--snip--]
```

The minimal-change order is obtained by changing the 'direction' in the recursion, an essentially identical mechanism (for the generation of set partitions) is shown in chapter 15 on page 319. The function is given in [FXT: comb/catalan.cc]:

```
bool
catalan::next_rec(ulong k)
{
    if ( k<1 )  return false;  // current is last

    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>as_[k-1]+1) : (as<0) );
    if ( ovq )  // have to recurse
    {
        ulong ns1 = next_rec(k-1);
        if ( 0==ns1 )  return false;

        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;

        as = ( (d>0) ? 0 : as_[k-1]+1 );
    }
    as_[k] = as;

    return true;
}
```

The program [FXT: comb/catalan-demo.cc] demonstrates the usage:

```
    ulong n = 4;
    bool xdr = true;
    int dr0 = -1;

    catalan C(n, xdr, dr0);
    do
    {
        // visit
    }
    while ( C.next() );
```

About 67 million strings per second are generated. Figure 13.2-B shows the minimal-change for $n = 5$ and `dr0=-1`, figure 13.2-C shows the order for `dr0=+1`.

More minimal-change orders

The Gray code order shown in figure 13.2-D can be generated via a simple recursion:

```
ulong n;    // Number of paren pairs
ulong *rv;  // restricted growth strings

void next_rec(ulong d, bool z)
{
    if ( d==n )  visit();
    else
```

```
 1:   0 0 0 0 0    1.1.1.1.1.        22:   0 1 2 2 1    111.1..1..
 2:   0 0 0 0 1    1.1.1.11..        23:   0 1 2 2 0    111.1...1.
 3:   0 0 0 1 2    1.1.111...        24:   0 1 2 1 0    111..1..1.
 4:   0 0 0 1 1    1.1.11.1..        25:   0 1 2 1 1    111..1.1..
 5:   0 0 0 1 0    1.1.11..1.        26:   0 1 2 1 2    111..11...
 6:   0 0 1 2 3    1.1111....        27:   0 1 2 0 1    111...11..
 7:   0 0 1 2 2    1.111.1...        28:   0 1 2 0 0    111...1.1.
 8:   0 0 1 2 1    1.111..1..        29:   0 1 1 0 0    11.1..1.1.
 9:   0 0 1 2 0    1.111...1.        30:   0 1 1 0 1    11.1..11..
10:   0 0 1 1 0    1.11.1..1.        31:   0 1 1 1 2    11.1.11...
11:   0 0 1 1 1    1.11.1.1..        32:   0 1 1 1 1    11.1.1.1..
12:   0 0 1 1 2    1.11.11...        33:   0 1 1 1 0    11.1.1..1.
13:   0 0 1 0 1    1.11..11..        34:   0 1 1 2 0    11.11...1.
14:   0 0 1 0 0    1.11..1.1.        35:   0 1 1 2 1    11.11..1..
15:   0 1 2 3 0    1111....1.        36:   0 1 1 2 2    11.11.1...
16:   0 1 2 3 1    1111...1..        37:   0 1 1 2 3    11.111....
17:   0 1 2 3 2    1111..1...        38:   0 1 0 1 0    11..11..1.
18:   0 1 2 3 3    1111.1....        39:   0 1 0 1 1    11..11.1..
19:   0 1 2 3 4    11111.....        40:   0 1 0 1 2    11..111...
20:   0 1 2 2 3    111.11....        41:   0 1 0 0 1    11..1.11..
21:   0 1 2 2 2    111.1.1...        42:   0 1 0 0 0    11..1.1.1.
```

Figure 13.2-D: Strings of 5 pairs of parenthesis in a Gray code order as generated by a recursive algorithm.

```
    {
        const long rv1 = rv[d-1];  // left neighbor
        if ( 0==z )
        {
            for (long x=0; x<=rv1+1; ++x)  // forward
            {
                rv[d] = x;
                next_rec(d+1, (x&1));
            }
        }
        else
        {
            for (long x=rv1+1; x>=0; --x)  // backward
            {
                rv[d] = x;
                next_rec(d+1, !(x&1));
            }
        }
    }
}
```

The initial call is `next_rec(0, 0);`. About 81 million strings per second are generated [FXT: comb/paren-gray-rec-demo.cc].

```
 1:  ()()()()()    1.1.1.1.1.        22:  (()()(()))    11.1.11...
 2:  ()()()(())    1.1.1.11..        23:  (()())()()    11.1.1..1.
 3:  ()()(()())    1.1.11.1..        24:  ((()))()()    111..1..1.
 4:  ()()((()))    1.1.111...        25:  ((()))(())    111..11...
 5:  ()()(())()    1.1.11..1.        26:  ((())()())    111..1.1..
 6:  ()(()())()    1.11.1..1.        27:  ((()())())    111.1..1..
 7:  ()(()(()))    1.11.11...        28:  ((()()()))    111.1.1...
 8:  ()(()()())    1.11.1.1..        29:  ((())(()))    111.11....
 9:  ()((()))()    1.111..1..        30:  ((())())()    111.1...1.
10:  ()((()()))    1.111.1...        31:  (((()))())    1111....1.
11:  ()(((())))    1.1111....        32:  ((((()))))    11111.....
12:  ()((()))()    1.111...1.        33:  ((()(())))    1111.1....
13:  ()(())()()    1.11..1.1.        34:  ((())(()))    1111..1...
14:  ()(())(())    1.11..11..        35:  ((()))()())    1111...1..
15:  (()())(())    11.1..11..        36:  ((()))(()))    111...11..
16:  (()())()()    11.1..1.1.        37:  ((()))()()    111...1.1.
17:  (()(()))()    11.11...1.        38:  (())()()()    11..1.1.1.
18:  (()((())))    11.111....        39:  (())()(()))    11..1.11..
19:  (()(()()))    11.11.1...        40:  (())(()()))    11..11.1..
20:  (()(())())    11.11..1..        41:  (())((()))    11..111...
21:  (()()()())    11.1.1.1..        42:  (())(())()    11..11..1.
```

Figure 13.2-E: Strings of 5 pairs of parenthesis in Gray code order as generated by a loopless algorithm.

An loopless algorithm (that does not use RGS) given in [225] is implemented in [FXT: `class paren_gray` in comb/paren-gray.h]. The generated order for five paren pairs is shown in figure 13.2-E About 54 million strings per second are generated [FXT: comb/paren-gray-demo.cc]. Still more algorithms for the parentheses strings in minimal-change order are given in [69], [232], and [249].

```
 0:   ....1111 == (((()))) 
 1:   ...1.111 == (((.))))   ^= ...11...
 2:   ...11.11 == ((.(.)))   ^= ....11..
 3:   ...111.1 == (.(((.)))  ^= .....11.
 4:   ..1.11.1 == (.(.(.))   ^= ..11....
 5:   ..1.1.11 == ((.(.).)   ^= .....11.
 6:   ..1..111 == (((.)).)   ^= ....11..
 7:   .1...111 == (((.))).   ^= .11.....
 8:   .1..1.11 == ((.(.).)   ^= ....11..
 9:   .1..11.1 == (.(((.).   ^= .....11.
10:   .1.1.1.1 == (.(.(.).   ^= ...11...
11:   ..11.1.1 == (.(.((.   ^= .11.....
12:   ..11..11 == ((.((.)   ^= .....11.
13:   .1.1..11 == ((.(.(.   ^= .11.....
```

Figure 13.2-F: A strong minimal-change order for the paren strings of 4 pairs.

For even values of n it is possible to generate paren strings in *strong minimal-change order* where changes occur only in adjacent positions. Figure 13.2-F shows an example for four pairs of parens. The listing was generated with [FXT: graph/graph-parengray-demo.cc] that uses directed graphs and the search algorithms described in chapter 19 on page 355.

13.3 The number of parenthesis strings: Catalan numbers

The number of valid combinations of n parenthesis pairs is

$$C_n \;=\; \frac{\binom{2\,n}{n}}{n+1} \;=\; \frac{\binom{2\,n+1}{n}}{2\,n+1} \;=\; \frac{\binom{2\,n-2}{n-1}}{n} \;=\; \binom{2\,n}{n} - \binom{2\,n}{n-1} \tag{13.3-1}$$

as nicely explained in [124, p.343-346]. These are the *Catalan numbers*, sequence A000108 of [214]:

n:	C_n	n:	C_n	n:	C_n
1:	1	11:	58786	21:	24466267020
2:	2	12:	208012	22:	91482563640
3:	5	13:	742900	23:	343059613650
4:	14	14:	2674440	24:	1289904147324
5:	42	15:	9694845	25:	4861946401452
6:	132	16:	35357670	26:	18367353072152
7:	429	17:	129644790	27:	69533550916004
8:	1430	18:	477638700	28:	263747951750360
9:	4862	19:	1767263190	29:	1002242216651368
10:	16796	20:	6564120420	30:	3814986502092304

The Catalan numbers are generated most easily with the relation

$$C_{n+1} \;=\; \frac{2\,(2\,n+1)}{n+2}\,C_n \tag{13.3-2}$$

The generating function is

$$C(x) \;=\; \frac{1-\sqrt{1-4\,x}}{2\,x} \;=\; \sum_{n=0}^{\infty} C_n\,x^n \;=\; 1 + x + 2\,x^2 + 5\,x^3 + 14\,x^4 + 42\,x^5 + \ldots \tag{13.3-3}$$

One further has the convolution property $[x\,C(x)] = x + [x\,C(x)]^2$

$$C_n \;=\; \sum_{k=0}^{n-1} C_k\,C_{n-1-k} \tag{13.3-4}$$

13.4 Increment-i RGS and k-ary trees

13.4.1 Generation in lexicographic order

```
       1:  [ 0 0 0 0 ]      21:  [ 0 1 2 1 ]      41:  [ 0 2 2 3 ]
       2:  [ 0 0 0 1 ]      22:  [ 0 1 2 2 ]      42:  [ 0 2 2 4 ]
       3:  [ 0 0 0 2 ]      23:  [ 0 1 2 3 ]      43:  [ 0 2 3 0 ]
       4:  [ 0 0 1 0 ]      24:  [ 0 1 2 4 ]      44:  [ 0 2 3 1 ]
       5:  [ 0 0 1 1 ]      25:  [ 0 1 3 0 ]      45:  [ 0 2 3 2 ]
       6:  [ 0 0 1 2 ]      26:  [ 0 1 3 1 ]      46:  [ 0 2 3 3 ]
       7:  [ 0 0 1 3 ]      27:  [ 0 1 3 2 ]      47:  [ 0 2 3 4 ]
       8:  [ 0 0 2 0 ]      28:  [ 0 1 3 3 ]      48:  [ 0 2 3 5 ]
       9:  [ 0 0 2 1 ]      29:  [ 0 1 3 4 ]      49:  [ 0 2 4 0 ]
      10:  [ 0 0 2 2 ]      30:  [ 0 1 3 5 ]      50:  [ 0 2 4 1 ]
      11:  [ 0 0 2 3 ]      31:  [ 0 2 0 0 ]      51:  [ 0 2 4 2 ]
      12:  [ 0 0 2 4 ]      32:  [ 0 2 0 1 ]      52:  [ 0 2 4 3 ]
      13:  [ 0 1 0 0 ]      33:  [ 0 2 0 2 ]      53:  [ 0 2 4 4 ]
      14:  [ 0 1 0 1 ]      34:  [ 0 2 1 0 ]      54:  [ 0 2 4 5 ]
      15:  [ 0 1 0 2 ]      35:  [ 0 2 1 1 ]      55:  [ 0 2 4 6 ]
      16:  [ 0 1 1 0 ]      36:  [ 0 2 1 2 ]
      17:  [ 0 1 1 1 ]      37:  [ 0 2 1 3 ]
      18:  [ 0 1 1 2 ]      38:  [ 0 2 2 0 ]
      19:  [ 0 1 1 3 ]      39:  [ 0 2 2 1 ]
      20:  [ 0 1 2 0 ]      40:  [ 0 2 2 2 ]
```

Figure 13.4-A: The 55 increment-2 restricted growths strings of length 4.

We now allow an increment of i in the restricted growth strings ($i = 1$ corresponds to the paren RGS of section 13.2). Figure 13.4-A shows the increment-2 restricted growths strings of length 4. The strings can be generated in lexicographic order via [FXT: `class rgs_binomial` in comb/rgs-binomial.h].

```
class rgs_binomial
// Restricted growth strings (RGS) s[0,...,n-1] so that s[k] <= s[k-1]+i
{
public:
    ulong *s_;  // restricted growth string
    ulong n_;   // Length of strings
    ulong i_;   // s[k] <= s[k-1]+i
  [--snip--]

    ulong next()
    // Return index of first changed element in s[],
    // Return zero if current string is the last
    {
        ulong k = n_;

    start:
        --k;
        if ( k==0 )  return 0;

        ulong sk = s_[k] + 1;
        ulong mp = s_[k-1] + i_;
        if ( sk > mp )  // "carry"
        {
            s_[k] = 0;
            goto start;
        }

        s_[k] = sk;
        return k;
    }
 [--snip--]
```

The rate of generation is about 129 M/s for $i = 1$ (corresponding to paren strings), 143 M/s for $i = 2$, and 156 M/s with $i = 3$ [FXT: comb/rgs-binomial-demo.cc].

13.4.2 The number of increment-i RGS

The number $C_{n,i}$ of length-n increment-i strings equals

$$C_{n,i} \quad = \quad \frac{\binom{(i+1)\,n}{n}}{i\,n+1} \tag{13.4-1}$$

A recursion generalizing relation 13.3-2 is

$$C_{n+1,i} \quad = \quad (i+1)\,\frac{\prod_{k=1}^{i}\left[(i+1)\,n+k\right]}{\prod_{k=1}^{i}\left[i\,n+k+1\right]}\,C_{n,i} \tag{13.4-2}$$

The sequences of numbers of length-n strings for $i = 1, 2, 3, 4$ start

```
     n: 1 2  3   4    5     6      7       8        9         10          11
  i=1:  1 2  5  14   42   132    429    1430     4862     16796       58786
  i=2:  1 3 12  55  273  1428   7752   43263   246675   1430715     8414640
  i=3:  1 4 22 140  969  7084  53820  420732  3362260  27343888   225568798
  i=4:  1 5 35 285 2530 23751 231880 2330445 23950355 250543370 2658968130
```

These are respectively the entries A000108, A001764, A002293, A002294 of [214] where combinatorial interpretations are given. We note that for the generating function $C_i(x)$ we have the following expression as a hypergeometric function (see section 35.2 on page 663):

$$C_{i-1}(x) \quad = \quad \sum_{n=0}^{\infty} C_{n,i-1}\,x^n \tag{13.4-3a}$$

$$\quad = \quad F\left(\begin{matrix} 1/(i+1),\ 2/(i+1),\ 3/(i+1),\ \ldots,\ (i+1)/(i+1) \\ 2/i,\ 3/i,\ \ldots,\ i/i,\ (i+1)/i \end{matrix}\,\middle|\,\frac{(i+1)^{(i+1)}}{i^i}\,x\right) \tag{13.4-3b}$$

Note that the last upper and second last lower parameter cancel. Now let $f_i(x) := x\,C_i(x^i)$, then

$$f_i(x) - f_i(x)^{i+1} \quad = \quad x \tag{13.4-4}$$

That is, $f_i(x)$ can be obtained as the series reversion of $x^{i+1} - x$. We choose $i = 2$ for an example:

```
? t1=serreverse(x-x^3+O(x^(17)))
  x + x^3 + 3*x^5 + 12*x^7 + 55*x^9 + 273*x^11 + 1428*x^13 + 7752*x^15 + O(x^17)
? t2=hypergeom([1/3,2/3,3/3],[2/2,3/2],3^3/2^2*x)+O(x^17)
  1 + x + 3*x^2 + 12*x^3 + 55*x^4 + 273*x^5 + 1428*x^6 + 7752*x^7 + ... + O(x^17)
? f=x*subst(t2,x,x^2);
? t1-f
  O(x^17)      \\ f is actually the series reversion of x-x^3
? f-f^3
  x + O(x^35)  \\ ... so  f - f^3 == id
```

We further have the following convolution property, generalizing relation 13.3-4 on the previous page,

$$C_{n,i} \quad = \quad \sum_{j_1+j_2+\ldots+j_i+j_{(i+1)}=n-1} C_{j_1,i}\,C_{j_2,i}\,C_{j_3,i}\,\cdots\,C_{j_i,i}\,C_{j_{(i+1)},i} \tag{13.4-5}$$

13.4.3 Gray code for k-ary trees

The length-n increment-i RGS correspond to k-ary trees with n internal nodes and $k = i+1$. An loopless algorithm for the generation of a Gray code for k-ary tress with only homogeneous changes is given in [28]. The RGS used in the algorithm gives the positions (one-based) of the ones in the delta sets, see figure 13.4-B. An implementation is [FXT: `class tree_gray` in comb/tree-gray.h]:

```
          positions      direction      delta set
 1:   [ 1 4 7 A ]    [ + + + + ]    1..1..1..1..
 2:   [ 1 4 7 8 ]    [ + + + + ]    1..1..11....
 3:   [ 1 4 7 9 ]    [ + + + - ]    1..1..1.1...
 4:   [ 1 4 5 9 ]    [ + + + - ]    1..11...1...
 5:   [ 1 4 5 8 ]    [ + + + - ]    1..11..1....
 6:   [ 1 4 5 7 ]    [ + + + - ]    1..11.1.....
 7:   [ 1 4 5 6 ]    [ + + + - ]    1..111......
 8:   [ 1 4 5 A ]    [ + + + + ]    1..11....1..
 9:   [ 1 4 6 A ]    [ + + - + ]    1..1.1...1..
10:   [ 1 4 6 7 ]    [ + + - + ]    1..1.11.....
11:   [ 1 4 6 8 ]    [ + + - + ]    1..1.1.1....
12:   [ 1 4 6 9 ]    [ + + - - ]    1..1.1..1...
13:   [ 1 2 6 9 ]    [ + + - - ]    11...1..1...
14:   [ 1 2 6 8 ]    [ + + - - ]    11...1.1....
15:   [ 1 2 6 7 ]    [ + + - - ]    11...11.....
16:   [ 1 2 6 A ]    [ + + - + ]    11...1...1..
17:   [ 1 2 5 A ]    [ + + - + ]    11..1....1..
18:   [ 1 2 5 6 ]    [ + + - + ]    11..11......
19:   [ 1 2 5 7 ]    [ + + - + ]    11..1.1.....
20:   [ 1 2 5 8 ]    [ + + - + ]    11..1..1....
21:   [ 1 2 5 9 ]    [ + + - - ]    11..1...1...
22:   [ 1 2 4 9 ]    [ + + - - ]    11.1....1...
23:   [ 1 2 4 8 ]    [ + + - - ]    11.1...1....
24:   [ 1 2 4 7 ]    [ + + - - ]    11.1..1.....
25:   [ 1 2 4 6 ]    [ + + - - ]    11.1.1......
26:   [ 1 2 4 5 ]    [ + + - - ]    11.11.......
27:   [ 1 2 4 A ]    [ + + - + ]    11.1.....1..
28:   [ 1 2 3 A ]    [ + + - + ]    111......1..
29:   [ 1 2 3 4 ]    [ + + - + ]    1111........
30:   [ 1 2 3 5 ]    [ + + - + ]    111.1.......
31:   [ 1 2 3 6 ]    [ + + - + ]    111..1......
32:   [ 1 2 3 7 ]    [ + + - + ]    111...1.....
33:   [ 1 2 3 8 ]    [ + + - + ]    111....1....
34:   [ 1 2 3 9 ]    [ + + - - ]    111.....1...
35:   [ 1 2 7 9 ]    [ + + + - ]    11.....1.1...
36:   [ 1 2 7 8 ]    [ + + + - ]    11.....11....
37:   [ 1 2 7 A ]    [ + + + + ]    11.....1..1..
38:   [ 1 3 7 A ]    [ + - + + ]    1.1...1..1..
39:   [ 1 3 7 8 ]    [ + - + + ]    1.1...11....
40:   [ 1 3 7 9 ]    [ + - + - ]    1.1...1.1...
41:   [ 1 3 4 9 ]    [ + - + - ]    1.11....1...
42:   [ 1 3 4 8 ]    [ + - + - ]    1.11...1....
43:   [ 1 3 4 7 ]    [ + - + - ]    1.11..1.....
44:   [ 1 3 4 6 ]    [ + - + - ]    1.11.1......
45:   [ 1 3 4 5 ]    [ + - + - ]    1.111.......
46:   [ 1 3 4 A ]    [ + - + + ]    1.11.....1..
47:   [ 1 3 5 A ]    [ + - + + ]    1.1.1....1..
48:   [ 1 3 5 6 ]    [ + - + + ]    1.1.11......
49:   [ 1 3 5 7 ]    [ + - + + ]    1.1.1.1.....
50:   [ 1 3 5 8 ]    [ + - + + ]    1.1.1..1....
51:   [ 1 3 5 9 ]    [ + - + - ]    1.1.1...1...
52:   [ 1 3 6 9 ]    [ + - - - ]    1.1..1..1...
53:   [ 1 3 6 8 ]    [ + - - - ]    1.1..1.1....
54:   [ 1 3 6 7 ]    [ + - - - ]    1.1..11.....
55:   [ 1 3 6 A ]    [ + - - + ]    1.1..1...1..
```

Figure 13.4-B: Gray code for 3-ary trees with 4 internal nodes with all changes being homogeneous. The left column shows the vectors of (one-based) positions, the symbol '`A`' is used for the number 10.

```
class tree_gray
{
public:
    ulong *sq_;  // sequence of bit positions (seq[])  elements \in {1,2,...,n}
    ulong *dr_;  // aux: direction            (dir[])
    ulong *np_;  // aux: next position        (nextPos[])
    ulong *mx_;  // aux: max position         (max[])
    ulong n_;    // n (internal) nodes
    ulong k_;    // k-ary tree

    tree_gray(ulong n, ulong k)
    {
        n_ = n;
        k_ = k;

        // all arrays are one-based
        sq_ = new ulong[n_+1];
        dr_ = new ulong[n_+1];
        np_ = new ulong[n_+2];  // one pad element right
        mx_ = new ulong[n_+1];  // unchanged in next()
        first();
    }
  [--snip--]

    void first(ulong k=0)
    {
        if ( k )  k_ = k;
        for (ulong j=1, e=1; j<=n_; ++j, e+=k_)  sq_[j] = mx_[j] = e;
        for (ulong j=0; j<=n_; ++j)  dr_[j] = 1;  // "right"
        for (ulong j=0; j<=n_+1; ++j)  np_[j] = j - 1;
    }
```

The computation of the successor is a variant of the method given in [41]:

```
    ulong next()
    {
        ulong i = np_[n_+1];

        if ( i==1 )  return 0;  // current string is last

        if ( dr_[i]==1 )  // direction == "right"
        {
            if ( sq_[i] == mx_[i] )   sq_[i] = sq_[i-1] + 1;
            else                      sq_[i] += 1;

            if ( sq_[i] == mx_[i] - 1 )
            {
                np_[i+1] = np_[i];  // can access element n+1
                np_[i] = i - 1;
                dr_[i] = -1UL;      // "left"
            }
        }
        else
        {
            if ( sq_[i] == sq_[i-1] + 1 )
            {
                sq_[i] = mx_[i];
                dr_[i] = 1;         // "right"
                np_[i+1] = np_[i];  // can access element n+1
                np_[i] = i - 1;
            }
            else  sq_[i] -= 1;
        }

        if ( i<n_ )  np_[n_+1] = n_;

        return i - 1;
    }
};
```

The rate of generation is about 97 M/s for 2-ary trees (corresponding to Catalan strings), 120 M/s for 3-ary trees, and 139 M/s with 4-ary trees [FXT: comb/tree-gray-demo.cc].

Chapter 14

Integer partitions

```
 1:    6 ==  6* 1 + 0    + 0    + 0    + 0    + 0     ==   1 + 1 + 1 + 1 + 1 + 1
 2:    6 ==  4* 1 + 1* 2 + 0    + 0    + 0    + 0     ==   1 + 1 + 1 + 1 + 2
 3:    6 ==  2* 1 + 2* 2 + 0    + 0    + 0    + 0     ==   1 + 1 + 2 + 2
 4:    6 ==  0    + 3* 2 + 0    + 0    + 0    + 0     ==   2 + 2 + 2
 5:    6 ==  3* 1 + 0    + 1* 3 + 0    + 0    + 0     ==   1 + 1 + 1 + 3
 6:    6 ==  1* 1 + 1* 2 + 1* 3 + 0    + 0    + 0     ==   1 + 2 + 3
 7:    6 ==  0    + 0    + 2* 3 + 0    + 0    + 0     ==   3 + 3
 8:    6 ==  2* 1 + 0    + 0    + 1* 4 + 0    + 0     ==   1 + 1 + 4
 9:    6 ==  0    + 1* 2 + 0    + 1* 4 + 0    + 0     ==   2 + 4
10:    6 ==  1* 1 + 0    + 0    + 0    + 1* 5 + 0     ==   1 + 5
11:    6 ==  0    + 0    + 0    + 0    + 0    + 1* 6  ==   6
```

Figure 14.0-A: All (eleven) integer partitions of 6.

An integer x is the sum of the positive integers less or equal to itself in various ways. The decompositions into sums of integers are called the *integer partitions* of the number x. Figure 14.0-A shows all integer partitions of $x = 6$.

14.1 Recursive solution of a generalized problem

We can solve a slightly more general problem and find all partitions of a number x with respect to a set $V = \{v_0, v_1, \ldots, v_{n-1}\}$ (where $v_i > 0$), that is all decompositions of the form $x = \sum_{k=0}^{n-1} c_k \cdot v_k$ (where $c_i \geq 0$). The integer partitions are the special case $V = \{1, 2, 3, \ldots, n\}$.

The algorithm to generate the partitions is to assign to the first bucket r_0 an integer multiple of the first set element v_0: $r_0 = c \cdot v_0$ (this has to be done for all $c \geq 0$ for which $r_0 \leq x$). Now set $c_0 = c$. If $r_0 = x$ we already found a partition (consisting of c_0 only), else (if $r_0 < x$) solve the remaining problem where $x' := x - c_0 \cdot v_0$ and $V' := \{v_1, v_2, \ldots, v_{n-1}\}$.

A C++ class for the generation of all partitions is [FXT: `class partition_rec` in comb/partition-rec.h]:

```
class partition_rec
// Integer partitions of x into supplied values pv[0],...,pv[n-1].
// pv[] defaults to [1,2,3,...,x]
{
public:
    ulong ct_;  // Number of partitions found so far
    ulong n_;   // Number of values
    ulong i_;   // level in iterative search

    long *pv_;  // values into which to partition
    ulong *pc_; // multipliers for values
    ulong pci_; // temporary for pc_[i_]
    long *r_;   // rest
    long ri_;   // temporary for r_[i_]
```

```
    long x_;    // value to partition
public:
    partition_rec(ulong x, ulong n=0, const ulong *pv=0)
    {
        if ( 0==n )  n = x;
        n_ = n;
        pv_ = new long[n_+1];
        if ( pv )  for (ulong j=0; j<n_; ++j)  pv_[j] = pv[j];
        else       for (ulong j=0; j<n_; ++j)  pv_[j] = j + 1;
        pc_ = new ulong[n_+1];
        r_  = new long[n_+1];
        init(x);
    }

    void init(ulong x)
    {
        x_ = x;
        ct_ = 0;
        for (ulong k=0; k<n_; ++k)  pc_[k] = 0;
        for (ulong k=0; k<n_; ++k)  r_[k] = 0;
        r_[n_-1] = x_;
        r_[n_] = x_;
        i_ = n_ - 1;
        pci_ = 0;
        ri_ = x_;
    }

    ~partition_rec()
    {
        delete [] pv_;
        delete [] pc_;
        delete [] r_;
    }

    ulong next();  // generate next partition
    ulong next_func(ulong i);  // aux

    [--snip--]
};
```

The routine to obtains the next partition is given in [FXT: comb/partition-rec.cc], it is actually an iterative version of the algorithm:

```
ulong
partition_rec::next()
{
    if ( i_>=n_ )  return  n_;

    r_[i_] = ri_;
    pc_[i_] = pci_;
    i_ = next_func(i_);

    for (ulong j=0; j<i_; ++j)  pc_[j] = r_[j] = 0;

    ++i_;
    ri_ = r_[i_] - pv_[i_];
    pci_ = pc_[i_] + 1;

    return  i_ - 1;  // >=0
}

ulong
partition_rec::next_func(ulong i)
{
 start:
    if ( 0!=i )
    {
        while ( r_[i]>0 )
        {
            pc_[i-1] = 0;
            r_[i-1] = r_[i];
            --i;  goto start;   // iteration
        }
    }
    else  // iteration end
    {
        if ( 0!=r_[i] )
        {
            long d = r_[i] / pv_[i];
```

```
            r_[i] -= d * pv_[i];
            pc_[i] = d;
        }
    }

    if ( 0==r_[i] )  // valid partition found
    {
        ++ct_;
        return i;
    }

    ++i;
    if ( i>=n_ )  return n_;  // search finished

    r_[i] -= pv_[i];
    ++pc_[i];

    goto start;  // iteration
}
```

The routines can easily adapted to the generation of partitions satisfying certain restrictions, for example, partitions into unequal parts (that is, $c_i \leq 1$).

The listing shown in figure 14.0-A can be generated with [FXT: comb/partition-rec-demo.cc]:

```
void
print_part(ulong n)
{
        partition pp(n);
        ulong ct = 0;
        while ( pp.next() < n )
        {
            cout << "    #" << setw(2) << ct << ": ";
            cout << setw(4) << pp.x_ << " == ";
            pp.print2();
            cout << "  ==  ";
            pp.print();
            cout << endl;
            ++ct;
        }
        cout << " " << n << ":  ct=" << ct << endl;
        cout << endl;
}
```

The 190,569,292 partitions of 100 are generated in less than 11 seconds, corresponding to a rate of about 18 million partitions per second.

14.2 Iterative algorithm

An iterative implementation for the special case $V = \{1, 2, 3, \ldots, n\}$ (the integer partitions) is given in [FXT: `class partition` in comb/partition.h]:

```
class partition
{
public:
    ulong *c_;  // partition:  c[1]* 1 + c[2]* 2 + ... + c[n]* n == n
    ulong *s_;  // cumulative sums:  s[j+1] = c[1]* 1 + c[2]* 2 + ... + c[j]* j
    ulong n_;   // partitions of n

public:
    partition(ulong n)
    {
        n_ = n;
        c_ = new ulong[n+1];
        s_ = new ulong[n+1];
        s_[0] = 0;  // unused
        c_[0] = 0;  // unused
        first();
    }

    ~partition()
    {
        delete [] c_;
        delete [] s_;
```

```
    }

    void first()
    {
        c_[1] = n_;
        for (ulong i=2; i<=n_; i++)  { c_[i] = 0; }
        s_[1] = 0;
        for (ulong i=2; i<=n_; i++)  { s_[i] = n_; }
    }

    void last()
    {
        for (ulong i=1; i<n_; i++)  { c_[i] = 0; }
        c_[n_] = 1;
        for (ulong i=1; i<n_; i++)  { s_[i] = 0; }
        // s_[n_+1] = n_;  // unused (and out of bounds)
    }
```

To obtain the next partition, find the smallest index $i \geq 2$ so that $[c_1, c_2, \ldots, c_{i-1}, c_i]$ can be replaced by $[z, 0, 0, \ldots, 0, c_i + 1]$ where $z \geq 0$. The index i (and z) is determined using cumulative sums. The partitions are generated in the same order as shown in figure 14.0-A. The algorithm was given (2006) by Torsten Finke [priv.comm.].

```
    bool next()
    {
        if ( c_[n_]!=0 )  return false;  // last == 1* n (c[n]==1)

        // Find first coefficient c[i], i>=2 that can be increased:
        ulong i = 2;
        while ( s_[i]<i )  ++i;

        ++c_[i];
        s_[i] -= i;
        ulong z = s_[i];
        // Now set c[1], c[2], ..., c[i-1] to the first partition
        // of z into i-1 parts, i.e. set to  z, 0, 0, ..., 0:
        while ( --i > 1 )
        {
            s_[i] = z;
            c_[i] = 0;
        }
        c_[1] = z;  // z* 1 == z
        // s_[1] unused

        return true;
    }
```

The preceding partition can be computed as follows:

```
    bool prev()
    {
        if ( c_[1]==n_ )  return false;  // first == n* 1 (c[1]==n)

        // Find first nonzero coefficient c[i] where i>=2:
        ulong i = 2;
        while ( c_[i]==0 )  ++i;

        --c_[i];
        s_[i] += i;
        ulong z = s_[i];
        // Now set c[1], c[2], ..., c[i-1] to the last partition
        // of z into i-1 parts:
        while ( --i > 1  )
        {
            ulong q = (z>=i ? z/i : 0); // == z/i;
            c_[i] = q;
            s_[i+1] = z;
            z -= q*i;
        }
        c_[1] = z;
        s_[2] = z;
        // s_[1] unused

        return true;
    }

  [--snip--]
};
```

Note that divisions which result in $q = 0$ are avoided, leading to a small speedup. The program [FXT: comb/partition-demo.cc] demonstrates the usage of the class. More than 140 million partitions per second are generated, about 66 million when going backward.

14.3 Partitions into m parts

```
 1:  1 1 1 1 1 1 1 1 1 1 9          12:  1 1 1 1 1 1 1 2 2 3 5
 2:  1 1 1 1 1 1 1 1 1 2 8          13:  1 1 1 1 1 1 1 2 2 4 4
 3:  1 1 1 1 1 1 1 1 1 3 7          14:  1 1 1 1 1 1 1 2 3 3 4
 4:  1 1 1 1 1 1 1 1 1 4 6          15:  1 1 1 1 1 1 1 3 3 3 3
 5:  1 1 1 1 1 1 1 1 1 5 5          16:  1 1 1 1 1 1 2 2 2 2 5
 6:  1 1 1 1 1 1 1 1 2 2 7          17:  1 1 1 1 1 1 2 2 2 3 4
 7:  1 1 1 1 1 1 1 1 2 3 6          18:  1 1 1 1 1 1 2 2 3 3 3
 8:  1 1 1 1 1 1 1 1 2 4 5          19:  1 1 1 1 1 2 2 2 2 2 4
 9:  1 1 1 1 1 1 1 1 3 3 5          20:  1 1 1 1 1 2 2 2 2 3 3
10:  1 1 1 1 1 1 1 1 3 4 4          21:  1 1 1 1 2 2 2 2 2 2 3
11:  1 1 1 1 1 1 1 2 2 2 6          22:  1 1 1 2 2 2 2 2 2 2 2
```

Figure 14.3-A: The 22 partitions of 19 into 11 parts in lexicographic order.

An algorithm for the generation of all partitions of n into m parts is given in [98, p.106] (method ascribed to Hindenburg):

> The initial partition contains $m-1$ units and the element $n-m+1$. To obtain a new partition from a given one, pass over the elements of the latter from right to left, stopping at the first element f which is less, by at least two units, than the final element [...]. Without altering any element at the left of f, write $f+1$ in place of f and every element to the right of f with the exception of the final element, in whose place is written the number which when added to all the other new elements gives the sum n. The process to obtain partitions stops when we reach one in which no part is less than the final part by at least two units.

Figure 14.3-A shows the partitions of 19 into 11 parts. The data was generated with the program [FXT: comb/mpartition-demo.cc].

An efficient implementation is given as [FXT: `class mpartition` in comb/mpartition.h]:

```
class mpartition
// Integer partitions of n into m parts
{
public:
    ulong *x_;  // partition: x[1]+x[2]+...+x[m] = n
    ulong *s_;  // aux: cumulative sums of x[]  (s[0]=0)
    ulong n_;   // integer partitions of n  (must have n>0)
    ulong m_;   // ... into m parts  (must have 0<m<=n)
public:
    mpartition(ulong n, ulong m)
        : n_(n), m_(m)
    {
        x_ = new ulong [m_+1];
        s_ = new ulong [m_+1];
        init();
    }

    ~mpartition()
    {
        delete [] x_;
        delete [] s_;
    }

    const ulong *data() const  { return x_+1; }

    void init()
    {
        x_[0] = 0;
        for (ulong k=1; k<m_; ++k)  x_[k] = 1;
        x_[m_] = n_ - m_ + 1;
        ulong s = 0;
```

```
        for (ulong k=0; k<=m_; ++k)  { s+=x_[k]; s_[k]=s; }
    }

    bool next()
    {
        ulong u = x_[m_];  // last element
        ulong k = m_;
        while ( --k )  { if ( x_[k]+2<=u ) break; }

        if ( k==0 )  return  false;

        ulong f = x_[k] + 1;
        ulong s = s_[k-1];
        while ( k < m_ )
        {
            x_[k] = f;
            s += f;
            s_[k] = s;
            ++k;
        }
        x_[m_] = n_ - s_[m_-1];
        // s_[m_] = n_;  // unchanged

        return  true;
    }
};
```

The auxiliary array of cumulative sums allows the recalculation of the final element without rescanning more than the elements just changed. About 105 million partitions per second can be generated.

A (complicated) construction for a Gray code for integer partitions is given in [198].

14.4 The number of integer partitions

n:	P_n	n:	P_n	n:	P_n	n:	P_n	n:	P_n
1:	1	11:	56	21:	792	31:	6842	41:	44583
2:	2	12:	77	22:	1002	32:	8349	42:	53174
3:	3	13:	101	23:	1255	33:	10143	43:	63261
4:	5	14:	135	24:	1575	34:	12310	44:	75175
5:	7	15:	176	25:	1958	35:	14883	45:	89134
6:	11	16:	231	26:	2436	36:	17977	46:	105558
7:	15	17:	297	27:	3010	37:	21637	47:	124754
8:	22	18:	385	28:	3718	38:	26015	48:	147273
9:	30	19:	490	29:	4565	39:	31185	49:	173525
10:	42	20:	627	30:	5604	40:	37338	50:	204226

Figure 14.4-A: The number of integer partitions of n for $n \leq 50$.

The total number of integer partitions of n is sequence A000041 of [214], the values for $1 \leq x \leq 50$ are shown in figure 14.4-A. If we denote the number of partitions of n into exactly m parts by $P(n,m)$ then

$$P(n,m) \quad = \quad P(n-1,m-1) + P(n-m,m) \tag{14.4-1}$$

were we set $P(0,0) = 1$. We obviously have $P_n = \sum_{m=1}^{n} P(n,m)$. Figure 14.4-B shows $P(n,m)$ for $n \leq 16$, it was created with the program [FXT: comb/num-partitions-demo.cc]. We note that the number of partitions into m parts equals the number of partitions with maximal part equal to m. This can easily be seen by drawing a diagram and its transposed as follows (for the partition $5+2+2+1$ of 10):

```
   43111      5221
5  XXXXX   4 XXXX
2  XX      3 XXX
2  XX      1 X
1  X       1 X
           1 X
```

Thereby any partition with maximal part m (here 5) corresponds to a partition into exactly m parts.

n:	P(n)	P(n,m) for m = 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1:	1	1															
2:	2	1	1														
3:	3	1	1	1													
4:	5	1	2	1	1												
5:	7	1	2	2	1	1											
6:	11	1	3	3	2	1	1										
7:	15	1	3	4	3	2	1	1									
8:	22	1	4	5	5	3	2	1	1								
9:	30	1	4	7	6	5	3	2	1	1							
10:	42	1	5	8	9	7	5	3	2	1	1						
11:	56	1	5	10	11	10	7	5	3	2	1	1					
12:	77	1	6	12	15	13	11	7	5	3	2	1	1				
13:	101	1	6	14	18	18	14	11	7	5	3	2	1	1			
14:	135	1	7	16	23	23	20	15	11	7	5	3	2	1	1		
15:	176	1	7	19	27	30	26	21	15	11	7	5	3	2	1	1	
16:	231	1	8	21	34	37	35	28	22	15	11	7	5	3	2	1	1

Figure 14.4-B: Numbers $P(n,m)$ of partitions of n into m parts.

The generating function for the partitions into exactly m parts is

$$\sum_{n=1}^{\infty} P(n,m)\, x^n \quad = \quad \frac{x^m}{\prod_{k=1}^{m}(1-x^k)} \tag{14.4-2}$$

For example, the row for $m=3$ in figure 14.4-B corresponds to the power series

```
? m=3; (x^m/prod(k=1,m,1-x^k)+O(x^17))
  x^3 + x^4 + 2*x^5 + 3*x^6 + 4*x^7 + 5*x^8 + 7*x^9 + 8*x^10 + \
  10*x^11 + 12*x^12 + 14*x^13 + 16*x^14 + 19*x^15 + 21*x^16 + O(x^17)
```

The generating function for the number P_n of integer partitions of n can be given as [109, p.357]

$$\sum_{n=0}^{\infty} P_n\, x^n \quad = \quad \frac{1}{\prod_{n=1}^{\infty}(1-x^n)} \quad =: \quad \frac{1}{\eta(x)} \tag{14.4-3a}$$

Then we have

$$\eta(x) \quad = \quad 1+\sum_{n=1}^{\infty}(-1)^n\left(x^{n(3n-1)/2}+x^{n(3n+1)/2}\right) \tag{14.4-3b}$$

and

$$\frac{1}{\eta(x)} \quad = \quad 1+\sum_{n=1}^{\infty}\frac{x^n}{\prod_{k=1}^{n}(1-x^k)} \tag{14.4-3c}$$

$$= \quad 1+\sum_{n=1}^{\infty}\frac{x^{n^2}}{\left[\prod_{k=1}^{n}(1-x^k)\right]^2} \tag{14.4-3d}$$

```
? N=10;x=t+O(t^N);
? 1/prod(k=1,N,1-x^k)
  1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + 15*t^7 + 22*t^8 + 30*t^9 + O(t^10)
\\ == 1+sum(n=1,N,x^n/prod(j=1,n,1-x^j))
\\ == 1+sum(n=1,N,x^(n^2)/(prod(k=1,n,(1-x^k))^2))
\\ == 1/(1+sum(n=1,N,(-1)^n*(x^(n*(3*n-1)/2)+x^(n*(3*n+1)/2))))
```

Relation 14.4-3c is the special case $a_n = x^n$ (and $N \to \infty$) of [180, p.83]

$$\frac{1}{\prod_{n=1}^{N}(1-a_n)} \quad = \quad 1+\sum_{n=1}^{N}\frac{a_n}{\prod_{k=1}^{n}(1-a_k)} \tag{14.4-4}$$

For the number D_n of partitions of n into distinct parts we have the generating function

$$\eta_+(x) \quad := \quad \prod_{n=1}^{\infty}(1+x^n) \;=\; \sum_{n=0}^{\infty} D_n\, x^n \tag{14.4-5}$$

We have (the number of partitions into distinct parts equals the number of partitions into odd parts)

$$\eta_+(x) \quad = \quad \frac{\eta(x^2)}{\eta(x)} \quad = \quad \frac{1}{\prod_{k=1}^{\infty}(1-x^{2\,k-1})} \tag{14.4-6}$$

Thereby relation 14.4-3b allows fast computation of $\eta_+(x)$. The sequence of coefficients D_n is entry A000009 of [214]:

```
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18, 22, 27, 32, 38, 46, 54,
 64, 76, 89, 104, 122, 142, 165, 192, 222, 256, ...]
```

The number of partitions where at most r elements of each partition are equal has the generating function

$$\prod_{n=1}^{\infty}\left(1+x+x^2+\ldots+x^r\right) \quad = \quad \frac{\eta(x^{r+1})}{\eta(x)} \tag{14.4-7a}$$

$$\quad = \quad \frac{1}{\prod_{k\neq 0 \bmod r+1}(1-x^k)} \tag{14.4-7b}$$

The second relation tells us that the number of such partitions equals the number of partitions into parts not divisible by $r+1$.

Chapter 15

Set partitions

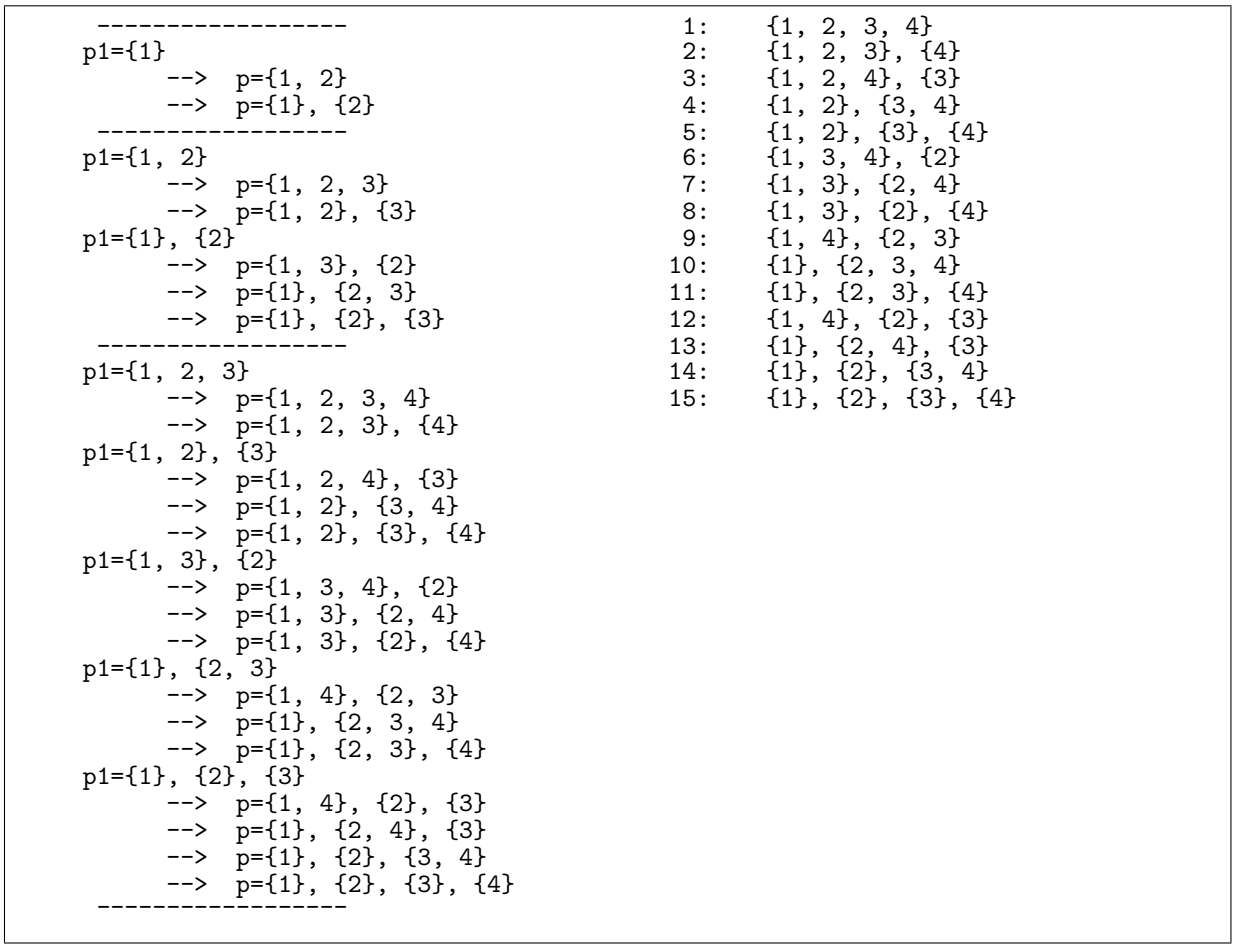


Figure 15.0-A: Recursive construction of the set partitions of the 4-element set $S_4 = \{1, 2, 3, 4\}$ (left). The resulting list of all set partitions of is shown on the right.

For a set of n elements, say $S_n := \{1, 2, \ldots, n\}$, a *set partition* is a set $P = \{s_1, s_2, \ldots, s_k\}$ of non-empty subsets s_i of S_n whose intersection is empty and whose union equals S.

For example, there are 5 set partitions of the set $S_3 = \{1, 2, 3\}$:

```
1:  { {1, 2, 3} }
2:  { {1, 2}, {3} }
3:  { {1, 3}, {2} }
4:  { {1}, {2, 3} }
5:  { {1}, {2}, {3} }
```

The following sets are not set partitions of S_3:

```
{ {1, 2, 3}, {1} }   // intersection not empty
{ {1}, {3} }         // union does not contain 2
```

As the order of elements in a set does not matter we sort them in ascending order. For a set of sets we order the sets in ascending order of the first elements.

Recursive generation

We write Z_n for the list of all set partitions of the n-element set S_n. In order to generate Z_n we observe that with a complete list Z_{n-1} of partitions of the set S_{n-1} we can generate the elements of Z_n in the following way: For each element (set partition) $P \in Z_{n-1}$, create set partitions of S_n by appending the element n to the first, second, ..., last subset and one more by appending the set n as the last subset.

For example, the partition $\{\{1, 2\}, \{3, 4\}\} \in Z_4$ leads to 3 partitions of S_5:

```
P = { {1, 2}, {3, 4} }
      -->  { {1, 2, 5}, {3, 4} }
      -->  { {1, 2}, {3, 4, 5} }
      -->  { {1, 2}, {3, 4}, {5} }
```

Now we start with the only partition of the one-element set, $\{\{1\}\}$, and apply the described step $n - 1$ times. The construction is shown in the left column of figure 15.0-A, the right column shows all set partitions for $n = 5$.

15.1 The number of set partitions: Stirling set numbers and Bell numbers

```
 n   b=   B(n)    k: 1     2     3     4     5     6     7     8     9    10
 1:  b=      1       1
 2:  b=      2       1     1
 3:  b=      5       1     3     1
 4:  b=     15       1     7     6     1
 5:  b=     52       1    15    25    10     1
 6:  b=    203       1    31    90    65    15     1
 7:  b=    877       1    63   301   350   140    21     1
 8:  b=   4140       1   127   966  1701  1050   266    28     1
 9:  b=  21147       1   255  3025  7770  6951  2646   462    36     1
10:  b= 115975       1   511  9330 34105 42525 22827  5880   750    45     1
```

Figure 15.1-A: Stirling numbers of the second kind and Bell numbers.

The sequence of numbers of set partitions of S_n for $n \geq 1$ is 1, 2, 5, 15, 52, 203, 877, ... These are the *Bell numbers*, sequence A000110 of [214]. The Bell numbers can be computed using the observation that in every step of the computation a partition of S_{x-1} into k subsets leads to $k + 1$ partitions of S_x. Of these partitions k contain k subsets and one contains $k + 1$ subsets.

If we write the number of partitions of S_n into k subsets as a triangular array, then the entry can be computed as the sum of its upper left neighbor plus k times its upper neighbor. We start with a single one in the row for $n = 1$. The scheme is shown in figure 15.1-A. The numbers are the *Stirling numbers of the second kind* (or *Stirling set numbers*), see sequence A008277 of [214].

The sum over all elements of row n gives the n-th Bell number. Another way of computing the Bell numbers is given in section 3.8.2 on page 131. The array shown can be generated with the program [FXT: comb/stirling2-demo.cc].

We further have the recursion

$$B_{n+1} \;=\; \sum_{k=0}^{n} \binom{n}{k} B_k \tag{15.1-1}$$

As pari/gp code:

```
? N=11; v=vector(N); v[1]=1;
? for (j=2, N, v[j]=sum(k=1, j-1, binomial(j-2,k-1)*v[k])); v
  [1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
```

The ordinary generating function for the Bell numbers can be given as

$$\sum_{n=0}^{\infty} B_n\, x^n \;=\; \sum_{k=0}^{\infty} \frac{x^k}{\prod_{j=1}^{k} (1 - j\, x)} \;=\; 1 + x + 2\,x^2 + 5\,x^3 + 15\,x^4 + 52\,x^5 + \ldots \tag{15.1-2}$$

The exponential generating function is

$$\exp(\exp(x) - 1) \;=\; \sum_{n=0}^{\infty} B_n\, \frac{x^n}{n!} \tag{15.1-3}$$

```
? sum(k=0,11,x^k/prod(j=1,k,1-j*x))+O(x^8)  \\ OGF
  1 + x + 2*x^2 + 5*x^3 + 15*x^4 + 52*x^5 + 203*x^6 + 877*x^7 + O(x^8)
? serlaplace(exp(exp(x)-1))                 \\ EGF
  1 + x + 2*x^2 + 5*x^3 + 15*x^4 + 52*x^5 + 203*x^6 + 877*x^7 + 4140*x^8 + ...
```

Dobinski's formula for the Bell numbers is

$$B_n \;=\; \frac{1}{e} \sum_{k=1}^{\infty} \frac{n^k}{k!} \tag{15.1-4}$$

15.2 Generation in minimal-change order

A modified version of the recursive construction generates the set partitions in a minimal-change order. We can generate the 'incremented' partitions in two orders, forward (left to right)

```
P = { {1, 2}, {3, 4} }
      -->  { {1, 2, 5}, {3, 4} }
      -->  { {1, 2}, {3, 4, 5} }
      -->  { {1, 2}, {3, 4}, {5} }
```

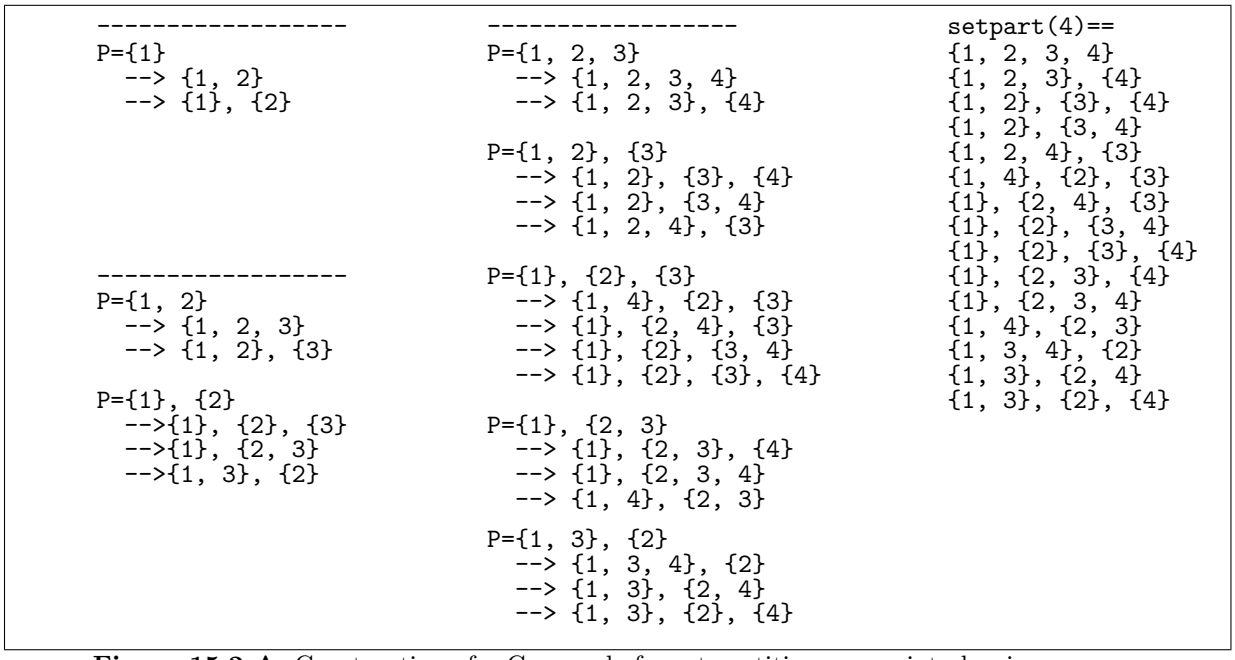
or backward (right to left)

```
P = { {1, 2}, {3, 4} }
      -->  { {1, 2}, {3, 4}, {5} }
      -->  { {1, 2}, {3, 4, 5} }
      -->  { {1, 2, 5}, {3, 4} }
```

The resulting process of interleaving elements is shown in figure 15.2-A. The method is similar to Trotter's construction for permutations, see figure 10.7-B on page 240. If we change the direction with every subset that is to be incremented, we obtain the minimal-change order shown in figure 15.2-B for $n = 4$. The left column is obtained by starting with the forward direction in each step of the recursion, the right by starting with the backward direction. The lists can be computed with [FXT: comb/setpartition-demo.cc].

The C++ class [FXT: `class set_partition` in comb/setpartition.h] stores the list in an array of signed characters. The stored value is negated if the element is the last in the subset. The work involved with the creation of Z_n is proportional to $\sum_{k=1}^{n} k\, B_k$ where B_k is the k-th Bell number.

The parameter `xdr` of the constructor determines the order in which the partitions are being created:



```
------------------
P={1}
  --> {1, 2}
  --> {1}, {2}

------------------
P={1, 2}
  --> {1, 2, 3}
  --> {1, 2}, {3}

P={1}, {2}
  -->{1}, {2}, {3}
  -->{1}, {2, 3}
  -->{1, 3}, {2}
```

```
------------------
P={1, 2, 3}
  --> {1, 2, 3, 4}
  --> {1, 2, 3}, {4}

P={1, 2}, {3}
  --> {1, 2}, {3}, {4}
  --> {1, 2}, {3, 4}
  --> {1, 2, 4}, {3}

P={1}, {2}, {3}
  --> {1, 4}, {2}, {3}
  --> {1}, {2, 4}, {3}
  --> {1}, {2}, {3, 4}
  --> {1}, {2}, {3}, {4}

P={1}, {2, 3}
  --> {1}, {2, 3}, {4}
  --> {1}, {2, 3, 4}
  --> {1, 4}, {2, 3}

P={1, 3}, {2}
  --> {1, 3, 4}, {2}
  --> {1, 3}, {2, 4}
  --> {1, 3}, {2}, {4}
```

```
setpart(4)==
{1, 2, 3, 4}
{1, 2, 3}, {4}
{1, 2}, {3}, {4}
{1, 2}, {3, 4}
{1, 2, 4}, {3}
{1, 4}, {2}, {3}
{1}, {2, 4}, {3}
{1}, {2}, {3, 4}
{1}, {2}, {3}, {4}
{1}, {2, 3}, {4}
{1}, {2, 3, 4}
{1, 4}, {2, 3}
{1, 3, 4}, {2}
{1, 3}, {2, 4}
{1, 3}, {2}, {4}
```

Figure 15.2-A: Construction of a Gray code for set partitions as an interleaving process.

```
 1:  {1, 2, 3, 4}
 2:  {1, 2, 3}, {4}
 3:  {1, 2}, {3}, {4}
 4:  {1, 2}, {3, 4}
 5:  {1, 2, 4}, {3}
 6:  {1, 4}, {2}, {3}
 7:  {1}, {2, 4}, {3}
 8:  {1}, {2}, {3, 4}
 9:  {1}, {2}, {3}, {4}
10:  {1}, {2, 3}, {4}
11:  {1}, {2, 3, 4}
12:  {1, 4}, {2, 3}
13:  {1, 3, 4}, {2}
14:  {1, 3}, {2, 4}
15:  {1, 3}, {2}, {4}
```

```
 1:  {1}, {2}, {3}, {4}
 2:  {1}, {2}, {3, 4}
 3:  {1}, {2, 4}, {3}
 4:  {1, 4}, {2}, {3}
 5:  {1, 4}, {2, 3}
 6:  {1}, {2, 3, 4}
 7:  {1}, {2, 3}, {4}
 8:  {1, 3}, {2}, {4}
 9:  {1, 3}, {2, 4}
10:  {1, 3, 4}, {2}
11:  {1, 2, 3, 4}
12:  {1, 2, 3}, {4}
13:  {1, 2}, {3}, {4}
14:  {1, 2}, {3, 4}
15:  {1, 2, 4}, {3}
```

Figure 15.2-B: Set partitions of $S_4 = \{1, 2, 3, 4\}$ in two different minimal-change orders.

```
class set_partition
// Set partitions of the set {1,2,3,...,n}
// By default in minimal-change order
{
public:
    ulong n_;   // Number of elements of set (set = {1,2,3,...,n})
    int *p_;    // p[] contains set partitions of length 1,2,3,...,n
    int **pp_;  // pp[k] points to start of set partition k
    int *ns_;   // ns[k] Number of Sets in set partition k
    int *as_;   // element k attached At Set (0<=as[k]<=k) of set(k-1)
    int *d_;    // direction with recursion (+1 or -1)
    int *x_;    // current set partition (==pp[n])
    bool xdr_;  // whether to change direction in recursion (==> minimal-change order)
    int dr0_;   // dr0: starting direction in each recursive step:
    //   dr0=+1  ==> start with partition  {{1,2,3,...,n}}
    //   dr0=-1  ==> start with partition  {{1},{2},{3},...,{n}}}
public:
    set_partition(ulong n, bool xdr=true, int dr0=+1)
        : n_(n)
    {
        ulong np = (n_*(n_+1))/2;  // == \sum_{k=1}^{n}{k}
        p_ = new int[np];

        pp_ = new int *[n_+1];
        pp_[0] = 0;  // unused
        pp_[1] = p_;
        for (ulong k=2; k<=n_; ++k)  pp_[k] = pp_[k-1] + (k-1);

        ns_ = new int[n_+1];
        as_ = new int[n_+1];
        d_ = new int[n_+1];
        x_ = pp_[n_];

        init(xdr, dr0);
    }
   [--snip--]  // destructor

    bool next()  { return next_rec(n_); }

    const int* data()  const  { return x_; }

    ulong print()  const
    // Print current set partition
    // Return number of chars printed
    { return print_p(n_); }

    ulong print_p(ulong k)  const;
    void print_internal()  const;  // print internal state
protected:
   [--snip--]  // internal methods
};
```

The actual work is done by the methods `next_rec()` and `cp_append()` [FXT: comb/setpartition.cc]:

```
int
set_partition::cp_append(const int *src, int *dst, ulong k, ulong a)
// Copy partition in src[0,...,k-2] to dst[0,...,k-1]
// append element k at subset a (a>=0)
// Return number of sets in created partition.
{
    ulong ct = 0;
    for (ulong j=0; j<k-1; ++j)
    {
        int e = src[j];
        if ( e > 0 )  dst[j] = e;
        else
        {
            if ( a==ct )  { dst[j]=-e; ++dst; dst[j]=-k; }
            else  dst[j] = e;
            ++ct;
        }
    }
    if ( a>=ct )  { dst[k-1] = -k; ++ct; }

    return ct;
}
```

```
int
set_partition::next_rec(ulong k)
// Update partition in level k from partition in level k-1  (k<=n)
// Return number of sets in created partition
{
    if ( k<=1 )  return 0;  // current is last

    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>ns_[k-1]) : (as<0) );
    if ( ovq )  // have to recurse
    {
        ulong ns1 = next_rec(k-1);
        if ( 0==ns1 )  return 0;

        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;

        as = ( (d>0) ? 0 : ns_[k-1] );
    }
    as_[k] = as;

    ulong ns = cp_append(pp_[k-1], pp_[k], k, as);
    ns_[k] = ns;
    return ns;
}
```

```
       1:  as[ 0 0 0 0 ]    x[ +1 +2 +3 -4 ]    {1, 2, 3, 4}
       2:  as[ 0 0 0 1 ]    x[ +1 +2 -3 -4 ]    {1, 2, 3}, {4}
       3:  as[ 0 0 1 0 ]    x[ +1 +2 -4 -3 ]    {1, 2, 4}, {3}
       4:  as[ 0 0 1 1 ]    x[ +1 -2 +3 -4 ]    {1, 2}, {3, 4}
       5:  as[ 0 0 1 2 ]    x[ +1 -2 -3 -4 ]    {1, 2}, {3}, {4}
       6:  as[ 0 1 0 0 ]    x[ +1 +3 -4 -2 ]    {1, 3, 4}, {2}
       7:  as[ 0 1 0 1 ]    x[ +1 -3 +2 -4 ]    {1, 3}, {2, 4}
       8:  as[ 0 1 0 2 ]    x[ +1 -3 -2 -4 ]    {1, 3}, {2}, {4}
       9:  as[ 0 1 1 0 ]    x[ +1 -4 +2 -3 ]    {1, 4}, {2, 3}
      10:  as[ 0 1 1 1 ]    x[ -1 +2 +3 -4 ]    {1}, {2, 3, 4}
      11:  as[ 0 1 1 2 ]    x[ -1 +2 -3 -4 ]    {1}, {2, 3}, {4}
      12:  as[ 0 1 2 0 ]    x[ +1 -4 -2 -3 ]    {1, 4}, {2}, {3}
      13:  as[ 0 1 2 1 ]    x[ -1 +2 -4 -3 ]    {1}, {2, 4}, {3}
      14:  as[ 0 1 2 2 ]    x[ -1 -2 +3 -4 ]    {1}, {2}, {3, 4}
      15:  as[ 0 1 2 3 ]    x[ -1 -2 -3 -4 ]    {1}, {2}, {3}, {4}
```

Figure 15.2-C: The partitions of the set $S_4 = \{1, 2, 3, 4\}$ together with the internal representations: the 'signed value' array `x[]` and the 'attachment' array `as[]`.

The partitions are stored as an array of signed integers that are greater than zero and smaller or equal to n. A negative value indicates that it is the last of the subset. For example, the set partitions of S_4 together with their 'signed value' representations are shown in figure 15.2-C. The array `as[]` contains a restricted growth string (RGS) with the condition $a_j \leq 1 + \max_{i<j}(a_i)$. A different sort of RGS is described in section 13.2 on page 301.

The copying is the performance bottleneck of the algorithm. Therefore 'only' about 11 million partitions are generated per second. An $O(1)$ algorithm for the Gray code starting with all elements in one set is given in [148].

For some applications the restricted growth strings (RGS) may suffice. We give an implementation for their generation and algorithms to generate two classes of generalized RGS that contain the RGS for set partitions as a special case.

15.2.1 RGS for set partitions in minimal-change order

The C++ implementation [FXT: `class set_partition_rgs` in comb/setpartition-rgs.h] generates the RGS for set partitions in lexicographic or minimal-change order. The RGS are updated by the recursive routine [FXT: comb/setpartition-rgs.cc]:

```
int
```

```
set_partition_rgs::next_rec(ulong k)
{
    if ( k<=1 )  return 0;  // current is last

    int d = d_[k];
    int as = as_[k] + d;
    bool ovq = ( (d>0) ? (as>ns_[k-1]) : (as<0) );
    if ( ovq )  // have to recurse
    {
        if ( 0==next_rec(k-1) )  return 0;

        d = ( xdr_ ? -d : dr0_ );
        d_[k] = d;

        as = ( (d>0) ? 0 : ns_[k-1] );
    }
    as_[k] = as;

    ulong ns = ns_[k] = max2(ns_[k-1], as_[k]+1);
    return ns;
}
```

An iterative version of the update routine is

```
    bool next()
    {
        ulong k = n_;
        while ( (ulong)(as_[k] + d_[k]) > (ulong)ns_[k-1] ) // <0 or >max
        {
            if ( --k <= 1 )  return 0;
        }

        as_[k] += d_[k];
        ns_[k] = max2(ns_[k-1], as_[k]+1);

        while ( ++k<=n_ )
        {
            ulong d = d_[k];
            d = ( xdr_ ? -d : dr0_ );
            d_[k] = d;
            ulong as = ( ((int)d>0) ? 0 : ns_[k-1] );
            as_[k] = as;
            ns_[k] = max2(ns_[k-1], as_[k]+1);
        }

        return 1;
    }
```

It is activated (by default) via the define

```
#ifdef SETPART_RGS_ITERATIVE
```

A program that shows the usage of the class is [FXT: comb/setpartition-rgs-demo.cc]. Note that while the RGS correspond to a Gray code for set partitions the RGS can change in more than one position, see the left column of figure 15.2-C. About 78 million RGS per second are generated with the recursive update routine, with the iterative update the rate is about 140 million per second.

15.2.2 Max-increment RGS *

The generation of RGSs $s = [s_0, s_1, \ldots, s_{n-1}]$ where $s_k \le i + \max_{j<k}(s_j)$ is a generalization of the RGSs for set partitions (where $i = 1$). Figure 15.2-D show RGSs in lexicographic order for $i = 2$ (left) and $i = 1$ (right). The strings can be generated in lexicographic order using [FXT: `class rgs_maxincr` in comb/rgs-maxincr.h]:

```
class rgs_maxincr
{
public:
    ulong *s_;  // restricted growth string
    ulong *m_;  // m_[k-1] == max possible value for s_[k]
    ulong n_;   // Length of strings
    ulong i_;   // s[k] <= max_{j<k}(s[j]+i)
    // i==1 ==> RGS for set partitions
public:
    rgs_maxincr(ulong n, ulong i=1)
```

```
        RGS(4,2)        max(4,2)
 1:  [ . . . . ]    [ . . . . ]
 2:  [ . . . 1 ]    [ . . . 1 ]
 3:  [ . . . 2 ]    [ . . . 2 ]
 4:  [ . . 1 . ]    [ . . 1 1 ]
 5:  [ . . 1 1 ]    [ . . 1 1 ]
 6:  [ . . 1 2 ]    [ . . 1 2 ]
 7:  [ . . 1 3 ]    [ . . 1 3 ]
 8:  [ . . 2 . ]    [ . . 2 2 ]
 9:  [ . . 2 1 ]    [ . . 2 2 ]
10:  [ . . 2 2 ]    [ . . 2 2 ]
11:  [ . . 2 3 ]    [ . . 2 3 ]
12:  [ . . 2 4 ]    [ . . 2 4 ]
13:  [ . 1 . . ]    [ . 1 1 1 ]
14:  [ . 1 . 1 ]    [ . 1 1 1 ]
15:  [ . 1 . 2 ]    [ . 1 1 2 ]
16:  [ . 1 . 3 ]    [ . 1 1 3 ]
17:  [ . 1 1 . ]    [ . 1 1 1 ]
18:  [ . 1 1 1 ]    [ . 1 1 1 ]
19:  [ . 1 1 2 ]    [ . 1 1 2 ]
20:  [ . 1 1 3 ]    [ . 1 1 3 ]
21:  [ . 1 2 . ]    [ . 1 2 2 ]
22:  [ . 1 2 1 ]    [ . 1 2 2 ]
23:  [ . 1 2 2 ]    [ . 1 2 2 ]
24:  [ . 1 2 3 ]    [ . 1 2 3 ]
25:  [ . 1 2 4 ]    [ . 1 2 4 ]
26:  [ . 1 3 . ]    [ . 1 3 3 ]
27:  [ . 1 3 1 ]    [ . 1 3 3 ]
28:  [ . 1 3 2 ]    [ . 1 3 3 ]
29:  [ . 1 3 3 ]    [ . 1 3 3 ]
30:  [ . 1 3 4 ]    [ . 1 3 4 ]
31:  [ . 1 3 5 ]    [ . 1 3 5 ]
32:  [ . 2 . . ]    [ . 2 2 2 ]
33:  [ . 2 . 1 ]    [ . 2 2 2 ]
34:  [ . 2 . 2 ]    [ . 2 2 2 ]
35:  [ . 2 . 3 ]    [ . 2 2 3 ]
36:  [ . 2 . 4 ]    [ . 2 2 4 ]
37:  [ . 2 1 . ]    [ . 2 2 2 ]
38:  [ . 2 1 1 ]    [ . 2 2 2 ]
39:  [ . 2 1 2 ]    [ . 2 2 2 ]
40:  [ . 2 1 3 ]    [ . 2 2 3 ]
41:  [ . 2 1 4 ]    [ . 2 2 4 ]
42:  [ . 2 2 . ]    [ . 2 2 2 ]
43:  [ . 2 2 1 ]    [ . 2 2 2 ]
44:  [ . 2 2 2 ]    [ . 2 2 2 ]
45:  [ . 2 2 3 ]    [ . 2 2 3 ]
46:  [ . 2 2 4 ]    [ . 2 2 4 ]
47:  [ . 2 3 . ]    [ . 2 3 3 ]
48:  [ . 2 3 1 ]    [ . 2 3 3 ]
49:  [ . 2 3 2 ]    [ . 2 3 3 ]
50:  [ . 2 3 3 ]    [ . 2 3 3 ]
51:  [ . 2 3 4 ]    [ . 2 3 4 ]
52:  [ . 2 3 5 ]    [ . 2 3 5 ]
53:  [ . 2 4 . ]    [ . 2 4 4 ]
54:  [ . 2 4 1 ]    [ . 2 4 4 ]
55:  [ . 2 4 2 ]    [ . 2 4 4 ]
56:  [ . 2 4 3 ]    [ . 2 4 4 ]
57:  [ . 2 4 4 ]    [ . 2 4 4 ]
58:  [ . 2 4 5 ]    [ . 2 4 5 ]
59:  [ . 2 4 6 ]    [ . 2 4 6 ]

        RGS(5,1)        max(5,1)
 1:  [ . . . . . ]    [ . . . . . ]
 2:  [ . . . . 1 ]    [ . . . . 1 ]
 3:  [ . . . 1 . ]    [ . . . 1 1 ]
 4:  [ . . . 1 1 ]    [ . . . 1 1 ]
 5:  [ . . . 1 2 ]    [ . . . 1 2 ]
 6:  [ . . 1 . . ]    [ . . 1 1 1 ]
 7:  [ . . 1 . 1 ]    [ . . 1 1 1 ]
 8:  [ . . 1 . 2 ]    [ . . 1 1 2 ]
 9:  [ . . 1 1 . ]    [ . . 1 1 1 ]
10:  [ . . 1 1 1 ]    [ . . 1 1 1 ]
11:  [ . . 1 1 2 ]    [ . . 1 1 2 ]
12:  [ . . 1 2 . ]    [ . . 1 2 2 ]
13:  [ . . 1 2 1 ]    [ . . 1 2 2 ]
14:  [ . . 1 2 2 ]    [ . . 1 2 2 ]
15:  [ . . 1 2 3 ]    [ . . 1 2 3 ]
16:  [ . 1 . . . ]    [ . 1 1 1 1 ]
17:  [ . 1 . . 1 ]    [ . 1 1 1 1 ]
18:  [ . 1 . . 2 ]    [ . 1 1 1 2 ]
19:  [ . 1 . 1 . ]    [ . 1 1 1 1 ]
20:  [ . 1 . 1 1 ]    [ . 1 1 1 1 ]
21:  [ . 1 . 1 2 ]    [ . 1 1 1 2 ]
22:  [ . 1 . 2 . ]    [ . 1 1 2 2 ]
23:  [ . 1 . 2 1 ]    [ . 1 1 2 2 ]
24:  [ . 1 . 2 2 ]    [ . 1 1 2 2 ]
25:  [ . 1 . 2 3 ]    [ . 1 1 2 3 ]
26:  [ . 1 1 . . ]    [ . 1 1 1 1 ]
27:  [ . 1 1 . 1 ]    [ . 1 1 1 1 ]
28:  [ . 1 1 . 2 ]    [ . 1 1 1 2 ]
29:  [ . 1 1 1 . ]    [ . 1 1 1 1 ]
30:  [ . 1 1 1 1 ]    [ . 1 1 1 1 ]
31:  [ . 1 1 1 2 ]    [ . 1 1 1 2 ]
32:  [ . 1 1 2 . ]    [ . 1 1 2 2 ]
33:  [ . 1 1 2 1 ]    [ . 1 1 2 2 ]
34:  [ . 1 1 2 2 ]    [ . 1 1 2 2 ]
35:  [ . 1 1 2 3 ]    [ . 1 1 2 3 ]
36:  [ . 1 2 . . ]    [ . 1 2 2 2 ]
37:  [ . 1 2 . 1 ]    [ . 1 2 2 2 ]
38:  [ . 1 2 . 2 ]    [ . 1 2 2 2 ]
39:  [ . 1 2 . 3 ]    [ . 1 2 2 3 ]
40:  [ . 1 2 1 . ]    [ . 1 2 2 2 ]
41:  [ . 1 2 1 1 ]    [ . 1 2 2 2 ]
42:  [ . 1 2 1 2 ]    [ . 1 2 2 2 ]
43:  [ . 1 2 1 3 ]    [ . 1 2 2 3 ]
44:  [ . 1 2 2 . ]    [ . 1 2 2 2 ]
45:  [ . 1 2 2 1 ]    [ . 1 2 2 2 ]
46:  [ . 1 2 2 2 ]    [ . 1 2 2 2 ]
47:  [ . 1 2 2 3 ]    [ . 1 2 2 3 ]
48:  [ . 1 2 3 . ]    [ . 1 2 3 3 ]
49:  [ . 1 2 3 1 ]    [ . 1 2 3 3 ]
50:  [ . 1 2 3 2 ]    [ . 1 2 3 3 ]
51:  [ . 1 2 3 3 ]    [ . 1 2 3 3 ]
52:  [ . 1 2 3 4 ]    [ . 1 2 3 4 ]
```

Figure 15.2-D: Length-4 max-increment RGS with maximal increment 2 and the corresponding array of maxima (left), and length-5 RGSs with maximal increment 1 (right). Dots denote zeros.

```
    {
        n_ = n;
        m_ = new ulong[n_];
        s_ = new ulong[n_];
        i_ = i;
        first();
    }

    ~rgs_maxincr()
    {
        delete [] m_;
        delete [] s_;
    }

    void first()
    {
        ulong n = n_;
        for (ulong k=0; k<n; ++k)  s_[k] = 0;
        for (ulong k=0; k<n; ++k)  m_[k] = i_;
    }
[--snip--]
```

The computation if the successor returns the index of first (leftmost) changed element in the string. Zero is returned if the current string is the last:

```
    ulong next()
    {
        ulong k = n_;
    start:
        --k;
        if ( k==0 )  return 0;

        ulong sk = s_[k] + 1;
        ulong m1 = m_[k-1];
        if ( sk > m1+i_ )  // "carry"
        {
            s_[k] = 0;
            goto start;
        }

        s_[k] = sk;
        if ( sk>m1 )  m1 = sk;
        for (ulong j=k; j<n_; ++j )  m_[j] = m1;

        return k;
    }
[--snip--]
```

About 115 million RGSs per second are generated with the routine. Figure 15.2-D was created with the program [FXT: comb/rgs-maxincr-demo.cc]. The sequence of numbers of Max-increment RGSs with increment i =1, 2, 3, and 4, start

n:	0	1	2	3	4	5	6	7	8	9
i=1:	1	2	5	15	52	203	877	4140	21147	115975
i=2:	1	3	12	59	339	2210	16033	127643	1103372	10269643
i=3:	1	4	22	150	1200	10922	110844	1236326	14990380	195895202
i=4:	1	5	35	305	3125	36479	475295	6811205	106170245	1784531879

The sequence for $i = 2$ is entry A080337 of [214], it has the exponential generating function (EGF)

$$\sum_{n=0}^{\infty} B_{n,2}\,\frac{x^n}{n!} \quad = \quad \exp\left(x + \exp(x) + \frac{\exp(2\,x)}{2} - \frac{3}{2}\right) \tag{15.2-1}$$

The sequence of numbers of increment-3 RGSs has the EGF

$$\sum_{n=0}^{\infty} B_{n,3}\,\frac{x^n}{n!} \quad = \quad \exp\left(x + \exp(x) + \frac{\exp(2\,x)}{2} + \frac{\exp(3\,x)}{3} - \frac{11}{6}\right) \tag{15.2-2}$$

Omitting the empty string, we restate the EGF for the Bell numbers as

$$\sum_{n=0}^{\infty} B_{n,1}\,\frac{x^n}{n!} \quad = \quad \exp\left(x + \exp(x) - 1\right) \;=\; \frac{1}{0!} + \frac{2}{1!}\,x + \frac{5}{2!}\,x^2 + \frac{15}{!3}\,x^3 + \frac{52}{4!}\,x^4 + \ldots \tag{15.2-3}$$

The EGF for the increment-i RGS is

$$\sum_{n=0}^{\infty} B_{n,i} \frac{x^n}{n!} = \exp\left(x + \sum_{j=1}^{i} \frac{\exp(j\,x) - 1}{j}\right) \tag{15.2-4}$$

15.2.3 F-increment RGS *

```
        RGS(4,2)        F(2)
 1:  [ . . . . . ]   [ . . . . . ]
 2:  [ . . . 1 ]     [ . . . . ]
 3:  [ . . . 2 ]     [ . . . 2 ]
 4:  [ . . 1 . ]     [ . . . . ]
 5:  [ . . 1 1 ]     [ . . . . ]
 6:  [ . . 1 2 ]     [ . . . 2 ]
 7:  [ . . 2 . ]     [ . . 2 2 ]
 8:  [ . . 2 1 ]     [ . . 2 2 ]
 9:  [ . . 2 2 ]     [ . . 2 2 ]
10:  [ . . 2 3 ]     [ . . 2 2 ]
11:  [ . . 2 4 ]     [ . . 2 4 ]
12:  [ . 1 . . ]     [ . . . . ]
13:  [ . 1 . 1 ]     [ . . . . ]
14:  [ . 1 . 2 ]     [ . . . 2 ]
15:  [ . 1 1 . ]     [ . . . . ]
16:  [ . 1 1 1 ]     [ . . . . ]
17:  [ . 1 1 2 ]     [ . . . 2 ]
18:  [ . 1 2 . ]     [ . . 2 2 ]
19:  [ . 1 2 1 ]     [ . . 2 2 ]
20:  [ . 1 2 2 ]     [ . . 2 2 ]
21:  [ . 1 2 3 ]     [ . . 2 2 ]
22:  [ . 1 2 4 ]     [ . . 2 4 ]
23:  [ . 2 . . ]     [ . 2 2 2 ]
24:  [ . 2 . 1 ]     [ . 2 2 2 ]
25:  [ . 2 . 2 ]     [ . 2 2 2 ]
26:  [ . 2 . 3 ]     [ . 2 2 2 ]
27:  [ . 2 . 4 ]     [ . 2 2 4 ]
28:  [ . 2 1 . ]     [ . 2 2 2 ]
29:  [ . 2 1 1 ]     [ . 2 2 2 ]
30:  [ . 2 1 2 ]     [ . 2 2 2 ]
31:  [ . 2 1 3 ]     [ . 2 2 2 ]
32:  [ . 2 1 4 ]     [ . 2 2 4 ]
33:  [ . 2 2 . ]     [ . 2 2 2 ]
34:  [ . 2 2 1 ]     [ . 2 2 2 ]
35:  [ . 2 2 2 ]     [ . 2 2 2 ]
36:  [ . 2 2 3 ]     [ . 2 2 2 ]
37:  [ . 2 2 4 ]     [ . 2 2 4 ]
38:  [ . 2 3 . ]     [ . 2 2 2 ]
39:  [ . 2 3 1 ]     [ . 2 2 2 ]
40:  [ . 2 3 2 ]     [ . 2 2 2 ]
41:  [ . 2 3 3 ]     [ . 2 2 2 ]
42:  [ . 2 3 4 ]     [ . 2 2 4 ]
43:  [ . 2 4 . ]     [ . 2 4 4 ]
44:  [ . 2 4 1 ]     [ . 2 4 4 ]
45:  [ . 2 4 2 ]     [ . 2 4 4 ]
46:  [ . 2 4 3 ]     [ . 2 4 4 ]
47:  [ . 2 4 4 ]     [ . 2 4 4 ]
48:  [ . 2 4 5 ]     [ . 2 4 4 ]
49:  [ . 2 4 6 ]     [ . 2 4 6 ]

        RGS(3,5)        F(5)
 1:  [ . . . ]       [ . . . ]
 2:  [ . . 1 ]       [ . . . ]
 3:  [ . . 2 ]       [ . . . ]
 4:  [ . . 3 ]       [ . . . ]
 5:  [ . . 4 ]       [ . . . ]
 6:  [ . . 5 ]       [ . . 5 ]
 7:  [ . 1 . ]       [ . . . ]
 8:  [ . 1 1 ]       [ . . . ]
 9:  [ . 1 2 ]       [ . . . ]
10:  [ . 1 3 ]       [ . . . ]
11:  [ . 1 4 ]       [ . . . ]
12:  [ . 1 5 ]       [ . . 5 ]
13:  [ . 2 . ]       [ . . . ]
14:  [ . 2 1 ]       [ . . . ]
15:  [ . 2 2 ]       [ . . . ]
16:  [ . 2 3 ]       [ . . . ]
17:  [ . 2 4 ]       [ . . . ]
18:  [ . 2 5 ]       [ . . 5 ]
19:  [ . 3 . ]       [ . . . ]
20:  [ . 3 1 ]       [ . . . ]
21:  [ . 3 2 ]       [ . . . ]
22:  [ . 3 3 ]       [ . . . ]
23:  [ . 3 4 ]       [ . . . ]
24:  [ . 3 5 ]       [ . . 5 ]
25:  [ . 4 . ]       [ . . . ]
26:  [ . 4 1 ]       [ . . . ]
27:  [ . 4 2 ]       [ . . . ]
28:  [ . 4 3 ]       [ . . . ]
29:  [ . 4 4 ]       [ . . . ]
30:  [ . 4 5 ]       [ . . 5 ]
31:  [ . 5 . ]       [ . 5 5 ]
32:  [ . 5 1 ]       [ . 5 5 ]
33:  [ . 5 2 ]       [ . 5 5 ]
34:  [ . 5 3 ]       [ . 5 5 ]
35:  [ . 5 4 ]       [ . 5 5 ]
36:  [ . 5 5 ]       [ . 5 5 ]
37:  [ . 5 6 ]       [ . 5 5 ]
38:  [ . 5 7 ]       [ . 5 5 ]
39:  [ . 5 8 ]       [ . 5 5 ]
40:  [ . 5 9 ]       [ . 5 5 ]
41:  [ . 5 10 ]      [ . 5 10 ]
```

Figure 15.2-E: Length-4 F-increment restricted growth strings with maximal increment 2 and the corresponding array of values of F (left), and length-3 RGSs with maximal increment 5 (right). Dots denote zeros.

To obtain a different generalization of the RGS for set partitions we rewrite the condition $s_k \leq i +$

$\max_{j<k}(s_j)$ for the RGS considered in the previous section:

$$s_k \leq M(k)+i \quad \text{where} \quad M(0) = 0 \quad \text{and} \tag{15.2-5a}$$

$$M(k+1) = \begin{cases} s_{k+1} & \text{if} \quad s_{k+1}-s_k > 0 \\ M(k) & \text{else} \end{cases} \tag{15.2-5b}$$

The function $M(k)$ is $\max_{j<k}(s_j)$ in notational disguise. We define *F-increment* RGSs with respect to a function F as follows:

$$s_k \leq F(k)+i \quad \text{where} \quad F(0) = 0 \quad \text{and} \tag{15.2-6a}$$

$$F(k+1) = \begin{cases} s_{k+1} & \text{if} \quad s_{k+1}-s_k = i \\ F(k) & \text{else} \end{cases} \tag{15.2-6b}$$

The function $F(k)$ is a 'maximum' that is increased only if the last increase $(s_k - s_{k-1})$ was maximal. For $i = 1$ we obtain the RGSs for set partitions. Figure 15.2-E shows all length-4 F-increment RGSs for $i = 2$ (left), and all length-3 RGSs for $i = 5$ (right), together with the arrays of F-values. The listings were created with the program [FXT: comb/rgs-fincr-demo.cc]. It uses the implementation [FXT: `class rgs_fincr` in comb/rgs-fincr.h]:

```
class rgs_fincr
{
public:
    ulong *s_;  // restricted growth string
    ulong *f_;  // values F(k)
    ulong n_;   // Length of strings
    ulong i_;   // s[k] <= f[k]+i
  [--snip--]

    ulong next()
    // Return index of first changed element in s[],
    // Return zero if current string is the last
    {
        ulong k = n_;

    start:
        --k;
        if ( k==0 )  return 0;

        ulong sk = s_[k] + 1;
        ulong m1 = f_[k-1];
        ulong mp = m1 + i_;
        if ( sk > mp )  // "carry"
        {
            s_[k] = 0;
            goto start;
        }

        s_[k] = sk;
        if ( sk==mp )  m1 += i_;
        for (ulong j=k; j<n_; ++j )  f_[j] = m1;

        return k;
    }
  [--snip--]
```

The sequences of numbers of F-increment RGSs with increments i =1, 2, 3, and 4, start

n:	0	1	2	3	4	5	6	7	8	9
i=1:	1	2	5	15	52	203	877	4140	21147	115975
i=2:	1	3	11	49	257	1539	10299	75905	609441	5284451
i=3:	1	4	19	109	742	5815	51193	498118	5296321	60987817
i=4:	1	5	29	201	1657	15821	170389	2032785	26546673	376085653

These are respectively entries A000110 (Bell numbers), A004211, A004212, and A004213 of [214]. In general, the number $F_{n,i}$ of F-increment RGSs (length n, with increment i) is

$$F_{n,i} = \sum_{k=0}^{n} i^{n-k}\, S(n,k) \tag{15.2-7}$$

where $S(n,k)$ are the Stirling numbers of the second kind. The exponential generating functions are

$$\sum_{n=0}^{\infty} F_{n,i} \frac{x^n}{n!} = \exp\left(\frac{\exp(i\,x)-1}{i}\right) \tag{15.2-8}$$

The ordinary generating functions are

$$\sum_{n=0}^{\infty} F_{n,i}\, x^n = \sum_{n=0}^{\infty} \frac{x^n}{\prod_{k=1}^{n} 1 - i\,k\,x} \tag{15.2-9}$$

Chapter 16

A string substitution engine

```
Number of symbols = 3
Start: x
Rules:
  x --> A
  A --> Bx
  B --> Bx
-------------
0:   (#=1)
  x
1:   (#=1)
  A
2:   (#=2)
  Bx
3:   (#=3)
  BxA
4:   (#=5)
  BxABx
5:   (#=8)
  BxABxBxA
6:   (#=13)
  BxABxBxABxABx
7:   (#=21)
  BxABxBxABxABxBxABxBxA
8:   (#=34)
  BxABxBxABxABxBxABxBxABxBxABxBxABx
9:   (#=55)
  BxABxBxABxABxBxABxBxABxABxBxABxABxBxABxBxABxABxBxABxBxA
```

Figure 16.0-A: Rules, axiom, and first steps of the evolution for a specific substitution engine.

String substitution: A finite set of symbols ('alphabet'), a set of substitution rules that map symbols to words (strings of symbols) and a start value ('axiom'). The rules have to be applied in parallel. An example, let $\{x, A, B\}$ be the symbols and $\{x \mapsto A, A \mapsto Bx, B \mapsto Bx\}$ the substitution rules. Start with a single x. The observed evolution is shown in figure 16.0-A. It is trivial to implement such a specific system, for the above:

```
void fx(ulong n);
void fA(ulong n);
void fB(ulong n);

void fx(ulong n)
{
    if ( 0==n ) { cout << "x"; return; }
    fA(n-1);
}

void fA(ulong n)
{
    if ( 0==n ) { cout << "A"; return; }
    fB(n-1);  fx(n-1);
}

void fB(ulong n)
{
```

```
    if ( 0==n ) { cout << "B"; return; }
    fB(n-1);  fx(n-1);
}
int main()
{
    ulong n = 10;  // max depth
    for (ulong k=0; k<=n; ++k)
    {
        fx(k);
        cout << endl;
    }
    return 0;
}
```

A utility class to create string substitution engines at run time is given in [FXT: `class string_subst` in comb/stringsubst.h]:

```
class string_subst
{
public:
    // example values generate rabbit sequence:
    ulong nsym_;    // # of symbols
    // == 2
    char *symbol_;   // alphabet
    // == { '0', '1' };
    char **symrule_; // symrule_[i] string to replace i-th symbol with
    // == { "0", "1",  "1", "10" };  for  0 |-> 1,  1 |-> 10
    ulong xlate_[256];  // translate char --> rule
    //  'a' --> symrule[xlate_['a']]
    ulong cmax_; // max string length
    char  *cc_;  // string to hold result
    ulong ctc_;  // count chars of result actually produced

public:
    string_subst(int cmax, int nsym, char*const* symrule);
```

Rules and symbols have to be supplied on construction. The member function `subst` takes an axiom and the number of generations as arguments and returns the numbers of produced characters. It calls the recursive function `do_subst`:

```
    void do_subst(ulong n, const char *rule)
    {
        if ( 0==n )  // add symbols to string:
        {
            for (ulong i=0;  ctc_<cmax_;  ++i)
            {
                char c = rule[i];
                if ( 0==c )  break;
                cc_[ctc_] = c;
                ++ctc_;
            }
        }
        else  // recurse:
        {
            for (ulong i=0;  0!=rule[i];  ++i)
            {
                ulong r = (ulong)rule[i];
                ulong x = xlate_[r];
                do_subst(n-1, symrule_[x]);  // recursion
            }
        }
    }
    ulong subst(ulong maxn, const char *start)
    // maxn:=number of generations,  start:=axiom
    {
        ctc_ = 0;
        do_subst(maxn, start);
        cc_[ctc_] = 0;  // terminate string
        return  ctc_;
    }
```

The resulting string is stored in an array of characters that can be accessed via the member function

`string()`. The characters that do not fit into the array are silently discarded. Using an array of one million characters, the rules and axiom as follows

```
Number of symbols = 5
Start: a
Rules:
  a --> -bF+aFa+Fb-
  b --> +aF-bFb-Fa+
  + --> +
  - --> -
  F --> F
```

Then one obtains (displaying only the string lengths from the fourth generation on):

```
 0:   (#=1)
   a
 1:   (#=11)
   -bF+aFa+Fb-
 2:   (#=51)
   -+aF-bFb-Fa+F+-bF+aFa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-
 3:   (#=211)
   -+-bF+aFa+Fb-F-+aF-bFb-Fa+F+aF-bFb-Fa+-F-bF+aFa+Fb-+F+-+aF-bFb-  \
    Fa+F+-bF+aFa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-F-+aF-bFb-Fa+F+-bF+a  \
    Fa+Fb-F-bF+aFa+Fb-+F+aF-bFb-Fa+-+F+-bF+aFa+Fb-F-+aF-bFb-Fa+F+a  \
    F-bFb-Fa+-F-bF+aFa+Fb-+-
 4:   (#=851)
 5:   (#=3411)
 6:   (#=13651)
 7:   (#=54611)
 8:   (#=218451)
 9:   (#=873811)
10:   (#=1000000)  // <-- truncation
11:   (#=1000000)
```

The computation takes a split second only. The strings generated by the shown rule correspond to the Hilbert curves in successively finer resolutions if one initializes

```
position:         (x, y)   := (0,0)
direction:        (dx, dy) := (1,0)
```

and identifies:

```
'-'  ==  "turn left":   { t= dx; dx=-dy; dy=t; }
'+'  ==  "turn right":  { t=-dx; dx= dy; dy=t; }
'F'  ==  "advance":     { line(x, y, x+dx, y+dy);  x+=dx; y+=dy; }
either 'a' or 'b'  ==   (ignore)
```

The program [FXT: ds/stringsubst-demo.cc] prints evolutions for several rules. A 3-dimensional Hilbert (space-filling) curve is generated by the rule `a` $\mapsto$ `^<aF^<aFa-F^>>aFavF+>>aFa-F>a->`, all other symbols are mapped to themselves. The curve is obtained by interpreting the symbols (unequal to `F`) of the string as rotations [FXT: ds/stringsubst-hilbert3d-demo.cc].

The number of symbols that occur in each generation can be computed by a simple matrix power. With i different symbols let $n_j(k)$ be the number of occurrences of the j-th symbol at generation k. Then $n_j(0)$ is determined by the axiom and

$$\vec{N}(k) \quad := \quad [n_1(k), n_2(k), \ldots, n_i(k)]^T \tag{16.0-1}$$

can be computed as

$$\vec{N}(k) \quad = \quad M^k \, \vec{N}(0) \tag{16.0-2}$$

where M is a matrix with columns according to the occurrences of the symbols in the transfer rules. For example, with the rules and axiom

```
Number of symbols = 3
Start: x
Rules:                  x  A  B
  x --> A    1 --> [0, 1, 0]
  A --> Bx   2 --> [1, 0, 1]  =: transpose(M)
  B --> Bx   3 --> [1, 0, 1]
```

we obtain

$$M \quad = \quad \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \qquad M^k \; = \; \begin{bmatrix} F_{k-1} & F_k & F_k \\ F_{k-2} & F_{k-1} & F_{k-1} \\ F_{k-1} & F_k & F_k \end{bmatrix} \qquad k \geq 2 \tag{16.0-3}$$

where F_k is the k-th Fibonacci number. For the tenth generation, $k = 10$:

$$M^{10}\,[1,0,0]^T \;=\; \begin{bmatrix} 34 & 55 & 55 \\ 21 & 34 & 34 \\ 34 & 55 & 55 \end{bmatrix} [1,0,1]^T \;=\; [34,21,34]^T \tag{16.0-4}$$

So we have 34 '`x`', 21 '`A`' and 34 '`B`' or a total of 89 symbols.

For the engine corresponding to a 3D Hilbert curve

```
Number of symbols = 5
Start: a
Rules:
  a --> -bF+aFa+Fb-   1 --> [2, 2, 2, 2, 3]
  b --> +aF-bFb-Fa+   2 --> [2, 2, 2, 2, 3]
  + --> +             3 --> [0, 0, 1, 0, 0] =: transpose(m)
  - --> -             4 --> [0, 0, 0, 1, 0]
  F --> F             5 --> [0, 0, 0, 0, 1]
```

we find

```
 k     total    [   n(a),    n(b),    n(+),   n(-),    n(F) ]
 0         1    [      1,       0,       0,      0,       0 ]
 1        11    [      2,       2,       2,      2,       3 ]
 2        51    [      8,       8,      10,     10,      15 ]
 3       211    [     32,      32,      42,     42,      63 ]
 4       851    [    128,     128,     170,    170,     255 ]
 5      3411    [    512,     512,     682,    682,    1023 ]
 6     13651    [   2048,    2048,    2730,   2730,    4095 ]
 7     54611    [   8192,    8192,   10922,  10922,   16383 ]
 8    218451    [  32768,   32768,   43690,  43690,   65535 ]
 9    873811    [ 131072,  131072,  174762, 174762,  262143 ]
10   3495251    [ 524288,  524288,  699050, 699050, 1048575 ]
```

In chapter 36 on page 691 the string substitution idea is used to construct fast iterations for various functions whose power series are determined by the limiting string.

An efficient algorithm for finding the n-th letter of the string produced by k iterations of a substitution rule is described in [213]. The problem of the identification of a finite automaton (substitution engine) that generates a given infinite string is discussed in [164]. The mathematical properties of sequences generated by finite automata are studied in [10].

Chapter 17

Necklaces and Lyndon words

A sequence that is minimal among all its cyclic rotations is called a *necklace* (see section 3.8.1 on page 128 for the definition in terms of equivalence classes). When there are k possible values for each element one talks about an n-bead, k-color (or k-ary length-n) necklaces. We restrict our attention to the case were only two sorts of beads are allowed and represent them by 0 and 1.

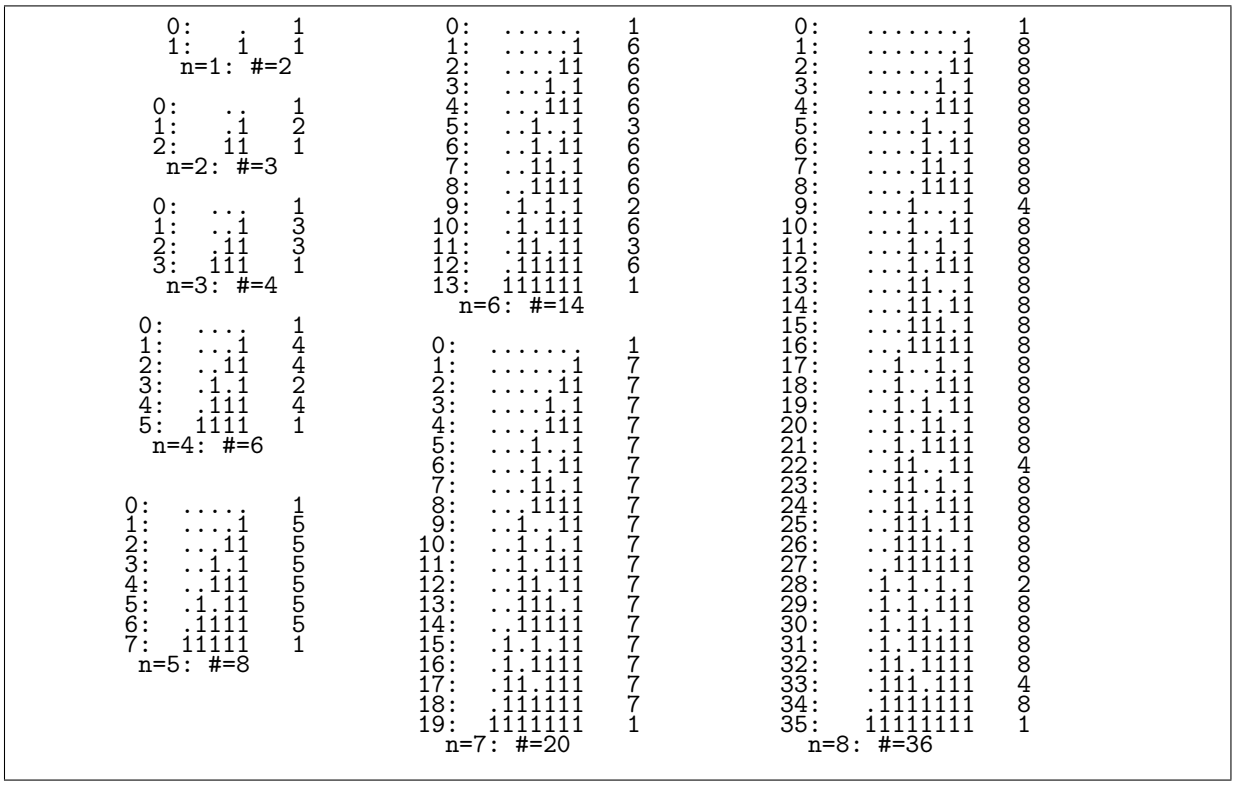


Figure 17.0-A: All binary necklaces of lengths up to 8 and their periods. Dots represent zeros.

Scanning all binary words of length n as to whether they are necklaces can easily be achieved by testing whether the word `x` is equal to the return value of the function `bit_cyclic_min(x,n)` shown in section 1.12 on page 27. For n up to 8 one obtains the sequences of binary necklaces shown in figure 17.0-A. As 2^n words have to be tested this approach gets ineffective for large n. Luckily there is both a much better algorithm for generating all necklaces and a formula for their number.

Not all necklaces are created equal. Each necklace can be assigned a period that is a divisor of the length.

That period is the smallest (nonzero) cyclic shift that transforms the word into itself. The periods are given directly right to each necklace in figure 17.0-A. For n prime the only periodic necklaces are those two that contain all ones or zeros. Aperiodic (or equivalently, period equals length) necklaces are called *Lyndon words*.

For a length-n binary word x the function `bit_cyclic_period(x,n)` from section 1.12 on page 27 returns the period of the word.

17.1 Generating all necklaces

We give several methods to generate all necklaces of a given size. An efficient algorithm for the generation of bracelets (see section 3.8.1.4 on page 129) is given in [207].

17.1.1 The FKM algorithm

```
 1:    [ . . . . ]   j=1  N
 2:    [ . . . 1 ]   j=4  N  L
 3:    [ . . . 2 ]   j=4  N  L
 4:    [ . . 1 . ]   j=3
 5:    [ . . 1 1 ]   j=4  N  L
 6:    [ . . 1 2 ]   j=4  N  L
 7:    [ . . 2 . ]   j=3
 8:    [ . . 2 1 ]   j=4  N  L
 9:    [ . . 2 2 ]   j=4  N  L
10:    [ . 1 . 1 ]   j=2  N
11:    [ . 1 . 2 ]   j=4  N  L
12:    [ . 1 1 . ]   j=3
13:    [ . 1 1 1 ]   j=4  N  L
14:    [ . 1 1 2 ]   j=4  N  L
15:    [ . 1 2 . ]   j=3
16:    [ . 1 2 1 ]   j=4  N  L
17:    [ . 1 2 2 ]   j=4  N  L
18:    [ . 2 . 2 ]   j=2  N
19:    [ . 2 1 . ]   j=3
20:    [ . 2 1 1 ]   j=4  N  L
21:    [ . 2 1 2 ]   j=4  N  L
22:    [ . 2 2 . ]   j=3
23:    [ . 2 2 1 ]   j=4  N  L
24:    [ . 2 2 2 ]   j=4  N  L
25:    [ 1 1 1 1 ]   j=1  N
26:    [ 1 1 1 2 ]   j=4  N  L
27:    [ 1 1 2 1 ]   j=3
28:    [ 1 1 2 2 ]   j=4  N  L
29:    [ 1 2 1 2 ]   j=2  N
30:    [ 1 2 2 1 ]   j=3
31:    [ 1 2 2 2 ]   j=4  N  L
32:    [ 2 2 2 2 ]   j=1  N
32 (4, 3) pre-necklaces.
24 necklaces and 18 Lyndon words.
```

```
 1:    [ . . . . . . ]   j=1  N
 2:    [ . . . . . 1 ]   j=6  N  L
 3:    [ . . . . 1 . ]   j=5
 4:    [ . . . . 1 1 ]   j=6  N  L
 5:    [ . . . 1 . . ]   j=4
 6:    [ . . . 1 . 1 ]   j=6  N  L
 7:    [ . . . 1 1 . ]   j=5
 8:    [ . . . 1 1 1 ]   j=6  N  L
 9:    [ . . 1 . . 1 ]   j=3  N
10:    [ . . 1 . 1 . ]   j=5
11:    [ . . 1 . 1 1 ]   j=6  N  L
12:    [ . . 1 1 . . ]   j=4
13:    [ . . 1 1 . 1 ]   j=6  N  L
14:    [ . . 1 1 1 . ]   j=5
15:    [ . . 1 1 1 1 ]   j=6  N  L
16:    [ . 1 . 1 . 1 ]   j=2  N
17:    [ . 1 . 1 1 . ]   j=5
18:    [ . 1 . 1 1 1 ]   j=6  N  L
19:    [ . 1 1 . 1 1 ]   j=3  N
20:    [ . 1 1 1 . 1 ]   j=4
21:    [ . 1 1 1 1 . ]   j=5
22:    [ . 1 1 1 1 1 ]   j=6  N  L
23:    [ 1 1 1 1 1 1 ]   j=1  N
23 (6, 2) pre-necklaces.
14 necklaces and 9 Lyndon words.
```

Figure 17.1-A: Ternary length-4 (left) and binary length-6 (right) pre-necklaces as generated by the FKM algorithm. Dots are used for zeros, necklaces are marked with '`N`', Lyndon words with '`L`'.

The following algorithm for generation of all necklaces actually produces *pre-necklaces* a subset of which are the necklaces. A pre-necklace is a string that is the prefix of some necklace. The *FKM algorithm* (for Fredericksen, Kessler, Maiorana) to generate all k-ary length-n pre-necklaces proceeds as follows:

1. Initialize the word $F = [f_1, f_2, \ldots, f_n]$ to all zeros. Set $j = 1$.
2. (Visit pre-necklace F. If j divides n then F is a necklace. If j equals n then F is a Lyndon word.)
3. Find the largest index j so that $f_j < k-1$. If there is no such index (then $F = [k-1, k-1, \ldots, k-1]$, the last necklace), then terminate.
4. Increment f_j. Fill the suffix starting at f_{j+1} with copies of $[f_1, \ldots, f_j]$. Goto step 2.

The crucial steps are [FXT: comb/necklace-fkm-demo.cc]:

```
    for (ulong i=1; i<=n; ++i)  f[i] = 0;  // Initialize to zero
    bool nq = 1;  // whether pre-necklace is a necklace
    bool lq = 0;  // whether pre-necklace is a Lyndon word
    ulong j = 1;
    while ( 1 )
    {
        // Print necklace:
        cout << setw(4) << pct << ":";
        print_vec("    ", f+1, n, true);
        cout << "   j=" << j;
        if ( nq )  cout << "  N";
        if ( lq )  cout << "  L";
        cout << endl;

        // Find largest index where we can increment:
        j = n;
        while ( f[j]==k-1 )  { --j; };

        if ( j==0 )  break;

        ++f[j];

        // Copy periodically:
        for (ulong i=1,t=j+1; t<=n; ++i,++t)  f[t] = f[i];

        nq = ( (n%j)==0 );  // necklace if j divides n
        lq = ( j==n );      // Lyndon word if j equals n
    }
```

Two example runs are shown in figure 17.1-A. An efficient implementation of the algorithm is [FXT: `class necklace` in comb/necklace.h]:

```
class necklace
{
public:
    ulong *a_;  // the string
    ulong *dv_; // delta sequence of divisors of n
    ulong n_;   // length of strings
    ulong m1_;  // m-ary strings, m1=m-1
    ulong j_;   // period of the word (if necklaces)
public:
    necklace(ulong m, ulong n)
    {
        n_ = ( n ? n : 1 );  // at least one
        m1_ = ( m>1 ? m-1 : 1); // at least two
        a_ = new ulong[n_+1];
        dv_ = new ulong[n_+1];
        for (ulong j=1; j<=n; ++j)  dv_[j] = ( 0==(n_%j ) );  // divisors
        first();
    }
  [--snip--]

    void first()
    {
        for (ulong j=0; j<=n_; ++j)  a_[j] = 0;
        j_ = 1;
    }
  [--snip--]
```

The method to compute the next pre-necklace is

```
    ulong next_pre()  // next pre-necklace
    // return j (zero when finished)
    {
        // Find rightmost digit that can be incremented:
```

```
        ulong j = n_;
        while ( a_[j] == m1_ )  { --j; }
        // Increment:
        // if ( 0==j_ )   return 0;  // last
        ++a_[j];
        // Copy periodically:
        for (ulong k=j+1; k<=n_; ++k)  a_[k] = a_[k-j];
        j_ = j;
        return  j;
    }
```

Note the commented out return with the last word, this gives a speedup (and no harm is done with the following copying). The array `dv` allows determination whether the current pre-necklace is also a necklace (or Lyndon word) via simple lookups:

```
    bool is_necklace()  const
    {
        return ( 0!=dv_[j_] );  // whether j divides n
    }
    bool is_lyn()  const
    {
        return ( j_==n_ );  // whether j equals n
    }
    ulong next()  // next necklace
    {
        do
        {
            next_pre();
            if ( 0==j_ )  return 0;
        }
        while ( 0==dv_[j_] );  // until j divides n
        return j_;
    }
    ulong next_lyn()  // next Lyndon word
    {
        do
        {
            next_pre();
            if ( 0==j_ )  return 0;
        }
        while ( j_==n_ );  // until j equals n
        return j_;  // == n
    }
};
```

The rate of generation for pre-necklaces is about 98 M/s for base 2, 140 M/s for base 3, and 180 M/s for base 4 [FXT: comb/necklace-demo.cc]. An specialization of the algorithm for binary necklaces is [FXT: `class binary_necklace` in comb/binary-necklace.h]. It can generate pre-necklaces at about 124 M/s (when arrays are used instead of pointers), see [FXT: comb/binary-necklace-demo.cc].

The binary necklaces that fit into a machine word can be generated via [FXT: `class bit_necklace` in bits/bit-necklace.h] which produces about 32 million necklaces per second. The program [FXT: bits/bit-necklace-demo.cc] shows the usage of the class.

The binary necklaces of length n can be used as cycle leaders in the length-2^n zip permutation (and its inverse) that is discussed in section 2.5 on page 93. An algorithm for the generation of all irreducible binary polynomials via Lyndon words is described in section 38.6 on page 824.

```
 0 :   a= ......1 =    1    ==   ......1
 1 :   a= .....11 =    3    ==   .....11
 2 :   a= ...1..1 =    9    ==   ...1..1
 3 :   a= ..11.11 =   27    ==   ..11.11
 4 :   a= 1.1...1 =   81    ==   ...11.1
 5 :   a= 111.1.. =  116    ==   ..111.1
 6 :   a= 1.1111. =   94    ==   .1.1111
 7 :   a= ..111.. =   28    ==   ....111
 8 :   a= 1.1.1.. =   84    ==   ..1.1.1
 9 :   a= 11111.1 =  125    ==   .111111
10 :   a= 1111..1 =  121    ==   ..11111
11 :   a= 11.11.1 =  109    ==   .11.111
12 :   a= 1..1..1 =   73    ==   ..1..11
13 :   a= 1.111.. =   92    ==   ..1.111
14 :   a= ..1.11. =   22    ==   ...1.11
15 :   a= 1....1. =   66    ==   ....1.1
16 :   a= 1...111 =   71    ==   ...1111
17 :   a= 1.1.11. =   86    ==   .1.1.11
18 :   a= ....1.. =    4    ==   ......1 <--= sequence restarts
19 :   a= ...11.. =   12    ==   .....11
20 :   a= .1..1.. =   36    ==   ...1..1
21 :   a= 11.11.. =  108    ==   ..11.11
22 :   a= 1...11. =   70    ==   ...11.1
23 :   a= 1.1..11 =   83    ==   ..111.1
24 :   a= 1111.1. =  122    ==   .1.1111
25 :   a= 111.... =  112    ==   ....111
[--snip--]
```

Figure 17.1-B: Generation of all (18) 7-bit Lyndon words as binary representations of the powers modulo 127 of the primitive root 3. The right column gives the cyclic minima. Dots are used for zeros.

17.1.2 Binary Lyndon words with length a Mersenne exponent

The length-n binary Lyndon words for n an exponent of a Mersenne prime $M_n = 2^n - 1$ can be generated efficiently as binary expansions of the powers of a primitive root r of m until the second word with just one bit is reached. With $n = 7$, $m = 127$ and the primitive root $r = 3$ we get the sequence shown in figure 17.1-B. The sequence of minimal primitive roots r_n of the first Mersenne primes $M_n = 2^n - 1$ is entry A096393 of [214]:

```
 2: 2      17: 3       107: 3
 3: 3      19: 3       127: 43
 5: 3      31: 7       521: 3
 7: 3      61: 37      607: 5  <--= 5 is a primitive root of 2**607-1
13: 17     89: 3      1279: 5
```

17.1.3 A constant amortized time (CAT) algorithm

A constant amortized time (CAT) algorithm to generate all k-ary length-n (pre-)necklaces is given in [73]. The crucial part of a recursive algorithm [FXT: comb/necklace-cat-demo.cc] is the function

```
ulong K, N;  // K-ary pre-necklaces of length N
ulong f[N];
void crsms_gen(ulong n, ulong j)
{
    if ( n > N )  visit(j);  // pre-necklace in f[1,...,N]
    else
    {
        f[n] = f[n-j];
        crsms_gen(n+1, j);

        for (ulong i=f[n-j]+1; i<K; ++i)
        {
            f[n] = i;
            crsms_gen(n+1, n);
        }
    }
}
```

After initializing the array with zeros the function must be called with both arguments equal to one. The routine generates about 71 million binary pre-necklaces per second. Ternary and 5-ary pre-necklaces are generated at a rate of about 100 and 113 million per second, respectively.

17.1.4 An order with fewer transitions

```
 1:  .......1        11:  ...11111        21:  ..1.1.11
 2:  ......11        12:  ...111.1        22:  ..1.1111
 3:  .....111        13:  ...1.1.1        23:  ..1.11.1
 4:  .....1.1        14:  ...1.111        24:  ..1..111 <<+1
 5:  ....11.1        15:  ...1..11        25:  ..1..1.1
 6:  ....1111        16:  ..11.111 <<+1   26:  .11.1111 <<+2
 7:  ....1.11        17:  ..11.1.1        27:  .1111111
 8:  ....1..1        18:  ..1111.1        28:  .1.11.11 <<+1
 9:  ...11..1        19:  ..111111        29:  .1.11111
10:  ...11.11        20:  ..111.11        30:  .1.1.111
```

Figure 17.1-C: The 30 binary 8-bit Lyndon words in an order with few changes between successive words. Transitions where more than one bit changes are marked with a '<<'.

$n:$	X_n	$n:$	X_n	$n:$	X_n	$n:$	X_n	$n:$	X_n
1:	0	7:	2	13:	95	19:	2598	25:	85449
2:	0	8:	5	14:	163	20:	4546	26:	155431
3:	0	9:	11	15:	290	21:	8135	27:	284886
4:	0	10:	15	16:	479	22:	14427	28:	522292
5:	1	11:	34	17:	859	23:	26122	29:	963237
6:	1	12:	54	18:	1450	24:	46957	30:	1778145

Figure 17.1-D: Excess (with respect to Gray code) of the number of bits changed.

The following routine generates the binary pre-necklaces words in the order that would be obtained by selecting valid words from the binary Gray code:

```
void xgen(ulong n, ulong j, int x=+1)
{
    if ( n > N )  visit(j);
    else
    {
        if ( -1==x )
        {
            if ( 0==f[n-j] )  { f[n] = 1;  xgen(n+1, n, -x); }
            f[n] = f[n-j];  xgen(n+1, j, +x);
        }
        else
        {
            f[n] = f[n-j];  xgen(n+1, j, +x);
            if ( 0==f[n-j] )  { f[n] = 1;  xgen(n+1, n, -x); }
        }
    }
}
```

The program [FXT: comb/necklace-gray-demo.cc] computes the binary Lyndon words with the given routine. The ordering obtained has fewer transitions between successive elements but is in general not a Gray code (for up to 6-bit words a Gray code is generated). Figure 17.1-C shows the output with 8-bit Lyndon words. The first $2^{\lfloor n/2 \rfloor} - 1$ Lyndon words of length n are in Gray code order. The number X_n of additional transitions of the length-n Lyndon words is, for $n \leq 30$, shown in figure 17.1-D.

17.1.5 An order with at most three changes per transition

An algorithm to generate necklaces in an order such that at most 3 elements change with each update is given in [242]. The recursion can be given as (corrected and shortened) [FXT: comb/necklace-gray3-demo.cc]:

```
long *f; // data in f[1..m],  f[0] = 0
long N; // word length
int k;  // k-ary necklaces, k==sigma in the paper

void gen3(int z, int t, int j)
{
```

```
 1:  .1111111          13:  ...1...1          25:  ..1.1111
 2:  .111.111          14:  ...1.1.1          26:  ..1.11.1
 3:  .11.1111 <<+1     15:  ...1.111          27:  ..1.1.11 <<+1
 4:  .1.1.111 <<+2     16:  .....111          28:  ..1..1.1 <<+2
 5:  .1.1.1.1          17:  .....1.1          29:  ..1..111
 6:  .1.11.11 <<+2     18:  .......1          30:  ..11.111
 7:  .1.11111          19:  ........          31:  ..11.1.1
 8:  ...11111          20:  .....11 <<+1      32:  ..11..11 <<+1
 9:  ...111.1          21:  ....1.11          33:  ..111.11
10:  ...11..1          22:  ....1..1          34:  ..1111.1 <<+1
11:  ...11.11          23:  ....11.1          35:  ..111111
12:  ...1..11          24:  ....1111          36:  11111111 <<+1
```

Figure 17.1-E: The 30 binary 8-bit necklaces in an order with at most 3 changes per transition. Transitions where more than one bit changes are marked with a '<<'.

$n:$	X_n	$n:$	X_n	$n:$	X_n	$n:$	X_n	$n:$	X_n
1:	0	7:	6	13:	200	19:	6462	25:	239008
2:	1	8:	12	14:	360	20:	11722	26:	441370
3:	2	9:	20	15:	628	21:	21234	27:	816604
4:	2	10:	38	16:	1128	22:	38754	28:	1515716
5:	2	11:	64	17:	1998	23:	70770	29:	2818928
6:	4	12:	116	18:	3606	24:	129970	30:	5256628

Figure 17.1-F: Excess (with respect to Gray code) of number of bits changed.

```
    if ( t > N ) { visit(j); }
    else
    {
        if ( (z&1)==0 ) // z (number of elements ==(k-1)) is even?
        {
            for (int i=f[t-j]; i<=k-1; ++i)
            {
                f[t] = i;
                gen3( z+(i!=k-1), t+1, (i!=f[t-j]?t:j) );
            }
        }
        else
        {
            for (int i=k-1; i>=f[t-j]; --i)
            {
                f[t] = i;
                gen3( z+(i!=k-1), t+1, (i!=f[t-j]?t:j) );
            }
        }
    }
}
```

The variable `z` counts the number of maximal elements. The output with length-8 binary necklaces is shown in figure 17.1-E. Selecting the necklaces from the reversed list of complemented Gray codes of the n-bit binary words produces the same list.

17.1.6 Binary necklaces of length 2^n via Gray-cycle leaders *

The algorithm for the generation of cycle leaders for the Gray code permutation given section 2.8.1 on page 97 and relation 1.18-10c on page 49, written as

$$S_k\, Y\, x \quad = \quad Y\, g^k\, x \tag{17.1-1}$$

(Y is the yellow code, or bit-wise Reed-Muller transform) allow us to generate the necklaces of length 2^n: The cyclic shifts of $Y\, x$ are equal to $Y\, g^k\, x$ for $k = 0, \ldots, l-1$ where l is the cycle length. Figure 17.1-G shows the correspondence between cycles of the Gray code permutation and cyclic shifts, it was generated with the program [FXT: comb/necklaces-via-gray-leaders-demo.cc].

```
 k = 7: 16 cycles of length= 8
L=  1.......  [ 1....... ]                 L=  1..1.11.  [ ...1.11. ]
L=  1......1  [ .1111111 ]                 --> 11.111.1  [ ....1.11 ]
L=  1.....1.  [ ..1.1.1. ]                 --> 1.11..11  [ 1....1.1 ]
L=  1.....11  [ 11.1.1.1 ]                 --> 111.1.1.  [ 11....1. ]
L=  1....1..  [ .1..11.. ]                 --> 1..11111  [ .11....1 ]
L=  1....1.1  [ 1.11..11 ]                 --> 11.1....  [ 1.11.... ]
L=  1....11.  [ 111..11. ]                 --> 1.111...  [ .1.11... ]
L=  1....111  [ ...11..1 ]                 --> 111..1..  [ ..1.11.. ]
L=  1..1....  [ .111.... ]
L=  1..1...1  [ 1...1111 ]                 L=  1..1.111  [ 111.1..1 ]
L=  1..1..1.  [ 11.11.1. ]                 --> 11.111..  [ 1111.1.. ]
L=  1..1..11  [ ..1..1.1 ]                 --> 1.11..1.  [ .1111.1. ]
L=  1..1.1..  [ 1.1111.. ]                 --> 111.1.11  [ ..1111.1 ]
L=  1..1.1.1  [ .1....11 ]                 --> 1..1111.  [ 1..1111. ]
L=  1..1.11.  [ ...1.11. ]                 --> 11.1...1  [ .1..1111 ]
L=  1..1.111  [ 111.1..1 ]                 --> 1.111..1  [ 1.1..111 ]
                                           --> 111..1.1  [ 11.1..11 ]
```

Figure 17.1-G: Left: the cycle leaders (minima) L of the Gray code permutation where is highest bit has index 7 and their bit-wise Reed-Muller transforms $Y(L)$. Right: the last two cycles and the transforms of their elements.

If no better algorithm for the cyclic leaders of the Gray code permutation was known we could generate them as $Y^{-1}(N) = Y(N)$ where N are the necklaces of length 2^n. The same idea, and relation 1.18-11b on page 49, give the relation

$$S_k\, B\, x \quad = \quad B\, e^{-k}\, x \tag{17.1-2}$$

where B is the blue code and e the reversed Gray code.

17.1.7 Binary necklaces via cyclic shifts and complements *

```
  n = 3          n = 6          n = 7          n = 8          [n=8 cont.]
1:  ..1       1:  .....1     1:  ......1    1:  .......1    19:  ..11..11
2:  .11       2:  ...11      2:  .....11    2:  ......11    20:  .....1.1
3:  111       3:  ...111     3:  ....111    3:  .....111    21:  ....1.11
              4:  ..1111     4:  ...1111    4:  ....1111    22:  ...1.111
  n = 4       5:  .11111     5:  ..11111    5:  ...11111    23:  ..1.1111
1:  ...1      6:  111111     6:  .111111    6:  ..111111    24:  .1.11111
2:  ..11      7:  ..11.1     7:  1111111    7:  .1111111    25:  ..1.11.1
3:  .111      8:  .11.11     8:  ..111.1    8:  11111111    26:  .1.11.11
4:  1111      9:  ...1.1     9:  ...11.1    9:  ..1111.1    27:  ...1.1.1
5:  .1.1     10:  ..1.11    10:  ..11.11   10:  ...111.1    28:  ..1.1.11
             11:  .1.111    11:  .11.111   11:  ..111.11    29:  .1.1.111
  n = 5      12:  .1.1.1    12:  ....1.1   12:  .111.111    30:  .1.1.1.1
1:  ....1    13:  ..1..1    13:  ...1.11   13:  ....11.1    31:  ....1..1
2:  ...11                   14:  ..1.111   14:  ...11.11    32:  ...1..11
3:  ..111                   15:  .1.1111   15:  ..11.111    33:  ..1..111
4:  .1111                   16:  ..1.1.1   16:  .11.1111    34:  ..1..1.1
5:  11111                   17:  .1.1.11   17:  ..11.1.1    35:  ...1...1
6:  ..1.1                   18:  ...1..1   18:  ...11..1
7:  .1.11                   19:  ..1..11
```

Figure 17.1-H: Nonzero binary necklaces of lengths $n = 3, 4, \ldots, 8$ as generated by the shift and complement algorithm.

A recursive algorithm to generate all nonzero binary necklaces via cyclic shifts and complements of the lowest bit is described in [201]. An implementation of the method is given in [FXT: comb/necklace-sigma-tau-demo.cc]:

```
inline ulong sigma(ulong x)  { return bit_rotate_left(x, 1, n); }
inline ulong tau(ulong x)  { return  x ^ 1; }

void search(ulong y)
{
    visit(y);
    ulong t = y;
```

```
    while ( 1 )
    {
        t = sigma(t);
        ulong x = tau(t);
        if ( (x&1) && (x == bit_cyclic_min(x, n)) )  search(x);
        else  break;
    }
}
```

The initial call is `search(1)`. The generated ordering for lengths $n = 3, 4, \ldots, 8$ is shown in figure 17.1-H.

17.2 The number of binary necklaces

n :	N_n	n :	N_n	n :	N_n	n :	N_n
1:	2	11:	188	21:	99880	31:	69273668
2:	3	12:	352	22:	190746	32:	134219796
3:	4	13:	632	23:	364724	33:	260301176
4:	6	14:	1182	24:	699252	34:	505294128
5:	8	15:	2192	25:	1342184	35:	981706832
6:	14	16:	4116	26:	2581428	36:	1908881900
7:	20	17:	7712	27:	4971068	37:	3714566312
8:	36	18:	14602	28:	9587580	38:	7233642930
9:	60	19:	27596	29:	18512792	39:	14096303344
10:	108	20:	52488	30:	35792568	40:	27487816992

Figure 17.2-A: The number of binary necklaces for $n \leq 40$.

n :	L_n	n :	L_n	n :	L_n	n :	L_n
1:	2	11:	186	21:	99858	31:	69273666
2:	1	12:	335	22:	190557	32:	134215680
3:	2	13:	630	23:	364722	33:	260300986
4:	3	14:	1161	24:	698870	34:	505286415
5:	6	15:	2182	25:	1342176	35:	981706806
6:	9	16:	4080	26:	2580795	36:	1908866960
7:	18	17:	7710	27:	4971008	37:	3714566310
8:	30	18:	14532	28:	9586395	38:	7233615333
9:	56	19:	27594	29:	18512790	39:	14096302710
10:	99	20:	52377	30:	35790267	40:	27487764474

Figure 17.2-B: The number of binary Lyndon words for $n \leq 40$.

The number of binary necklaces of length n equals

$$N_n \;=\; \frac{1}{n}\sum_{d\backslash n}\varphi(d)\,2^{n/d} \;=\; \frac{1}{n}\sum_{j=1}^{n}2^{\gcd(j,n)} \tag{17.2-1}$$

Replace 2 by k to get the number for k different sorts of beads (possible values at each digit). The values for $n \leq 40$ are shown in figure 17.2-A. The sequence is entry A000031 of [214].

The number of Lyndon words (aperiodic necklaces) equals

$$L_n \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(d)\,2^{n/d} \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(n/d)\,2^{d} \tag{17.2-2}$$

The Möbius function μ is defined in relation 35.1-6 on page 657. The values for $n \leq 40$ are given in figure 17.2-B. The sequence is entry A001037 of [214]. For prime $n = p$ we have $L_p = N_p - 2$ and

$$L_p \;=\; \frac{2^p - 2}{p} \;=\; \frac{1}{p} \sum_{k=1}^{p-1} \binom{p}{k} \tag{17.2-3}$$

The latter form transpires that there are exactly $\binom{p}{k}/p$ Lyndon words with k ones for $1 \leq k \leq p-1$. The difference of two is due to the necklaces that consist of all zeros or ones. The number of irreducible binary polynomials (see section 38.5 on page 823) of degree n also equals L_n. For the equivalence between necklaces and irreducible polynomials see section 38.6 on page 824.

Let d be a divisor of n. There are 2^n binary words of length n, each having some period d that divides n. There are d different shifts of the corresponding word, thereby

$$2^n \;=\; \sum_{d \backslash n} d\, L_d \tag{17.2-4}$$

Möbius inversion gives relation 17.2-2. The necklaces of length n and period d are obtained by concatenation of n/d Lyndon words of length d, so

$$N_n \;=\; \sum_{d \backslash n} L_d \tag{17.2-5}$$

We note the relations (see section 35.1.4 on page 659)

$$(1 - 2\,y) \;=\; \prod_{k=1}^{\infty} (1 - y^k)^{L_k} \tag{17.2-6a}$$

$$\sum_{k=1}^{\infty} L_k\, y^k \;=\; \sum_{k=1}^{\infty} \frac{-\,\mu(k)}{k} \log\left(1 - 2\,y^k\right) \tag{17.2-6b}$$

Defining

$$\eta_B(x) \;:=\; \prod_{k=1}^{\infty} \left(1 - B\,y^k\right) \tag{17.2-7a}$$

we have

$$\eta_2(x) \;=\; \prod_{k=1}^{\infty} (1 - y^k)^{N_k} \tag{17.2-7b}$$

$$\eta_2(x) \;=\; \prod_{k=1}^{\infty} \eta_1(y^k)^{L_k} \tag{17.2-7c}$$

17.3 The number of binary necklaces with fixed content

Let $N_{\binom{n}{n_0}}$ be the number of binary length-n necklaces with exactly n_0 zeros (and $n_1 = n - n_0$ ones). We call these *necklaces with fixed density*. One has

$$N_{\binom{n}{n_0}} \;=\; \frac{1}{n} \sum_{j \backslash g} \varphi(j) \binom{n/j}{d/j} \tag{17.3-1}$$

where $g = \gcd(n, n_0)$. One has the symmetry relation $N_{\binom{n}{n_0}} = N_{\binom{n}{n-n_0}} = N_{\binom{n}{n_1}}$. A table of small values is given in figure 17.3-A.

n:	N_n	$N_{\binom{n}{0}}$	$N_{\binom{n}{1}}$	$N_{\binom{n}{2}}$	$N_{\binom{n}{3}}$	$N_{\binom{n}{4}}$	$N_{\binom{n}{5}}$	$N_{\binom{n}{6}}$	$N_{\binom{n}{7}}$	$N_{\binom{n}{8}}$	$N_{\binom{n}{9}}$	$N_{\binom{n}{10}}$
1:	2	1	1									
2:	3	1	1	1								
3:	4	1	1	1	1							
4:	6	1	1	2	1	1						
5:	8	1	1	2	2	1	1					
6:	14	1	1	3	4	3	1	1				
7:	20	1	1	3	5	5	3	1	1			
8:	36	1	1	4	7	10	7	4	1	1		
9:	60	1	1	4	10	14	14	10	4	1	1	
10:	108	1	1	5	12	22	26	22	12	5	1	1
11:	188	1	1	5	15	30	42	42	30	15	5	1
12:	352	1	1	6	19	43	66	80	66	43	19	6
13:	632	1	1	6	22	55	99	132	132	99	55	22
14:	1182	1	1	7	26	73	143	217	246	217	143	73
15:	2192	1	1	7	31	91	201	335	429	429	335	201
16:	4116	1	1	8	35	116	273	504	715	810	715	504
17:	7712	1	1	8	40	140	364	728	1144	1430	1430	1144
18:	14602	1	1	9	46	172	476	1038	1768	2438	2704	2438
19:	27596	1	1	9	51	204	612	1428	2652	3978	4862	4862
20:	52488	1	1	10	57	245	776	1944	3876	6310	8398	9252

Figure 17.3-A: The number of binary necklaces with a prescribed number of ones.

n:	L_n	$L_{\binom{n}{0}}$	$L_{\binom{n}{1}}$	$L_{\binom{n}{2}}$	$L_{\binom{n}{3}}$	$L_{\binom{n}{4}}$	$L_{\binom{n}{5}}$	$L_{\binom{n}{6}}$	$L_{\binom{n}{7}}$	$L_{\binom{n}{8}}$	$L_{\binom{n}{9}}$	$L_{\binom{n}{10}}$
1:	2	1	1									
2:	1	0	1	0								
3:	2	0	1	1	0							
4:	3	0	1	1	1	0						
5:	6	0	1	2	2	1	0					
6:	9	0	1	2	3	2	1	0				
7:	18	0	1	3	5	5	3	1	0			
8:	30	0	1	3	7	8	7	3	1	0		
9:	56	0	1	4	9	14	14	9	4	1	0	
10:	99	0	1	4	12	20	25	20	12	4	1	0
11:	186	0	1	5	15	30	42	42	30	15	5	1
12:	335	0	1	5	18	40	66	75	66	40	18	5
13:	630	0	1	6	22	55	99	132	132	99	55	22
14:	1161	0	1	6	26	70	143	212	245	212	143	70
15:	2182	0	1	7	30	91	200	333	429	429	333	200
16:	4080	0	1	7	35	112	273	497	715	800	715	497
17:	7710	0	1	8	40	140	364	728	1144	1430	1430	1144
18:	14532	0	1	8	45	168	476	1026	1768	2424	2700	2424
19:	27594	0	1	9	51	204	612	1428	2652	3978	4862	4862
20:	52377	0	1	9	57	240	775	1932	3876	6288	8398	9225

Figure 17.3-B: The number of binary Lyndon words with a prescribed number of ones.

Let $L_{\binom{n}{n_0}}$ be the number of binary length-n Lyndon words with exactly n_0 ones (*Lyndon words with fixed density*), then

$$L_{\binom{n}{n_0}} = \frac{1}{n}\sum_{j\backslash g}\mu(j)\binom{n/j}{d/j} \tag{17.3-2}$$

where $g=\gcd(n,n_0)$. The symmetry relation is the same as for $N_{\binom{n}{n_0}}$. A table of small values is given in figure 17.3-B.

Let $N(n_0,n_1,\ldots,n_{k-1})$ be the number of k-symbol length-n necklaces with n_j occurrences of symbol j. For the number of such *necklaces with fixed content* we have ($n=\sum_{j<s}n_j$ and):

$$N(n_0,n_1,\ldots,n_{k-1}) = \frac{1}{n}\sum_{d\backslash g}\varphi(d)\,\frac{(n/d)!}{(n_0/d)!\cdots(n_{k-1}/d)!} \tag{17.3-3}$$

where $g=\gcd(n_0,n_1,\ldots,n_{k-1})$. The equivalent formula for the *Lyndon words with fixed content* is

$$L(n_0,n_1,\ldots,n_{k-1}) = \frac{1}{n}\sum_{d\backslash g}\mu(d)\,\frac{(n/d)!}{(n_0/d)!\cdots(n_{k-1}/d)!} \tag{17.3-4}$$

where $g=\gcd(n_0,n_1,\ldots,n_{k-1})$. The relations were taken from [202] and [208] which also give efficient algorithms for the generation of necklaces and Lyndon words with fixed density and content, respectively. The number of strings with fixed content is a *multinomial coefficient*, see relation 11.2-1a on page 276.

Chapter 18

Hadamard and conference matrices

The $2^k \times 2^k$-matrices corresponding to the Walsh transforms (see chapter 22 on page 429) are special cases of so-called Hadamard matrices. Such matrices also exist for certain sizes $n \times n$ for n not a power of two. We give construction schemes for Hadamard matrices that come from the theory of finite fields.

If we denote the transform matrix for an N-point Walsh transform by $\mathbf{H}$, then

$$\mathbf{H}\,\mathbf{H}^T \quad = \quad N\ \mathrm{id} \tag{18.0-1}$$

where id is the unit matrix. The matrix $\mathbf{H}$ is orthogonal (up to normalization) and its determinant equals

$$\det(\mathbf{H}) \quad = \quad \det\left(\mathbf{H}\,\mathbf{H}^T\right)^{1/2} \ = \ N^{N/2} \tag{18.0-2}$$

Further, all entries are either $+1$ or -1. An orthogonal matrix with these properties is called a *Hadamard matrix*. We know that for $N = 2^n$ we always can find such a matrix. For $N = 2$ we have

$$\mathbf{H}_2 \quad = \quad \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \tag{18.0-3}$$

and we can use the Kronecker product (see section 22.3 on page 433) to construct $\mathbf{H}_{2N}$ from $\mathbf{H}_N$ via

$$\mathbf{H}_n \quad = \quad \begin{bmatrix} +\mathbf{H}_{N/2} & +\mathbf{H}_{N/2} \\ +\mathbf{H}_{N/2} & -\mathbf{H}_{N/2} \end{bmatrix} \ = \ \mathbf{H}_2 \otimes \mathbf{H}_{N/2} \tag{18.0-4}$$

The problem of determining Hadamard matrices (especially for N not a power of two) comes from combinatorics. Hadamard matrices of size $N \times N$ can only exist if N equals 1, 2, or $4\,k$.

18.1 Hadamard matrices via LFSR

We start with a construction for certain Hadamard matrices for N a power of two that uses m-sequences that are created by shift registers (see section 39.1 on page 833 and section 39.4 on page 838). Figure 18.1-A shows three Hadamard matrices that were constructed as follows:

1. Choose $N = 2^n$ and create a maximum length binary shift register sequence S of length $N - 1$.
2. Make S signed, that is, replace all ones by -1 and all zeros by $+1$.
3. The $N \times N$ matrix $\mathbf{H}$ is obtained by filling the first row and the first column with ones and filling the remaining entries with cyclical copies of s: for $r = 1, 2, \ldots N-1$ and $c = 1, 2, \ldots N-1$ set $\mathbf{H}_{r,c} := S_{c-r+1 \bmod N-1}$.

```
Signed SRS:
   - + + + - + + - - + - + - - -
Hadamard matrix H:
 + + + + + + + + + + + + + + + +
 + - + + + - + + - - + - + - - -
 + - - + + + - + + - - + - + - -
 + - - - + + + - + + - - + - + -
 + - - - - + + + - + + - - + - +
 + + - - - - + + + - + + - - + -
 + - + - - - - + + + - + + - - +
 + + - + - - - - + + + - + + - -
 + - + - + - - - - + + + - + + -
 + - - + - + - - - - + + + - + +
 + + - - + - + - - - - + + + - +
 + + + - - + - + - - - - + + + -
 + - + + - - + - + - - - - + + +
 + + - + + - - + - + - - - - + +
 + + + - + + - - + - + - - - - +
 + + + + - + + - - + - + - - - -

Signed SRS:
   - + + - + - -
Hadamard matrix H:
 + + + + + + + +
 + - + + + - + -
 + - - + + - + -
 + - - - + + - +
 + + - - - + + -
 + - + - - - + +
 + + - + - - - +
 + + + - + - - -

Signed SRS:
   - + -
Hadamard matrix H:
 + + + +
 + - + -
 + - - +
 + + - -
```

Figure 18.1-A: Hadamard matrices created with binary shift register sequences (SRS) of maximum length. Only the sign of the entries is given, all entries are ± 1.

For given n we can obtain as many different Hadamard matrices as there are m-sequences.

The matrices in figure 18.1-A where produced with the program [FXT: comb/hadamard-srs-demo.cc].

```
#include "bpol/lfsr.h" // class lfsr
#include "aux1/copy.h"  // copy_cyclic()

#include "matrix/matrix.h"  // class matrix
typedef matrix<int>  Smat;  // matrix with integer entries

  [--snip--]
    ulong n = 5;
    ulong N = 1UL << n;
  [--snip--]

    // --- create signed SRS:
    int vec[N-1];
    lfsr S(n);
    for (ulong k=0; k<N-1; ++k)
    {
        ulong x = 1UL & S.get_a();
        vec[k] = ( x ? -1 : +1 );
        S.next();
    }

    // --- create Hadamard matrix:
    Smat H(N,N);
    for (c=0; c<N; ++c)  H.set(0, c, +1);  // first row = [1,1,1,...,1]
    for (ulong r=1; r<N; ++r)
    {
        H.set(r, 0, +1);  // first column = [1,1,1,...,1]^T
        copy_cyclic(vec, H.rowp_[r]+1, N-1, N-r);
    }
  [--snip--]
```

The function `copy_cyclic()` is defined in [FXT: aux1/copy.h]:

```
template <typename Type>
inline void copy_cyclic(const Type *src, Type *dst, ulong n, ulong s)
// Copy array src[] to dst[]
// starting from position s in src[]
// wrap around end of src[]  (src[n-1])
//
// src[] is assumed to be of length n
// dst[] must be length n at least
//
// Equivalent to:  { copy(src, dst, n); rotate_right(dst, n, s)}
{
    ulong k = 0;
    while ( s<n )  dst[k++] = src[s++];

    s = 0;
    while ( k<n )  dst[k++] = src[s++];
}
```

If we define the matrix $\mathbf{X}$ to be the $(N-1) \times (N-1)$ block of $\mathbf{H}$ obtained by deleting the first row and column then

$$\mathbf{X}\,\mathbf{X}^T \quad = \quad \begin{bmatrix} N-1 & -1 & -1 & \cdots & -1 \\ -1 & N-1 & -1 & \cdots & -1 \\ -1 & -1 & N-1 & \cdots & -1 \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & N-1 \end{bmatrix} \tag{18.1-1}$$

Equivalently, for the (cyclic) auto correlation of S (see section 39.5 on page 839):

$$\sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \quad = \quad \begin{cases} +L & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{18.1-2}$$

where $L = N - 1$ is the length of the sequence.

A alternative way to obtain Hadamard matrices of dimension 2^n is to use the signs in the multiplication table for hypercomplex numbers described in section 37.16 on page 787.

18.2 Hadamard matrices via conference matrices

```
Quadratic characters modulo 13:
   0 + - + + - - - - + + - +
 14x14 conference matrix C:
  0 + + + + + + + + + + + +
  + 0 + - + + - - - - + + - +
  + + 0 + - + + - - - - + + -
  + - + 0 + - + + - - - - + +
  + + - + 0 + - + + - - - - +
  + + + - + 0 + - + + - - -
  + - + + - + 0 + - + + - -
  + - - + + - + 0 + - + + -
  + - - - + + - + 0 + - + +
  + - - - - + + - + 0 + - +
  + + - - - - + + - + 0 + -
  + - + + - - - - + + - + 0
  + + - + + - - - - + + - +
```

```
Quadratic characters modulo 11:
   0 + - + + + - - - + -
12x12 conference matrix C:
  0 + + + + + + + + + + +
  - 0 + - + + + - - - + -
  - - 0 + - + + + - - - +
  - + - 0 + - + + + - - -
  - - + - 0 + - + + + - -
  - - - + - 0 + - + + + -
  - - - - + - 0 + - + + +
  - + - - - + - 0 + - + +
  - + + - - - + - 0 + - +
  - + + + - - - + - 0 + -
  - - + + + - - - + - 0 +
  - + - + + + - - - + - 0
```

Figure 18.2-A: Two Conference matrices, the entries not on the diagonal are ± 1 and only the sign is given. The left is a symmetric 14×14 matrix ($13 \equiv 1 \bmod 4$), the right is a antisymmetric 12×12 matrix ($11 \equiv 3 \bmod 4$). Replacing all diagonal elements of the right matrix with $+1$ gives a 12×12 Hadamard matrix.

A *conference matrix* $\mathbf{C}_Q$ is a $Q \times Q$ matrix with zero diagonal and all other entries ± 1 so that

$$\mathbf{C}\,\mathbf{C}^T \quad = \quad (Q-1)\ \mathrm{id} \tag{18.2-1}$$

An algorithm for the construction of $\mathbf{C}_q$ for $Q = q + 1$ where q is an odd prime:

1. Create a length-q array S with entries $S_k \in \{-1,\, 0,\, +1\}$ as follows: set $S_0 = 0$ and, for $1 \le k < q$ set $S_k = +1$ if k is a square modulo q, $S_k = -1$ else.
2. Set $y = 1$ if $q \equiv 1 \bmod 4$, else $y = -1$ (then $q \equiv 3 \bmod 4$).
3. Create a $Q \times Q$ matrix $\mathbf{C}$ as follows: set $\mathbf{C}_{0,0} = 0$ and $\mathbf{C}_{0,k} = +1$ for $1 \le k < Q$ (first row). Set $\mathbf{C}_{k,0} = y$ for $1 \le k < Q$ (first column). Fill the remaining entries with cyclical copies of S: for $1 \le r < q$ and $1 \le c < q$ set $\mathbf{C}_{r,c} = S_{c-r+1 \bmod q}$.

The quantity y tells us whether $\mathbf{C}$ is symmetric ($y = +1$) or antisymmetric ($y = -1$).

```
12x12 Hadamard matrix H:
 + + + + + + - + + + + +
 + + + - - + + - + - - +
 + + + + - - + + - + - -
 + - + + + - + - + - + -
 + - - + + + + - - + - +
 + + - - + + + + - - + -
 - + + + + + - - - - - -
 + - + - - + - - - + + -
 + + - + - - - - - - + +
 + - + - + - - + - - - +
 + - - + - + - + + - - -
 + + - - + - - - + + - -

Quadratic characters modulo 5:
   0 + - - +
6x6 conference matrix C:
 0 + + + + +
 + 0 + - - +
 + + 0 + - -
 + - + 0 + -
 + - - + 0 +
 + + - - + 0
```

Figure 18.2-B: A 12×12 Hadamard matrix (left) created from the symmetric 6×6 conference matrix on the right.

If $\mathbf{C}_Q$ is antisymmetric then $H_Q = \mathbf{C}_Q + \mathrm{id}$ is a Hadamard matrix. For example, replacing all zeros in the 12×12 matrix in figure 18.2-A by $+1$ gives a 12×12 Hadamard matrix..

If $\mathbf{C}_Q$ is symmetric then a $2Q \times 2Q$ Hadamard matrix is given by

$$\mathbf{H}_{2Q} \quad := \quad \begin{bmatrix} +\mathrm{id} + \mathbf{C} & -\mathrm{id} + \mathbf{C} \\ -\mathrm{id} + \mathbf{C} & -\mathrm{id} - \mathbf{C} \end{bmatrix} \tag{18.2-2}$$

Figure 18.2-B shows a 12×12 Hadamard matrix that was created using this formula.

The program [FXT: comb/conference-quadres-demo.cc] outputs for a given q the $Q \times Q$ conference matrix and the corresponding Hadamard matrix:

```
#include "mod/numtheory.h" // kronecker()
#include "matrix/matrix.h"  // class matrix
#include "aux1/copy.h"  // copy_cyclic()

  [--snip--]
    int y = ( 1==q%4 ? +1 : -1 );
    ulong Q = q+1;
  [--snip--]
    // --- create table of quadratic characters modulo q:
    int vec[q];  fill<int>(vec, q, -1);  vec[0] = 0;
    for (ulong k=1; k<(q+1)/2; ++k)  vec[(k*k)%q] = +1;
  [--snip--]
    // --- create Q x Q conference matrix:
    Smat C(Q,Q);
    C.set(0,0, 0);
    for (ulong c=1; c<Q; ++c)  C.set(0, c, +1);  // first row = [1,1,1,...,1]
    for (ulong r=1; r<Q; ++r)
    {
        C.set(r, 0, y);  // first column = +-[1,1,1,...,1]^T
        copy_cyclic(vec, C.rowp_[r]+1, q, Q-r);
    }
  [--snip--]
    // --- create a N x N Hadamard matrix:
    ulong N = ( y<0 ? Q : 2*Q );
    Smat H(N,N);
    if ( N==Q )
    {
        copy(C, H);
        H.diag_add_val(1);
    }
    else
    {
        Smat K2(2,2);  K2.fill(+1); K2.set(1,1, -1);  // K2 = [+1,+1; +1,-1]
        H.kronecker(K2, C);  // Kronecker product of matrices
        for (ulong k=0; k<Q; ++k)  // adjust diagonal of submatrices
        {
            ulong r, c;
            r=k;   c=k;   H.set(r,c, H.get(r,c)+1);
            r=k;   c=k+Q; H.set(r,c, H.get(r,c)-1);
            r=k+Q; c=k;   H.set(r,c, H.get(r,c)-1);
            r=k+Q; c=k+Q; H.set(r,c, H.get(r,c)-1);
        }
```

```
  }
[--snip--]
```

If both $\mathbf{H}_a$ and $\mathbf{H}_b$ are Hadamard matrices (of dimensions a and b, respectively) then their Kronecker product $\mathbf{H}_{ab} = \mathbf{H}_a \otimes \mathbf{H}_b$ is again a Hadamard matrix:

$$\mathbf{H}_{ab}\,\mathbf{H}_{ab}^T \;=\; (\mathbf{H}_a \otimes \mathbf{H}_b)\,(\mathbf{H}_a \otimes \mathbf{H}_b)^T \;=^*\; (\mathbf{H}_a \otimes \mathbf{H}_b)\,\left(\mathbf{H}_a^T \otimes \mathbf{H}_b^T\right) \;= \tag{18.2-3a}$$

(the starred equality uses relation 22.3-11a on page 434)

$$= \;\left(\mathbf{H}_a\,\mathbf{H}_a^T\right) \otimes \left(\mathbf{H}_b \otimes \mathbf{H}_b^T\right) \;=^*\; (a\ \mathrm{id}) \otimes (b\ \mathrm{id}) \;=\; a\,b\ \mathrm{id} \tag{18.2-3b}$$

(the starred equality uses relation 22.3-10a on page 434).

18.3 Conference matrices via finite fields

The algorithm for odd primes q can be modified to work also for powers of odd primes. Then one has to work with the finite fields $\mathrm{GF}(q^n)$. The entries $\mathbf{C}_{r+1,c+1}$ for $r = 0, 1, \ldots, q^n - 1$ and $c = 0, 1, \ldots, q^n - 1$ have to be the quadratic character of $z_r - z_c$ where $z_0, z_1, \ldots, z_{q^n-1}$ are the elements in $\mathrm{GF}(q^n)$ in some (fixed) order.

We implement the algorithm in pari/gp. Firstly, we give two simple routines that map the elements $z_i \in \mathrm{GF}(q^n)$ (represented as polynomials modulo q) to the numbers $0, 1, \ldots, q^n - 1$. The polynomial $p(x) = c_0 + c_1\,x + \ldots + c_{n-1}\,x^{n-1}$ is mapped to $N = c_0 + c_1\,q + \ldots + c_{n-1}\,q^{n-1}$.

```
pol2num(p,q)=
\\ Return number for polynomial p.
{
    local(t, n);  n=0;
    for (k=0, poldegree(p),
        t = polcoeff(p, k);
        t *= Mod(1,q);
        t = component(t, 2);
        t *= q^k;
        n += t;
    );
    return( n );
}
```

The inverse routine is

```
num2pol(n,q)=
\\ Return polynomial for number n.
{
    local(p, mq, k);
    p = Pol(0,'x);
    k = 0;
    while ( 0!=n,
        mq = n % q;
        p += mq * ('x)^k;
        n -= mq;
        n \= q;
        k++;
    );
    return( p );
}
```

The quadratic character of an element z can be determined by computing $z^{(q^n-1)/2}$ modulo the field polynomial. The result will be zero for $z = 0$, else ± 1. The following routine determines the character of the difference of two elements as required for the computation of conference matrices:

```
getquadchar_n(n1, n2, q, fp, n)=
\\ Return the quadratic character of (n2-n1) in GF(q^n)
\\ Powering method
{
    local(p1, p2, d, nd, sc);
    if ( n1==n2, return(0) );
```

```
    p1 = num2pol(n1, q);
    p2 = num2pol(n2, q);
    d = Mod(1,q)* (p2-p1);
    d = Mod(d,fp)^((q^n-1)/2);
    d = component(d, 2);
    if ( Mod(1,q)==d,  sc=+1, sc=-1 );
    return( sc );
}
```

The input are two numbers that are mapped to the corresponding field elements.

In order to reduce the computational work we create a table of the quadratic characters for later lookup:

```
quadcharvec(fp, q)=
\\ Return a table of quadratic characters in GF(q^n)
\\ fp is the field polynomial.
{
    local(n, qn, sv, pl);
    n=poldegree(fp);
    qn=q^n-1;
    sv=vector(qn+1, j, -1);
    sv[1] = 0;
    for (k=1, qn,
        pl = num2pol(k,q);
        pl = Mod(Mod(1,q)*pl, fp);
        sq = pl * pl;
        sq = component(sq, 2);
        i = pol2num( sq, q );
        sv[i+1] = +1;
    );
    return( sv );
}
```

Using this table we can compute the quadratic characters of the difference of two elements in a more efficient manner:

```
getquadchar_v(n1, n2, q, fp, sv)=
\\ Return the quadratic character of (n2-n1) in GF(q^n)
\\ Table lookup method
{
    local(p1, p2, d, nd, sc);
    if ( n1==n2, return(0) );
    p1 = num2pol(n1, q);
    p2 = num2pol(n2, q);
    d = (p2-p1) % fp;
    nd = pol2num(d, q);
    sc = sv[nd+1];
    return( sc );
}
```

Now we can compute conference matrices:

```
matconference(q, fp, sv)=
\\ Return a QxQ conference matrix.
\\ q  an odd prime.
\\ fp an irreducible polynomial modulo q.
\\ sv table of quadratic characters in GF(q^n)
\\   where n is the degree of fp.
{
    local(y, Q, C, n);
    n = poldegree(fp);
    Q=q^n+1;
    C = matrix(Q,Q);
    for (k=2, Q, C[1,k]=+1);  \\ first row
    for (k=2, Q, C[k,1]=y);  \\ first column
    for (r=2, Q,
        for (c=2, Q,
            sc = getquadchar_n(r-2, c-2, q, fp, n);
\\            sc = getquadchar_v(r-2, c-2, q, fp, sv);  \\ same result
            C[r,c] = sc;
        );
    );
    return( C );
}
```

```
q = 3   fp = x^2 + 1   GF(3^2)
Table of quadratic characters:
   0 + + + - - + - -
10x10 conference matrix C:
 0 + + + + + + + + +
 - 0 + + + - - + - -
 - + 0 + - + - - + -
 - + + 0 - - + - - +
 - + - - 0 + + + - -
 - - + - + 0 + - + -
 - - - + + + 0 - - +
 - + - - + - - 0 + +
 - - + - - + - + 0 +
 - - - + - - + + + 0
```

Figure 18.3-A: A 10×10 conference matrix for $q = 3$ and the field polynomial $f = x^2 + 1$.

```
q = 3   fp = x^3 - x + 1   GF(3^3)
Table of quadratic characters:
   0 + - - - - + + + + - + + + - + + - - - + - + - - + -
28x28 conference matrix C:
 0 + + + + + + + + + + + + + + + + + + + + + + + + + + +
 - 0 + - - - - + + + + - + + + - + + - - - + - + - - + -
 - - 0 + - - - + + + + + - - + + - + + + - - - - + - - +
 - + - 0 - - - + + + - + + + - + + - + - + - + - - + - -
 - + + + 0 + - - - - + + - + - + + + - - + - - - + - + -
 - + + + - 0 + - - - - + + + + - - + + - - + + - - - - +
 - + + + + - 0 - - - + - + - + + + - + + - - - + - + - -
 - - - - + + + 0 + - + + - + + - + - + - + - - + - - - +
 - - - - + + + - 0 + - + + - + + + + - - - + - - + + - -
 - - - - + + + + - 0 + - + + - + - + + + - - + - - - + -
 - - - + - + - - + - 0 + - - - - + + + + - + + + - + + -
 - + - - - - + - - + - 0 + - - - + + + + + - - + + - + +
 - - + - + - - + - - + - 0 - - - + + + - + + + - + + - +
 - - + - - - + - + - + + + 0 + - - - - + + - + - + + + -
 - - - + + - - - - + + + + - 0 + - - - - + + + + - - + +
 - + - - - + - + - - + + + + - 0 - - - + - + - + + + - +
 - - + - - + - - - + - - - + + + 0 + - + + - + + - + - +
 - - - + - - + + - - - - - + + + - 0 + - + + - + + + + -
 - + - - + - - - + - - - - + + + + - 0 + - + + - + - + +
 - + - + + + - + + - - - + - + - - + - 0 + - - - - + + +
 - + + - - + + - + + + - - - - + - - + - 0 + - - - + + +
 - - + + + - + + - + - + - + - - + - - + - 0 - - - + + +
 - + + - + - + + + - - + - - - + - + - + + + 0 + - - - -
 - - + + + + - - + + - - + + - - - - + + + + - 0 + - - -
 - + - + - + + + - + + - - - + - + - - + + + + - 0 - - -
 - + + - + + - + - + - + - - + - - - + - - - + + + 0 + -
 - - + + - + + + + - - - + - - + + - - - - - + + + - 0 +
 - + - + + - + - + + + - - + - - - + - - - - + + + + - 0
```

Figure 18.3-B: A 28×28 conference matrix for $q = 3$ and the field polynomial $f = x^3 - x + 1$.

To compute a $Q \times Q$ conference matrix where $Q = q^n$ we need to find an polynomial of degree n that is irreducible modulo q. With $q = 3$ and the field polynomial $f = x^2 + 1$ we obtain the 10×10 conference matrix shown in figure 18.3-A. A conference matrix for $q = 3$ and $f = x^3 - x + 1$ is given in figure 18.3-B. Hadamard matrices can be created in the same manner as before, the symmetry criterion being whether $q^n \equiv \pm 1 \bmod 4$.

The construction of Hadamard matrices via conference matrices is due to R. E. A. C. Paley. The conference matrices obtained are of size $c = q^n + 1$ where q is an odd prime. The values $c \leq 100$ are:

```
4, 6, 8, 10, 12, 14, 18, 20, 24, 26, 28, 30, 32, 38, 42, 44, 48,
50, 54, 60, 62, 68, 72, 74, 80, 82, 84, 90, 98
```

We do not obtain conference matrices for any odd c and these even values $c \leq 100$:

```
2, 16, 22, 34, 36, 40, 46, 52, 56, 58, 64, 66, 70, 76, 78, 86, 88, 92, 94, 96, 100
```

For example, $c = 16 = 15 + 1 = 3 \cdot 5 + 1$ has not the required form.

If a conference matrix of size c exists then we can create Hadamard matrices of sizes $N = c$ whenever $q^n \equiv 3 \bmod 4$, and $N = 2\,c$ whenever $q^n \equiv 1 \bmod 4$. Further, if Hadamard matrices of sizes N and M exist then a $(N \cdot M) \times (N \cdot M)$ Hadamard matrix can be obtained via the Kronecker product.

The values of $N = 4\,k \leq 2000$ such that this construction does *not* give a $N \times N$ Hadamard matrix are:

```
92, 116, 156, 172, 184, 188, 232, 236, 260, 268, 292, 324, 356, 372,
376, 404, 412, 428, 436, 452, 472, 476, 508, 520, 532, 536, 584,
596, 604, 612, 652, 668, 712, 716, 732, 756, 764, 772, 808, 836,
852, 856, 872, 876, 892, 904, 932, 940, 944, 952, 956, 964, 980,
988, 996, 1004, 1012, 1016, 1028, 1036, 1068, 1072, 1076, 1100,
1108, 1132, 1148, 1168, 1180, 1192, 1196, 1208, 1212, 1220, 1244,
1268, 1276, 1300, 1316, 1336, 1340, 1364, 1372, 1380, 1388, 1396,
1412, 1432, 1436, 1444, 1464, 1476, 1492, 1508, 1528, 1556, 1564,
1588, 1604, 1612, 1616, 1636, 1652, 1672, 1676, 1692, 1704, 1712,
1732, 1740, 1744, 1752, 1772, 1780, 1796, 1804, 1808, 1820, 1828,
1836, 1844, 1852, 1864, 1888, 1892, 1900, 1912, 1916, 1928, 1940,
1948, 1960, 1964, 1972, 1976, 1992
```

This is sequence A046116 of [214]. It can be obtained by starting with a list of all numbers of the form $4\,k$ and deleting all values $k = 2^a\,(q + 1)$ where q is a power of an odd prime. Constructions for Hadamard matrices for numbers of certain forms are known, see [170]. Whether Hadamard matrices exist for all values $N = 4\,k$ is an open problem. A readable source about constructions for Hadamard matrices is [217].

Chapter 19

Searching paths in directed graphs

We describe how certain combinatorial structures can be represented as paths or cycles in a directed graph. As an example consider Gray codes of n-bit binary words: we are looking for sequences of all 2^n binary words such that only one bit changes between two successive words. A convenient representation of the search space is that of a graph. The nodes are the binary words and an edge is drawn between two nodes if the node's values differ by exactly one bit. Every path that visits all nodes of that graph corresponds to a Gray code. If the path is a cycle then a Gray cycle was found.

In general we can, depending on the size of the problem,

1. try to find at least one object
2. generate all objects
3. show that no such object exists

The method used is usually called *backtracking*. We will see how to reduce the search space if additional constraints are imposed on the paths. Finally, we show how careful optimization can lead to surprising algorithms for objects of a size where one would hardly expect to obtain a result at all. In fact, Gray cycles through the n-bit binary Lyndon words for all odd $n \leq 37$ are determined.

Terminology and conventions

We will use the terms *node* (instead of *vertex*) and *edge* (sometimes called *arc*). We restrict our attention to *directed graphs* (or *digraphs*) as undirected graphs are just the special case of these: an edge in an undirected graph corresponds to two antiparallel edges (think: 'arrows') in a directed graph.

A length-k *path* is a sequence of nodes where an edge leads from each node to its successor. A path is called *simple* if the nodes are pairwise distinct. We restrict our attention to simple paths of length N where N is the number of nodes of in the graph. We use the term *full path* for a simple path of length N.

If in a simple path there is an edge from the last node of the path to the starting node the path is a *cycle* (or *circuit*). A full path that is a cycle is called a *Hamiltonian cycle*, a graph containing such a cycle is called *Hamiltonian*.

We allow for *loops* (edges that start and point to the same node). Graphs that contain loops are called *pseudo graphs*. The algorithms used will effectively ignore loops. We disallow *multigraphs* (where multiple edges can start and end at the same two nodes), as these would lead to repeated output of identical objects.

The *neighbors* of a node are those nodes where outgoing edges point to. Neighbors can be reached with one step. The neighbors of a node a called *adjacent* to the node. The *adjacency matrix* of a graph with N nodes is a $N \times N$ matrix A where $A_{i,j} = 1$ if there is an edge from node i to node j, else $A_{i,j} = 0$.

While trivial to implement (and later modify) we will not use this kind of representation as the memory requirement would be prohibitive for large graphs.

19.1 Representation of digraphs

For our purposes a static implementation of the graph as arrays of nodes and (outgoing) edges will suffice. The container class `digraph` merely allocates memory for the nodes and edges. The correct initialization is left to the user [FXT: `class digraph` in graph/digraph.h]:

```
class digraph
{
public:
    ulong ng_;   // number of Nodes of Graph
    ulong *ep_;  // e[ep[k]], ..., e[ep[k+1]-1]: outgoing connections of node k
    ulong *e_;   // outgoing connections (Edges)
    ulong *vn_;  // optional: sorted values for nodes
    // if vn is used, then node k must correspond to vn[k]
public:
    digraph(ulong ng, ulong ne, ulong *&ep, ulong *&e, bool vnq=false)
        : ng_(0), ep_(0), e_(0), vn_(0)
    {
        ng_ = ng;
        ep_ = new ulong[ng_+1];
        e_ = new ulong[ne];
        ep = ep_;
        e = e_;
        if ( vnq )  vn_ = new ulong[ng_];
    }

    ~digraph()
    {
        delete [] ep_;
        delete [] e_;
        if ( vn_ )  delete [] vn_;
    }

 [--snip--]

    void get_edge_idx(ulong p, ulong &fe, ulong &en)  const
    // Setup fe and en so that the nodes reachable from p are
    //   e[fe], e[fe+1], ..., e[en-1].
    // Must have:  0<=p<ng
    {
        fe = ep_[p];   // (index of) First Edge
        en = ep_[p+1]; // (index of) first Edge of Next node
    }

 [--snip--]
    void print(const char *bla=0)  const;
};
```

The nodes reachable from node p could be listed using

```
    // ulong p; // == position
    cout << "The nodes reachable from node " << p << " are:" << endl;
    ulong fe, en;
    g_.get_edge_idx(p, fe, en);
    for (ulong ep=fe; ep<en; ++ep)  cout << e_[ep] << endl;
```

Using our representation there is no cheap method to find the incoming edges. We will not need this information for our purposes. If the graph is known to be undirected, the same routine obviously lists the incoming edges.

Initialization routines for certain digraphs are declared in [FXT: graph/mk-special-digraphs.h]. A simple example is [FXT: `make_complete_digraph()` in graph/mk-complete-digraph.cc]:

```
digraph
make_complete_digraph(ulong n)
// Initialization for the complete graph.
{
    ulong ng = n, ne = n*(n-1);
```

```
    ulong *ep, *e;
    digraph dg(ng, ne, ep, e);
    ulong j = 0;
    for (ulong k=0; k<ng; ++k)  // for all nodes
    {
        ep[k] = j;
        for (ulong i=0; i<n; ++i)  // connect to all nodes
        {
            if ( k==i )  continue;  // skip loops
            e[j++] = i;
        }
    }
    ep[ng] = j;
    return  dg;
}
```

We initialize the *complete graph* (the undirected graph that has edges between any two of its nodes) for $n = 5$ and print it [FXT: graph/graph-perm-demo.cc]:

```
    digraph dg = make_complete_digraph(5);
    dg.print("Graph =");
```

The output is

```
Graph =
Node: Edge0 Edge1 ...
  0:    1   2   3   4
  1:    0   2   3   4
  2:    0   1   3   4
  3:    0   1   2   4
  4:    0   1   2   3
  #nodes=5  #edges=20
```

For many purposes it suffices to implicitly represent the nodes as values p with $0 \le p < N$ where N is the number of nodes. If not, the values of the nodes have to be stored in the array `vn_[]`. One such example is a graph where the value of node p is the p-th (cyclically minimal) Lyndon word that we will meet at the end of this chapter. To make the search for a node by value reasonably fast the array `vn_[]` should be sorted so that binary search can be used.

19.2 Searching full paths

In order to search full paths starting from some position p_0 we need two additional arrays for the bookkeeping: A record `rv_[]` of the path so far, its k-th entry shall be p_k, the node visited at step k. Further a tag array `qq_[]` that shall contain a zero for nodes not visited so far, else one. The crucial parts of the implementation are [FXT: `class digraph_paths` in graph/digraph-paths.h]:

```
class digraph_paths
// Find all full paths in a directed graph.
{
public:
    digraph &g_;  // the graph
    ulong *rv_;  // Record of Visits: rv[k] == node visited at step k
    ulong *qq_;  // qq[k] == whether node k has been visited yet
 [--snip--]
    // function to call with each path found with all_paths():
    ulong (*pfunc_)(digraph_paths &);
 [--snip--]
    // function to impose condition with all_cond_paths():
    bool (*cfunc_)(digraph_paths &, ulong ns);
public:
    // graph/digraph.cc:
    digraph_paths(digraph &g);
    ~digraph_paths();
 [--snip--]
    bool path_is_cycle()  const;
 [--snip--]
    void print_path() const;
 [--snip--]
```

```
    // graph/digraphpaths-search.cc:
    ulong all_paths(ulong (*pfunc)(digraph_paths &),
                    ulong ns=0, ulong p=0, ulong maxnp=0);
private:
    void next_path(ulong ns, ulong p);  // called by all_paths()
 [--snip--]
};
```

We could have used a bit-array for the tag values `qq_[]`. It turns out that some additional information can be saved there as we will see in a moment.

To keep matters simple a recursive algorithm is used to search for (full) paths. The search is started via call to `all_paths()` [FXT: graph/digraph-paths.cc]:

```
ulong
digraph_paths::all_paths(ulong (*pfunc)(const digraph_paths &),
                         ulong ns/*=0*/, ulong p/*=0*/, ulong maxnp/*=0*/)
// pfunc: function to visit (process) paths
// ns: start at node index ns (for fixing start of path)
// p: start at node value p   (for fixing start of path)
// maxnp: stop if maxnp paths were found
{
    pct_ = 0;
    cct_ = 0;
    pfct_ = 0;
    pfunc_ = pfunc;
    pfdone_ = 0;
    maxnp_ = maxnp;
    next_path(ns, p);
    return pfct_;  // Number of paths where pfunc() returned true
}
```

The search is done by the function `next_path()`:

```
void
digraph_paths::next_path(ulong ns, ulong p)
// ns+1 == how many nodes seen
// p == position (node we are on)
{
    if ( pfdone_ )  return;

    rv_[ns] = p;  // record position
    ++ns;

    if ( ns==ng_ )  // all nodes seen ?
    {
        pfunc_(*this);
    }
    else
    {
        qq_[p] = 1; // mark position as seen (else loops lead to errors)
        ulong fe, en;
        g_.get_edge_idx(p, fe, en);
        ulong fct = 0;  // count free reachable nodes  // FCT
        for (ulong ep=fe; ep<en; ++ep)
        {
            ulong t = g_.e_[ep]; // next node
            if ( 0==qq_[t] )  // node free?
            {
                ++fct;
                qq_[p] = fct; // mark position as seen: record turns  // FCT
                next_path(ns, t);
            }
        }
        // if ( 0==fct )  { "dead end: this is a U-turn"; }  // FCT

        qq_[p] = 0; // unmark position
    }
}
```

The lines that are commented with `// FCT` record which among the free nodes is visited. The algorithm still works if these lines are commented out.

19.2.1 Paths in the complete graph: permutations

The program [FXT: graph/graph-perm-demo.cc] shows the paths in the complete graph from section 19.1, page 357. The version here is slightly simplified:

```
ulong pfunc_perm(digraph_paths &dp)
// Function to be called with each path:
//  print all but the first node.
{
    const ulong *rv = dp.rv_;
    ulong ng = dp.ng_;

    cout << setw(4) << dp.pfct_ << ":  ";
    for (ulong k=1; k<ng; ++k)  cout << " " << rv[k];
    cout << endl;

    return 1;
}

int
main(int argc, char **argv)
{
    ulong n = 5;
    digraph dg = make_complete_digraph(n);
    digraph_paths dp(dg);

    dg.print("Graph =");
    cout << endl;

    dp.all_paths(pfunc_perm, 0, 0, maxnp);
    return 0;
}
```

The output is [FXT: graph/graph-perm-out.txt]:

```
    Graph =
    Node: Edge0 Edge1 ...
      0:    1   2   3   4
      1:    0   2   3   4
      2:    0   1   3   4
      3:    0   1   2   4
      4:    0   1   2   3
      #nodes=5  #edges=20

       0:   1 2 3 4
       1:   1 2 4 3
       2:   1 3 2 4
       3:   1 3 4 2
       4:   1 4 2 3
       5:   1 4 3 2
       6:   2 1 3 4
       7:   2 1 4 3
       8:   2 3 1 4
       9:   2 3 4 1
      10:   2 4 1 3
      11:   2 4 3 1
      12:   3 1 2 4
      13:   3 1 4 2
      14:   3 2 1 4
      15:   3 2 4 1
      16:   3 4 1 2
      17:   3 4 2 1
      18:   4 1 2 3
      19:   4 1 3 2
      20:   4 2 1 3
      21:   4 2 3 1
      22:   4 3 1 2
      23:   4 3 2 1
```

These are the permutations of the numbers $1, 2, 3, 4$ in lexicographic order (see section 10.1 on page 219).

19.2.2 Paths in the De Bruijn graph: De Bruijn sequences

The graph with $2\,n$ nodes and two outgoing edges from node k to $2\,k \bmod 2\,n$ and $2\,k+1 \bmod 2\,n$ is called a *De Bruijn graph*. For $n = 8$ the graph is (printed horizontally):

```
    Node:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    Edge 0:  0  2  4  6  8 10 12 14  0  2  4  6  8 10 12 14
    Edge 1:  1  3  5  7  9 11 13 15  1  3  5  7  9 11 13 15
```

The graph has two loops at the first and the last node. All paths in the De Bruijn graph are cycles, the graph is Hamiltonian.

With n a power of two the paths correspond to the *De Bruijn sequences* (DBS) of length $2\,n$. The graph has as many full paths as there are DBSs and the zeros/ones in the DBS correspond to even/odd values of the nodes, respectively. This is demonstrated in [FXT: graph/graph-debruijn-demo.cc] (shortened):

```
ulong pq = 1; // whether and what to print with each cycle

ulong pfunc_db(digraph_paths &dp)
// Function to be called with each cycle.
{
    switch ( pq )
    {
    case 0:  break;  // just count
    case 1:  // print lowest bits (De Bruijn sequence)
        {
            ulong *rv = dp.rv_, ng = dp.ng_;
            for (ulong k=0; k<ng; ++k)  cout << (rv[k]&1UL ? '1' : '.');
            cout << endl;
            break;
        }
    [--snip--]
    }
    return 1;
}

int main(int argc, char **argv)
{
    ulong n = 8;
    NXARG(pq, "what to do in pfunc()");
    ulong maxnp = 0;
    NXARG(maxnp, "stop after maxnp paths (0: never stop)");
    ulong p0 = 0;
    NXARG(p0, "start position <2*n");

    digraph dg = make_debruijn_digraph(n);
    digraph_paths dp(dg);

    dg.print_horiz("Graph =");

    // call pfunc() with each cycle:
    dp.all_paths(pfunc_db, 0, p0, maxnp);

    cout << "n = " << n;
    cout << " (ng=" << dg.ng_ << ")";
    cout << "   #cycles = " << dp.cct_;
    cout << endl;

    return 0;
}
```

The macro `NXARG()` read one argument, it is defined in [FXT: nextarg.h]. The output is

```
    arg 1: 8 == n  [size of graph == 2*n]  default=8
    arg 2: 1 == pq  [what to do in pfunc()]  default=1
    arg 3: 0 == maxnp  [stop after maxnp paths (0: never stop)]  default=0
    arg 4: 0 == p0  [start position <2*n]  default=0
    Graph =
      Node:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
    Edge 0:  0  2  4  6  8 10 12 14  0  2  4  6  8 10 12 14
    Edge 1:  1  3  5  7  9 11 13 15  1  3  5  7  9 11 13 15
    .1..11.1.1111...
    .1..1111.1.11...
    .1.1..11.1111...
    .1.1..1111.11...
    .1.11..1111.1...
    .1.11.1..1111...
    .1.1111..11.1...
    .1.1111.1..11...
    .11..1.1111.1...
    .11.1..1.1111...
    .11.1.1111..1...
    .11.1111..1.1...
    .1111..1.11.1...
    .1111.1..1.11...
    .1111.1.11..1...
    .1111.11..1.1...
    n = 8 (ng=16)   #cycles = 16
```

The algorithm is a very effective way of generating all DBSs of a given length, the 67,108,864 DBSs of length 64 are generated in 140 seconds when printing is disabled (set argument `pq` to zero), corresponding to a rate of more than 450,000 DBSs per second.

Setting the argument pq to 4 prints the binary values of the successive nodes in the path horizontally:

```
./bin 32 4 1
    -#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######-----
    --#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######----
    ---#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######---
    ----#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######--
    -----#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######-
    ------#----##---#-#---###--#--#-##--##-#--####-#-#-###-##-######
```

The graph is constructed in a way that each word is the predecessor shifted by one with either zero or one inserted at position zero (top row).

The number of cycles in the De Bruijn graph equals the number of degree-n normal binary polynomials. An efficient procedure to compute it is given in section 40.6.3 on page 871. A closed form for the special case $n = 2^k$ is given in section 39.4 on page 838.

19.2.3 A modified De Bruijn graph: complement-shift sequences

A modification of the De Bruijn graph forces the nodes to be the complement of its predecessor shifted by one (again with either zero or one inserted at position zero). The routine to set up the graph is [FXT: make_complement_shift_digraph() in graph/mk-debruijn-digraph.cc]:

```
digraph
make_complement_shift_digraph(ulong n)
{
    ulong ng = 2*n, ne = 2*ng;
    ulong *ep, *e;
    digraph dg(ng, ne, ep, e);

    ulong j = 0;
    for (ulong k=0; k<ng; ++k)  // for all nodes
    {
        ep[k] = j;
        ulong r = (2*k) % ng;
        e[j++] = r;  // connect node k to node (2*k) mod ng
        r = (2*k+1) % ng;
        e[j++] = r;  // connect node k to node (2*k+1) mod ng
    }
    ep[ng] = j;
    // Here we have a De Bruijn graph.

    for (ulong k=0,j=ng-1; k<j; ++k,--j) swap2(e[ep[k]], e[ep[j]]); // end with ones
    for (ulong k=0,j=ng-1; k<j; ++k,--j) swap2(e[ep[k]+1], e[ep[j]+1]);

    return  dg;
}
```

The program [FXT: graph/graph-complementshift-demo.cc] gives:

```
    arg 1: 32 == n  [size of graph == 2*n]  default=32
    arg 2: 4 == pq  [whether and what to print with each path]  default=2
    arg 3: 1 == maxnp  [stop after maxnp paths (0: never stop)]  default=1

    ------#---#-#-##----##--#--#-###---###-#--##-####--######-##-#-#
    -######-###-#-#--####--##-##-#---###---#-##--#----##------#--#-#
    -#------#---#-#-##----##--#--#-###---###-#--##-####--######-##-#
    -#-######-###-#-#--####--##-##-#---###---#-##--#----##------#--#
    -#-#------#---#-#-##----##--#--#-###---###-#--##-####--######-##
    -#-#-######-###-#-#--####--##-##-#---###---#-##--#----##------#-
```

For n a power of two the sequence of binary words has the interesting property that the changes between successive words depend on their sequency: words with higher sequency change in less positions. Further, if two neighbor bits are set in some word then the next word never contains both bits again. Out of a run of $k > 0$ consecutive set bits in a word only one is contained in the next word.

See section 8.3 on page 197 for the connection with De Bruijn sequences.

19.3 Conditional search

Sometimes one wants to find paths that are subject to certain restrictions. Testing for each path found whether it has the desired property and discarding it if not is the most simple way. However, this will in many cases be extremely ineffective. An upper bound for the number of recursive calls of the search function `next_path()` with an graph with N nodes and an maximal number of v outgoing edges at each node is $u = N^v$.

For example, the graph corresponding to Gray codes of n-bit binary words has $N = 2^n$ nodes and (exactly) $c = n$ outgoing edges at each node. The graph is the n-dimensional hypercube.

n :	N	$u = N^c = N^n = 2^{n \cdot n}$
1:	2	2
2:	4	16
3:	8	512
4:	16	65,536
5:	32	33,554,432
6:	64	68,719,476,736
7:	128	562,949,953,421,312
8:	256	18,446,744,073,709,551,616
9:	512	2,417,851,639,229,258,349,412,352
10:	1024	1,267,650,600,228,229,401,496,703,205,376

We are obviously interested in the reduction of the size of the search space. This can in many cases be achieved by a function that rejects branches that would lead to a path not satisfying the imposed restrictions.

A conditional search can be started via `all_cond_paths()` that has an additional function pointer `cfunc()` as argument that shall implement the condition. It is declared as

```
bool (*cfunc_)(digraph_paths &, ulong ns);
```

Besides the data from the digraph-class it needs the number of nodes seen so far (`ns`) as an argument. A slight modification of the search routine `next_path()` does what we want:

```
void
digraph_paths::next_cond_path(ulong ns, ulong p)
{
    [--snip--]  // same as next_path()
    if ( ns==ng_ )  // all nodes seen ?
    [--snip--]  // same as next_path()
    else
    {
        qq_[p] = 1; // mark position as seen (else loops lead to errors)
        ulong fe, en;
        g_.get_edge_idx(p, fe, en);
        ulong fct = 0;  // count free reachable nodes
        for (ulong ep=fe; ep<en; ++ep)
        {
            ulong t = g_.e_[ep]; // next node
            if ( 0==qq_[t] )  // node free?
            {
                rv_[ns] = t;  // for cfunc()
                if ( cfunc_(*this, ns) )
                {
                    ++fct;
                    qq_[p] = fct; // mark position as seen: record turns
                    next_cond_path(ns, t);
                }
            }
        }
        qq_[p] = 0; // unmark position
    }
}
```

The free node under consideration is written to the end of the record of visited nodes so `cfunc()` does

not need it as an explicit argument.

19.3.1 Modular adjacent changes (MAC) Gray codes

```
 0:  .... 0    0    ...1  0          0:  .... 0    0    ...1  0
 1:  ...1 1    1    ..1.  1          1:  ...1 1    1    ..1.  1
 2:  ..11 2    3    .1..  2          2:  ..11 2    3    .1..  2
 3:  .111 3    7    1...  3          3:  .111 3    7    ..1.  1
 4:  1111 4   15    ...1  0          4:  .1.1 2    5    ...1  0
 5:  111. 3   14    ..1.  1          5:  .1.. 1    4    ..1.  1
 6:  11.. 2   12    ...1  0          6:  .11. 2    6    .1..  2
 7:  11.1 3   13    1...  3          7:  ..1. 1    2    1...  3
 8:  .1.1 2    5    ...1  0          8:  1.1. 2   10    .1..  2
 9:  .1.. 1    4    ..1.  1          9:  111. 3   14    ..1.  1
10:  .11. 2    6    .1..  2         10:  11.. 2   12    ...1  0
11:  ..1. 1    2    1...  3         11:  11.1 3   13    ..1.  1
12:  1.1. 2   10    ...1  0         12:  1111 4   15    .1..  2
13:  1.11 3   11    ..1.  1         13:  1.11 3   11    ..1.  1
14:  1..1 2    9    ...1  0         14:  1..1 2    9    ...1  0
15:  1... 1    8   [1...  3]       15:  1... 1    8   [1...  3]
```

Figure 19.3-A: Two four-bit modular adjacent changes (MAC) Gray codes. Both are cycles.

We search for Gray codes that have the *modular adjacent changes* (MAC) property: the values of successive elements of the delta sequence shall change by ± 1 modulo n. Two examples are show in figure 19.3-A. The sequence on the right side even has the stated property if the term 'modular' is omitted: It has the *adjacent changes* (AC) property.

As bit-wise cyclic shifts and reflections of MAC Gray codes are again MAC Gray codes we consider paths starting $0 \to 1 \to 2$ as canonical paths.

In the demo [FXT: graph/graph-macgray-demo.cc] the search is done as follows (shortened):

```
int main(int argc, char **argv)
{
    ulong n = 5;
    NXARG(n, "size in bits");
    cf_nb = n;

    digraph dg = make_gray_digraph(n, 0);
    digraph_paths dp(dg);

    ulong ns = 0, p = 0;
    // MAC: canonical paths start as 0-->1-->3
    {
        dp.mark(0, ns);
        dp.mark(1, ns);
        p = 3;
    }

    dp.all_cond_paths(pfunc, cfunc_mac, ns, p, maxnp);
    return 0;
}
```

The function used to impose the MAC condition is:

```
ulong cf_nb; // number of bits,  set in main()
bool cfunc_mac(digraph_paths &dp, ulong ns)
// Condition: difference of successive delta values (modulo n) == +-1
{
    // path initialized, we have ns>=2
    ulong p  = dp.rv_[ns],  p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
    ulong c = p ^ p1,  c1 = p1 ^ p2;
    if ( c & bit_rotate_left(c1,1,cf_nb) )  return  true;
    if ( c1 & bit_rotate_left(c,1,cf_nb) )  return  true;
    return false;
}
```

One finds paths for $n \leq 7$ ($n = 7$ takes about 15 minutes). Whether MAC Gray codes exist for $n \geq 8$ is unknown (none is found with a 40 hour search).

19.3.2 Adjacent changes (AC) Gray codes

For AC paths one can only discard track-reflected solutions, the canonical paths are those where the delta sequence starts with a values that is $\leq \lceil n/2 \rceil$. A function to impose the AC condition is

```
ulong cf_mt; // mid track < cf_mt,  set in main()
bool cfunc_ac(digraph_paths &dp, ulong ns)
// Condition: difference of successive delta values == +-1
{
    if ( ns<2 )  return  (dp.rv_[1] < cf_mt); // avoid track-reflected solutions
    ulong p  = dp.rv_[ns],  p1 = dp.rv_[ns-1], p2 = dp.rv_[ns-2];
    ulong c = p ^ p1, c1 = p1 ^ p2;
    if ( c & (c1<<1) )  return  true;
    if ( c1 & (c<<1) )  return  true;
    return false;
}
```

```
 0:  ..... 0    0    ..1..  2          0:  ..... 0    0    ....1  0
 1:  ..1.. 1    4    .1...  3          1:  ....1 1    1    ...1.  1
 2:  .11.. 2   12    1....  4          2:  ...11 2    3    ..1..  2
 3:  111.. 3   28    .1...  3          3:  ..111 3    7    .1...  3
 4:  1.1.. 2   20    ..1..  2          4:  .1111 4   15    ..1..  2
 5:  1.... 1   16    ...1.  1          5:  .1.11 3   11    ...1.  1
 6:  1..1. 2   18    ..1..  2          6:  .1..1 2    9    ..1..  2
 7:  1.11. 3   22    .1...  3          7:  .11.1 3   13    .1...  3
 8:  1111. 4   30    ..1..  2          8:  ..1.1 2    5    1....  4
 9:  11.1. 3   26    ...1.  1          9:  1.1.1 3   21    .1...  3
10:  11... 2   24    ....1  0         10:  111.1 4   29    ..1..  2
11:  11..1 3   25    ...1.  1         11:  11..1 3   25    ...1.  1
12:  11.11 4   27    ..1..  2         12:  11.11 4   27    ..1..  2
13:  11111 5   31    .1...  3         13:  11111 5   31    .1...  3
14:  1.111 4   23    ..1..  2         14:  1.111 4   23    ..1..  2
15:  1..11 3   19    ...1.  1         15:  1..11 3   19    ...1.  1
16:  1...1 2   17    ..1..  2         16:  1...1 2   17    ....1  0
17:  1.1.1 3   21    .1...  3         17:  1.... 1   16    ...1.  1
18:  111.1 4   29    1....  4         18:  1..1. 2   18    ..1..  2
19:  .11.1 3   13    .1...  3         19:  1.11. 3   22    .1...  3
20:  ..1.1 2    5    ..1..  2         20:  1111. 4   30    ..1..  2
21:  ....1 1    1    ...1.  1         21:  11.1. 3   26    ...1.  1
22:  ...11 2    3    ..1..  2         22:  11... 2   24    ..1..  2
23:  ..111 3    7    .1...  3         23:  111.. 3   28    .1...  3
24:  .1111 4   15    ..1..  2         24:  1.1.. 2   20    1....  4
25:  .1.11 3   11    ...1.  1         25:  ..1.. 1    4    .1...  3
26:  .1..1 2    9    ....1  0         26:  .11.. 2   12    ..1..  2
27:  .1... 1    8    ...1.  1         27:  .1... 1    8    ...1.  1
28:  .1.1. 2   10    ..1..  2         28:  .1.1. 2   10    ..1..  2
29:  .111. 3   14    .1...  3         29:  .111. 3   14    .1...  3
30:  ..11. 2    6    ..1..  2         30:  ..11. 2    6    ..1..  2
31:  ...1. 1    2   [...1.  1]       31:  ...1. 1    2   [...1.  1]
```

Figure 19.3-B: Two five-bit adjacent changes (AC) Gray codes that are cycles.

The program [FXT: graph/graph-acgray-demo.cc] allows searches for AC Gray codes. Two cycles for $n = 5$ are shown in figure 19.3-B. It turns out that such paths exist for $n \leq 6$ (the only path for $n = 6$ is shown in figure 19.3-C) but there is no AC Gray code for $n = 7$:

```
 time ./bin 7
arg 1: 7 == n  [size in bits]  default=5
arg 2: 0 == maxnp  [ stop after maxnp paths (0: never stop)]  default=0
n = 7   #pfct = 0
   #paths = 0   #cycles = 0
./bin 7  20.77s user 0.11s system 98% cpu 21.232 total
```

Nothing is known about the case $n \geq 8$.

Inspection of the AC Gray codes for different values of n result in a hand-woven algorithm. The function [FXT: `ac_gray_delta()` in comb/acgray.cc] computes the delta sequence for an AC Gray codes for $n \leq 6$:

```
void
ac_gray_delta(uchar *d, ulong ldn)
// Generate a delta sequence for an adjacent-changes (AC) Gray code
//  of length n=2**ldn where ldn<=6.
{
    if ( ldn<=2 )  // standard Gray code
    {
        d[0] = 0;
```

```
     0:  ...... 0    0    ...1..  2        32:  1.1... 2   40    .1....  4
     1:  ...1.. 1    4    ....1.  1        33:  111... 3   56    ..1...  3
     2:  ...11. 2    6    ...1..  2        34:  11.... 2   48    ...1..  2
     3:  ....1. 1    2    ..1...  3        35:  11.1.. 3   52    ....1.  1
     4:  ..1.1. 2   10    ...1..  2        36:  11.11. 4   54    ...1..  2
     5:  ..111. 3   14    ....1.  1        37:  11..1. 3   50    ..1...  3
     6:  ..11.. 2   12    .....1  0        38:  111.1. 4   58    ...1..  2
     7:  ..11.1 3   13    ....1.  1        39:  11111. 5   62    ....1.  1
     8:  ..1111 4   15    ...1..  2        40:  1111.. 4   60    .....1  0
     9:  ..1.11 3   11    ..1...  3        41:  1111.1 5   61    ....1.  1
    10:  ....11 2    3    ...1..  2        42:  111111 6   63    ...1..  2
    11:  ...111 3    7    ....1.  1        43:  111.11 5   59    ..1...  3
    12:  ...1.1 2    5    ...1..  2        44:  11..11 4   51    ...1..  2
    13:  .....1 1    1    ..1...  3        45:  11.111 5   55    ....1.  1
    14:  ..1..1 2    9    .1....  4        46:  11.1.1 4   53    ...1..  2
    15:  .11..1 3   25    ..1...  3        47:  11...1 3   49    ..1...  3
    16:  .1...1 2   17    ...1..  2        48:  111..1 4   57    .1....  4
    17:  .1.1.1 3   21    ....1.  1        49:  1.1..1 3   41    ..1...  3
    18:  .1.111 4   23    ...1..  2        50:  1....1 2   33    ...1..  2
    19:  .1..11 3   19    ..1...  3        51:  1..1.1 3   37    ....1.  1
    20:  .11.11 4   27    ...1..  2        52:  1..111 4   39    ...1..  2
    21:  .11111 5   31    ....1.  1        53:  1...11 3   35    ..1...  3
    22:  .111.1 4   29    .....1  0        54:  1.1.11 4   43    ...1..  2
    23:  .111.. 3   28    ....1.  1        55:  1.1111 5   47    ....1.  1
    24:  .1111. 4   30    ...1..  2        56:  1.11.1 4   45    .....1  0
    25:  .11.1. 3   26    ..1...  3        57:  1.11.. 3   44    ...1.   1
    26:  .1..1. 2   18    ...1..  2        58:  1.111. 4   46    ...1..  2
    27:  .1.11. 3   22    ....1.  1        59:  1.1.1. 3   42    ..1...  3
    28:  .1.1.. 2   20    ...1..  2        60:  1...1. 2   34    ...1..  2
    29:  .1.... 1   16    ..1...  3        61:  1..11. 3   38    ....1.  1
    30:  .11... 2   24    .1....  4        62:  1..1.. 2   36    ...1..  2
    31:  ..1... 1    8    1.....  5        63:  1..... 1   32   [1.....  5]
```

Figure 19.3-C: The (essentially unique) AC Gray code for $n = 6$. While the path is a cycle in the graph the AC condition does not hold for the transition from the last to the first word. No AC Gray code for $n = 7$ exists.

```
        if ( ldn==2 ) { d[1] = 1; d[2] = 0; }
        return;
    }

    ac_gray_delta(d, ldn-1);  // recursion

    ulong n = 1UL<<ldn;
    ulong nh = n/2;
    if ( 0==(ldn&1) )
    {
        if ( ldn>=6 )
        {
            reverse(d, nh-1);
            for (ulong k=0;  k<nh;  ++k)  d[k] = (ldn-2) - d[k];
        }

        for (ulong k=0,j=n-2;  k<j;  ++k,--j)  d[j] = d[k];
        d[nh-1] = ldn - 1;
    }
    else
    {
        for (ulong k=nh-2,j=nh-1;  0!=j;  --k,--j)  d[j] = d[k] + 1;
        for (ulong k=2,j=n-2;  k<j;  ++k,--j)  d[j] = d[k];
        d[0] = 0;
        d[nh] = 0;
    }
}
```

The Gray code is computed via

```
void
ac_gray(ulong *g, ulong ldn)
// Create an AC Gray code.
{
    ulong n = 1UL<<ldn;
    ALLOCA(uchar, d, n);
    ac_gray_delta(d, ldn);
    delta2gray(d, ldn, g);
}
```

where the routine `delta2gray()` is given in [FXT: comb/delta2gray.cc]:

```
void
delta2gray(const unsigned char *d, ulong ldn, ulong *g, ulong g0/*=0*/)
// Fill into g[0..N-1] (N=2**ldn) the Gray path
//  corresponding to the delta sequence d[0..N-2].
{
    g[0] = g0;
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n-1; ++k)  g[k+1] = g[k] ^ (1UL << d[k]);
}
```

The program [FXT: comb/acgray-demo.cc] can be used to create AC Gray codes for $n \le 6$. For $n \ge 7$ the algorithm produces near-AC Gray codes, where the number of non-AC transitions equals $2^{n-5}-1$ for odd values of n and $2^{n-5}-2$ for n even:

```
 # non-AC transitions:
   n =0..6  #non-ac = 0
   n =  7   #non-ac = 3
   n =  8   #non-ac = 6
   n =  9   #non-ac = 15
   n = 10   #non-ac = 30
   n = 11   #non-ac = 63
   n = 12   #non-ac = 126
   ...
```

It seems likely that near-AC Gray codes with fewer non-AC transitions exist.

19.4 Edge sorting and lucky paths

The order of the nodes in the representation of the graph does not matter with finding paths as the algorithm does at no point refer to it. The order of the outgoing edges, however, *does* matter.

Edge sorting

Consider a large graph that has only a few paths. The calling tree of the recursive function `next_path()` obviously depends on the edge order. Thereby the first path can appear earlier or later in the search. 'Later' may well mean that the path is not found within any reasonable amount of time.

With a bit of luck one might find an ordering of the edges of the graph that will shorten the time span until the first path is found. The program [FXT: graph/graph-monotonicgray-demo.cc] searches for monotonic Gray codes and optionally sorts the edges of the graph. The method [FXT: `digraph::sort_edges()` in graph/digraph.cc] sorts the outgoing edges of each node according to a supplied comparison function.

The comparison function actually used imposes the lexicographic order shown in section 1.27 on page 64:

```
int my_cmp(const ulong &a, const ulong &b)
{
    if ( a==b )  return  0;
#define CODE(x) lexrev2negidx(x);
    ulong ca = CODE(a);
    ulong cb = CODE(b);
    return (ca<cb ? +1 : -1);
}
```

The choice was inspired by the observation that the bit-wise difference of successive elements in bit-lex order is either one or three. We search until the first path for 8-bit words is found: for the unsorted graph this task takes 1.14 seconds, for the sorted it takes 0.03 seconds.

Lucky paths

The first Gray code found in the hypercube graph with randomized edge order is shown in figure 19.4-A (left). The corresponding path (as reported by [FXT: `digraph_paths::print_turns()` in graph/digraph-paths.cc]) is described in the right column. Here `nn` is the number of neighbors of `node`, `xe` is the index

```
 0:  .... 0    0    1...  3          step: node -> next [xf xe / nn]
 1:  1... 1    8    ..1.  1             0:    0 ->   8  [ 0  0 /  4]
 2:  1.1. 2   10    .1..  2             1:    8 ->  10  [ 0  0 /  4]
 3:  111. 3   14    ...1  0             2:   10 ->  14  [ 0  0 /  4]
 4:  1111 4   15    1...  3             3:   14 ->  15  [ 0  0 /  4]
 5:  .111 3    7    .1..  2             4:   15 ->   7  [ 0  0 /  4]
 6:  ..11 2    3    ...1  0             5:    7 ->   3  [ 0  1 /  4]
 7:  ..1. 1    2    .1..  2             6:    3 ->   2  [ 1  2 /  4]
 8:  .11. 2    6    ..1.  1             7:    2 ->   6  [ 0  3 /  4]
 9:  .1.. 1    4    1...  3             8:    6 ->   4  [ 0  0 /  4]
10:  11.. 2   12    ...1  0             9:    4 ->  12  [ 1  3 /  4]
11:  11.1 3   13    1...  3            10:   12 ->  13  [ 0  0 /  4]
12:  .1.1 2    5    .1..  2            11:   13 ->   5  [ 0  1 /  4]
13:  ...1 1    1    1...  3            12:    5 ->   1  [ 0  3 /  4]
14:  1..1 2    9    ..1.  1            13:    1 ->   9  [ 0  2 /  4]
15:  1.11 3   11   [1.11  -]           14:    9 ->  11  [ 0  0 /  4]
                                    Path: #non-first-free turns = 2
```

Figure 19.4-A: A Gray code in the hypercube graph with randomized edge order (left) and the path description (right, see text).

of the neighbor (`next`) in the list of edges of `node`. Finally `xf` is the index among the *free* nodes in the list. The latter corresponds to the value `fct-1` in the function `next_path()` given on page 358.

If `xf` equals zero at some step then the first free neighbor was visited next. If `xf` is nonzero then a dead end was reached in the course of the search and there was at least one U-turn. If the path is not the first found then the U-turn might well correspond to a previous path.

If there was no U-turn then the number of *non-first-free* turns is zero (the number is given as the last line of the report). If it is zero we call the path found a *lucky path*. For each given ordering of the edges and each starting position of the search there is at most one lucky path and if there is, it is the first path found.

When the first path is a lucky path then the search effectively 'falls through': the number of operations is a constant times the number of edges. That is, if a lucky path exists it is found almost immediately even for huge graphs.

19.5 Gray codes for Lyndon words

We search Gray codes for n-bit binary Lyndon words where n is a prime. Here is a Gray code for the 5-bit Lyndon words that is a cycle:

```
...1
..11
.111
.1111
..111
..1.1
```

An important application of such Gray codes is the construction of so-called *single track Gray codes* which can be obtained by appending rotated versions of the block. The following is a single track Gray code based on the block given. At each stage, the block is rotated by two positions (horizontal format):

```
######  --##--  -####-  ------  ---###
-####-  ------  ---###  ######  --##--
---###  ######  --##--  -####-  ------
--##--  -####-  ------  ---###  ######
------  ---###  ######  --##--  -####-
```

The *transition count* (the number of zero-one transitions) is by construction the same for each track. The all-zero and the all-one words are missing in the Gray code, its length equals $2^n - 2$.

```
 0:   .....1    .....1    .....1    .....1    .....1    .....1
 1:   ....11    ...1..1   ...1..1   ...1..1   ...1..1   ....11
 2:   ...111    ...11.1   ...11.1   ...11.1   ...1.11   ....111
 3:   ...1111   ..111.1   ..111.1   ....1.1   .1.1.11   ....1.1
 4:   ..11111   ..1.1.1   ..11111   ..1.1.1   .1.1111   ..1.1.1
 5:   ..11.11   ..1.111   .111111   ..111.1   .111111   ..111.1
 6:   ..1..11   .11.111   .11.111   ..11111   .11.111   ...11.1
 7:   ..1.111   .111111   ..1.111   ..11.11   ..1.111   ...1..1
 8:   .11.111   .1.1111   ..1.1.1   ..1..11   ..1.1.1   ...1.11
 9:   .111111   .1.1.11   ....1.1   ..1.111   ..111.1   ...1111
10:   .1.1111   ...1.11   ....111   .11.111   ...11.1   ..11111
11:   .1.1.11   ...1111   ...1111   .111111   ....1.1   ..11.11
12:   ...1.11   ..11111   .1.1111   .1.1111   ....111   ..1..11
13:   ...1..1   ..11.11   .1.1.11   .1.1.11   .....11   ..1.111
14:   ...11.1   ..1..11   ...1.11   ...1.11   ..1..11   .11.111
15:   ..111.1   .....11   ..11.11   ...1111   ..11.11   .111111
16:   ..1.1.1   ....111   ..1..11   ....111   ..11111   .1.1111
17:   ....1.1   ....1.1   .....11   .....11   ...1111   .1.1.11
```

Figure 19.5-A: Various Gray codes through the length-7 binary Lyndon words. The first four are cycles.

19.5.1 Graph search with edge sorting

Gray codes for the 7-bit binary Lyndon words like those shown in figure 19.5-A can easily be found by a graph search. In fact, all of them can be generated in short time: for $n = 7$ there are 395 Gray codes (starting with the word `0000..001`) of which 112 are cycles.

```
 k :  [ node]  lyn_dec lyn_bin #rot rot(lyn)     diff  delta
 0 :  [    0]        1  ......1  0   ......1   ......1  0
 1 :  [    1]        3  .....11  0   .....11   .....1.  1
 2 :  [    3]        7  ....111  0   ....111   ....1..  2
 3 :  [    7]       15  ...1111  0   ...1111   ...1...  3
 4 :  [   13]       31  ..11111  0   ..11111   ..1....  4
 5 :  [   17]       63  .111111  0   .111111   .1.....  5
 6 :  [   15]       47  .1.1111  0   .1.1111   ..1....  4
 7 :  [   10]       23  ..1.111  1   .1.111.   ......1  0
 8 :  [   16]       55  .11.111  1   11.111.   1......  6
 9 :  [   11]       27  ..11.11  2   11.11..   .....1.  1
10 :  [    5]       11  ...1.11  2   .1.11..   1......  6
11 :  [   14]       43  .1.1.11  2   .1.11.1   ......1  0
12 :  [    6]       13  ...11.1  0   ...11.1   .1.....  5
13 :  [   12]       29  ..111.1  0   ..111.1   ..1....  4
14 :  [    8]       19  ..1..11  3   ..11..1   ....1..  2
15 :  [    4]        9  ...1..1  0   ...1..1   ..1....  4
16 :  [    9]       21  ..1.1.1  3   .1.1..1   .1.....  5
17 :  [    2]        5  ....1.1  3   .1.1...   ......1  0
```

Figure 19.5-B: A Gray code through the length-7 binary Lyndon words found by searching through all 7-bit binary words.

The search for such a path for the next prime, $n = 11$, does not seem to give a result in reasonable time. If we do not insist on a Gray code through the cyclic minima but arbitrary rotations of the Lyndon words then more Gray codes exist. For that purpose nodes are declared adjacent if there is any cyclic rotation of the second node's value that differs in exactly one bit to the first node's value. The cyclic rotations can be recovered easily after a path is found. This is done in [FXT: graph/graph-lyndon-gray-demo.cc] whose output is shown in figure 19.5-B. Still, already for $n = 11$ we do not get a result. As the corresponding graph has 186 nodes and 1954 edges, this is not a surprise.

Now we try edge sorting, we sort the edges according to the comparison function [FXT: graph/lyndon-cmp.cc]

```
int lyndon_cmp0(const ulong &a, const ulong &b)
{
    int bc = bit_count_cmp(a, b);
    if ( bc )  return  -bc;  // more bits first
    else
    {
        if ( a==b )  return 0;
        return  (a>b ?  +1 : -1);  // bigger numbers last
    }
```

```
}
```

where `bit_count_cmp()` is defined in [FXT: bits/bitcount.h]:

```
static inline int bit_count_cmp(const ulong &a, const ulong &b)
{
    ulong ca = bit_count(a);
    ulong cb = bit_count(b);
    return  ( ca==cb ? 0 : (ca>cb ? +1 : -1) );
}
```

```
      k :  [ node]  lyn_dec       lyn_bin #rot      rot(lyn)           diff  delta
      0 :  [    0]         1  ...........1  0   ...........1   ...........1  0
      1 :  [    1]         3  ..........11  0   ..........11   ..........1.  1
      2 :  [    3]         7  .........111  0   .........111   .........1..  2
      3 :  [    7]        15  ........1111  0   ........1111   ........1...  3
      4 :  [   15]        31  .......11111  0   .......11111   .......1....  4
      5 :  [   31]        63  ......111111  0   ......111111   ......1.....  5
      6 :  [   63]       127  .....1111111  0   .....1111111   .....1......  6
      7 :  [  125]       255  ....11111111  0   ....11111111   ....1.......  7
      8 :  [  239]       511  ...111111111  0   ...111111111   ...1........  8
      9 :  [  417]      1023  ..1111111111  0   ..1111111111   ..1.........  9
     10 :  [  589]      2047  .11111111111  0   .11111111111   .1.......... 10
     11 :  [  629]      4095  .111111111111 0   .111111111111  .1..........  11
     12 :  [  618]      3071  .1.111111111  0   .1.111111111   ..1.........  10
     13 :  [  514]      1535  ..1.11111111  1   .1.11111111.   ...........1  0
     14 :  [  624]      3583  .11.11111111  1   11.11111111.   1...........  12
     15 :  [  550]      1791  ..11.1111111  2   11.1111111..   ..........1.  1
     16 :  [  626]      3839  .111.1111111  2   11.1111111.1   ...........1  0
     17 :  [  567]      1919  ..111.111111  3   11.111111..1   .........1..  2
     18 :  [  627]      3967  .1111.111111  3   11.111111.11   ..........1.  1
     19 :  [  576]      1983  ..1111.11111  4   11.11111..11   ........1...  3
     20 :  [  628]      4031  .11111.11111  4   11.11111.111   .........1..  2
     21 :  [  581]      2015  ..11111.1111  5   11.11111..111  .......1....  4
     22 :  [  404]       991  ...1111.1111  5   11.11111...11  .........1..  2
     23 :  [  614]      3039  .1.1111.1111  5   11.11111.1.11  .......1...   3
     24 :  [  508]      1519  ..1.1111.111  6   11.1111..1.11  ......1.....  5
     25 :  [  584]      2031  ..111111.111  6   11.1111..1111  .........1..  2
      [--snip--]
    615 :  [    4]         9  .........1..1  5   ....1..1.....  ..1.........  10
    616 :  [   36]        73  ......1..1..1  2   ....1..1..1..  .........1..  2
    617 :  [   32]        65  ......1.....1  2   ....1.....1..  .......1.....  5
    618 :  [   33]        67  ......1....11  2   ....1....11..  .........1...  3
    619 :  [  153]       323  ....1.1....11  2   ..1.1....11..  ..1.........  10
    620 :  [   65]       133  .....1....1.1  8   ..1.1.....1..  .........1...  3
    621 :  [  154]       325  ....1.1...1.1  2   ..1.1...1.1..  ........1....  4
    622 :  [   79]       161  .....1.1....1 10   ..1.....1.1..  ....1........  8
    623 :  [   16]        33  .......1....1 10   ..1.......1..  ........1....  4
    624 :  [  126]       265  ....1....1..1  2   ..1....1..1..  .......1.....  5
    625 :  [  145]       305  ....1..11...1 10   ..1....1..11.  ..........1.  1
    626 :  [  130]       273  ....1...1...1 10   ..1....1...1.  .........1..  2
    627 :  [  188]       401  ....11..1...1 10   ..1....11..1.  ........1....  4
    628 :  [   71]       145  .....1..1...1 10   ..1.....1..1.  .......1.....  5
    629 :  [    8]        17  ........1...1 10   ..1........1.  ........1....  4
```

Figure 19.5-C: Begin and end of a Gray cycle through the 13-bit binary Lyndon words.

We find a Gray code (which also is a cycle) for $n = 11$ immediately. Same for $n = 13$, again a cycle. The graph for $n = 13$ has 630 nodes and 8,056 edges, so finding a path is quite unexpected. The cycle found starts and ends as shown in figure 19.5-C.

For next candidate ($n = 17$) we do no find a Gray code within many hours of search. No surprise for a graph with 7,710 nodes and 130,828 edges.

We try another edge sorting scheme, an ordering based on the binary Gray code [FXT: graph/lyndon-cmp.cc]:

```
int lyndon_cmp2(const ulong &a, const ulong &b)
{
    if ( a==b )  return 0;
#define CODE(x) gray_code(x)
    ulong ta = CODE(a), tb = CODE(b);
    return  ( ta<tb ? +1 : -1);
}
```

There we go, we find a cycle for $n = 17$ and all smaller primes. All are cycles and all paths are lucky paths. The following edge sorting scheme also leads to Gray codes for all prime n where $3 \le n \le 17$:

```
int lyndon_cmp3(const ulong &a, const ulong &b)
{
    if ( a==b )  return 0;
#define CODE(x) inverse_gray_code(x)
    ulong ta = CODE(a), tb = CODE(b);
    return  ( ta<tb ? +1 : -1);
}
```

Same for $n = 19$, the graph has 27,594 nodes and 523,978 edges. Indeed the sorting scheme leads to cycles for all odd $n \le 27$!. All these paths are lucky paths, a fact that we can exploit for an optimized search.

19.5.2 An optimized algorithm

n	number of nodes	tag-size	time
23	364,722	0.25 MB	1 sec
25	1,342,182	1 MB	3 sec
27	4,971,066	4 MB	12 sec
29	18,512,790	16 MB	1 min
31	69,273,666	64 MB	4 min
33	260,301,174	256 MB	16 min

n	number of nodes	tag-size	time
35	981,706,830	1 GB	1 h
37	3,714,566,310	4 GB	7 h
39	14,096,303,342	16 GB	2 d
41	53,634,713,550	64 GB	10 d
43	204,560,302,842	256 GB	>40 d
45	781,874,934,568	1 TB	>160 d

Figure 19.5-D: Memory and (approximate) time needed for computing Gray codes with n-bit Lyndon words. The number of nodes equals the number of length-n necklaces minus two. The size of the tag array equals $2^n/4$ bits or $2^n/32$ bytes.

With edge sorting functions that lead to a lucky path we can discard most of the data used with graph searching. We only need to keep track of whether a node has been visited so far. A tag-array ([FXT: ds/bitarray.h], see section 4.6 on page 152) suffices.

With n-bit Lyndon words the amount of tag-bits needed is 2^n. Find an implementation of the algorithm as [FXT: `class lyndon_gray` in graph/lyndon-gray.h].

If only the cyclical minima of the values are tagged then only $2^n/2$ bits are needed if the access to the single necklace consisting of all ones is treated separately. This variant of the algorithm is activated by uncommenting the line `#define ALT_ALGORITM`. Noting that the lowest bit in a necklace is always one we need only $2^n/4$ bits: simply shift the words right by one before testing or writing to the tag array. This can be achieved by additionally uncommenting the line `#define ALTALT` in the file.

When a node is visited the algorithm creates a table of neighbors and selects the minimum among the free nodes with respect to the edge sorting function used. The table of neighbors is discarded then in order to minimize memory usage.

When no neighbor is found the number of nodes visited so far is returned. If this number equals the number of n-bit Lyndon words a lucky path was found. With composite n a Gray code for n-bit necklaces with the exception of the all-ones and the all-zeros word will be searched.

Four flavors of the algorithm have been found so far, corresponding to edge sorting with the 3rd, 5th, 21th, and 29th power of the Gray code. We refer to these functions as comparison functions 0, 1, 2, and 3, respectively. All of these lead to cycles for all primes $n \le 31$. The resources needed with larger values of n are shown in figure 19.5-D.

Using a 64-bit machine equipped with more than 4 Gigabyte of RAM it can be verified that three of the edge sorting functions lead to a Gray cycles also for $n = 37$, the 3rd power version fails. One of the

sorting functions *may* lead to a Gray code for $n = 41$.

```
% ./bin 7 2 0  # 7 bits, full output, comparison function 0
n = 7   #lyn = 18
   1:    .....1   0  .....1   .....1  0
   2:    ...1..1  0  ...1..1  ...1...  3
   3:    ..1..11  3  ..11..1  ..1....  4
   4:    ..1.111  3  .111..1  .1.....  5
   5:    .1.1111  2  .1111.1  ....1..  2
   6:    .1.1.11  2  .1.11.1  ..1....  4
   7:    .11.111  5  11.11.1  1......  6
   8:    .111111  2  11111.1  ..1....  4
   9:    ..11111  2  11111..  .....1  0
  10:    ..111.1  2  111.1..  ...1...  3
  11:    ..1.1.1  2  1.1.1..  .1.....  5
  12:    ....1.1  2  ..1.1..  1......  6
  13:    ...1.11  1  ..1.11.  .....1.  1
  14:    ..11.11  1  .11.11.  .1.....  5
  15:    ...11.1  2  .11.1..  ....1.  1
  16:    ...1111  2  .1111..  ...1...  3
  17:    ....111  2  ..111..  .1.....  5
  18:    .....11  2  ...11..  ..1....  4
last = .....11     crc=0b14a5846c41d57f
n = 7   #lyn = 18   #= 18
```

Figure 19.5-E: A Gray code for 7-bit Lyndon words.

A program to compute the Gray codes is [FXT: graph/lyndon-gray-demo.cc], four arguments can be given:

```
arg 1: 13 == n  [ a prime < BITS_PER_LONG ]  default=17
arg 2: 1 == wh  [printing: 0==>none, 1==>delta seq., 2==>full output]  default=1
arg 3: 3 == ncmp  [use comparison function (0,1,2,3)]  default=2
arg 4: 0 == testall  [special: test all odd values <= value]  default=0
```

An example with full output is given in figure 19.5-E. A 64-bit CRC (see section 1.31 on page 77) is computed from the delta sequence (rightmost column) and printed with the last word.

```
% ./bin 13 1 2  # 13 bits, delta seq. output, comparison function 2
 n = 13   #lyn = 630
 06B57458354645962546436734A74684A106C0145120825747A745247AC8564567018A7654647484A756A546457CA1ACBC1C
 856BA9A64B974565486456596452194259425215315BC82BC75BA02926256354267A462475A3ACB9761560C37412583758CA5624
 B8C6A6C6A87A9C20CBA45340420145405231290756976515631602042 30A7BA31C1485C6105201510490BCA891BA9B1B9AC0
 A9A89B898A565B878574586574784 5A9546702305A41275315458767465747A8457845470379A8586B0A76985787679767 59
 A976567686A5676 56A576B86581305A20AB0ACB0AB535234382354653252475 63A432532A372354657643572373624634642
 45323974234352356532364232632352343275323423253969268532342325826424368236323463623584232423832423 27
 523242325323432642324235323423
 last = ..........11     crc=568dab04b55aa2fb
 n = 13   #lyn = 630   #= 630


% ./bin 13 1 3  # 13 bits, delta seq. output, comparison function 3
n = 13   #lyn = 630
 06B57458354645962546436735371CA8B1587BA7610635285A0C2484B9713476B689A897AC98768968B9A106326016261050
 1424B8979A78987B97898C9892194131531369831428 1687BCB9469C489C6210205B050A1A7A4568A9BC5CB79AB647B74812
 0AB30BC1A131ACB120B0164CA1CABA121ABACA2B0BACAB184578678498958486764 6A84561916546947457875458654901 37
 4020103101210427017121650745 7B854606C16BC52380136516413016 4BC7987A09872CBA9A87A20B787AC9B7CBA834C0C1
 3C341C1042010C14C01C4145878546 45A854C95035A6A9570A9756586B9B5969580A0872C3123B0CB316BC6C0B21B2C0C2C0
 5301C0530CB1C1530C01CB0BC20CBC0CB1C87565756865A75A65A40898A898B91CA898A8B898A81BC8A9ACA989AB817A9BC1
 BA9ABA9CA9AB918A1CACBAC9BCB0BC
 last = ..........11     crc=745def277b1fbed0
 n = 13   #lyn = 630   #= 630
```

Figure 19.5-F: Delta sequences for two different Gray codes for 13-bit Lyndon words.

For larger n one might want to print only the delta sequence, this is shown in figure 19.5-F. The CRC allows us to conveniently determine whether two delta sequences are different. Different sequences sometimes start identically.

For still larger values of n even the delta sequence tends to get huge (for example, with $n = 37$ the sequence would be approximately 3.7 GB). One can suppress all output except for a progress indication, as shown in figure 19.5-G. Here the CRC checksum is updated only with every (cyclically unadjusted) 2^{16}-th Lyndon word.

Sometimes a Gray code through the necklaces (except for the all-zeros and all-ones words) is also found for composite n. Comparison functions 0, 1, and 2 lead to Gray codes (which are cycles) for all odd

```
% ./bin 29 0 0  # 29 bits, output=progress, comparison function 0
n = 29   #lyn = 18512790
.............    1048576 ( 5.66406 % )   crc=ceabc5f2056be699
.............    2097152 ( 11.3281 % )   crc=76dd94f1a554b50d
.............    3145728 ( 16.9922 % )   crc=6b39957f1e141f4d
.............    4194304 ( 22.6563 % )   crc=53419af1f1185dc0
.............    5242880 ( 28.3203 % )   crc=45d45b193f8ee566
.............    6291456 ( 33.9844 % )   crc=95a24c824f56e196
.............    7340032 ( 39.6484 % )   crc=003ee5af5b248e34
.............    8388608 ( 45.3125 % )   crc=23cb74d3ea0c4587
.............    9437184 ( 50.9766 % )   crc=896fd04c87dd0d43
.............   10485760 ( 56.6406 % )   crc=b00d8c899f0fc791
.............   11534336 ( 62.3047 % )   crc=d148f1b95b23eeab
.............   12582912 ( 67.9688 % )   crc=82971e2ed4863050
.............   13631488 ( 73.6328 % )   crc=f249ad5b4fed252d
.............   14680064 ( 79.2969 % )   crc=909821d0c7246a98
.............   15728640 ( 84.9609 % )   crc=1c5d68e38e55b3ca
.............   16777216 ( 90.625 % )    crc=0e64f82c67c79cf1
.............   17825792 ( 96.2891 % )   crc=62c17b9f3c644396
.........
last = ............................11     crc=5736fc9365da927e
n = 29   #lyn = 18512790   #= 18512790
```

Figure 19.5-G: Computation of a Gray code through the 29-bit Lyndon words. Most output is suppressed, only the CRC is printed at certain checkpoints.

$n \leq 33$. Gray cycles are also found with comparison function 3, except for $n = 21$, 27, and 33. All functions give Gray cycles also for $n = 4$ and $n = 6$. The values of n for which no Gray code could be found are the even values ≥ 8.

19.5.3 No Gray codes for even $n \geq 8$

As the parity of the words in a Gray code sequence alternates between one and zero the difference between the numbers words of odd and even weight must be zero or one. If it is ones, no Gray cycle can exist because the parity of the first and last word is identical. We use the relations from section 17.3 on page 344.

For Lyndon words of odd length there are the same number of words for odd and even weight by symmetry, see figure 17.3-B on page 345. So a Gray code (and also a Gray cycle) can exist.

For even lengths the sequence of numbers of Lyndon words of odd and even weights start as:

```
   n:   2, 4, 6,  8, 10,  12,  14,   16,   18,    20,    22,     24,      26, ...
 odd:   1, 2, 5, 16, 51, 170, 585, 2048, 7280, 26214, 95325, 349520, 1290555, ...
even:   0, 1, 4, 14, 48, 165, 576, 2032, 7252, 26163, 95232, 349350, 1290240, ...
diff:   1, 1, 1,  2,  3,   5,   9,   16,   28,    51,    93,    170,     315, ...
```

The last row gives the differences, entry A000048 of [214]. All entries for $n \geq 8$ are greater than one, so no Gray code does exist.

For the number of necklaces we obtain, for $n = 2, 4, 6, \ldots$

```
   n:   2, 4, 6,  8, 10,  12,  14,   16,   18,    20,    22,     24,      26, ...
 odd:   1, 2, 6, 16, 52, 172, 586, 2048, 7286, 26216, 95326, 349536, 1290556, ...
even:   2, 4, 8, 20, 56, 180, 596, 2068, 7316, 26272, 95420, 349716, 1290872, ...
diff:   1, 2, 2,  4,  4,   8,  10,   20,   30,    56,    94,    180,     316, ...
```

The (absolute) difference of both sequences is entry A000013 of [214]. We see that for $n \geq 4$ the numbers are greater than one, so no Gray code exists.

If we exclude the all-ones and all-zeros words, then the differences are

```
   n:   2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22,  24,  26, ...
diff:   1, 0, 0, 2,  2,  6,  8, 18, 28, 54, 92, 178, 314, ...
```

And again, no Gray code exists for $n \geq 8$. That is, we have found Gray codes, and even cycles, for all computational feasible sizes were they can exist.

Part III

Fast orthogonal transforms

Chapter 20

The Fourier transform

We introduce the discrete Fourier transform and give algorithms for its fast computation. Implementations and optimization considerations for complex and real valued transforms are given. The fast Fourier transforms (FFTs) are the basis of the algorithms for fast convolution (given in chapter 21) which are in turn the core of the fast high precision multiplication routines described in chapter 27. The number theoretic Fourier transforms (NTTs) are treated in chapter 25. Algorithms for Fourier transforms based on fast convolution are given in chapter 21, the transform for arbitrary length (Bluestein's algorithm) in section 21.5, and prime-length transforms (Rader's algorithm) in section 21.6.

20.1 The discrete Fourier transform

The *discrete Fourier transform* (DFT or simply FT) of a complex sequence $a = [a_0, a_1, \ldots, a_{n-1}]$ of length n is the complex sequence $c = [c_0, c_1, \ldots, c_{n-1}]$ defined by

$$c = \mathcal{F}\big[\, a \,\big] \tag{20.1-1a}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \, z^{+x\,k} \quad \text{where} \quad z = e^{2\,\pi\,i/n} \tag{20.1-1b}$$

z is a primitive n-th root of unity: $z^n = 1$ and $z^j \neq 1$ for $0 < j < n$.

Back-transform (or *inverse discrete Fourier transform*, IDFT or simply IFT) is then

$$a = \mathcal{F}^{-1}\big[\, c \,\big] \tag{20.1-2a}$$

$$a_x := \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, z^{-x\,k} \tag{20.1-2b}$$

To see this, consider element y of the IFT of the FT of a:

$$\mathcal{F}^{-1}\big[\, \mathcal{F}\big[\, a \,\big] \,\big]_y = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} (a_x \, z^{x\,k}) \, z^{-y\,k} \tag{20.1-3a}$$

$$= \frac{1}{n} \sum_{x} a_x \sum_{k} (z^{x-y})^k \tag{20.1-3b}$$

Now $\sum_k (z^{x-y})^k = n$ for $x = y$ and zero else. This is because z is an n-th primitive root of unity: with $x = y$ the sum consists of n times $z^0 = 1$, with $x \neq y$ the summands lie on the unit circle (on the vertices

of an equilateral polygon with center zero) and add up to zero. Therefore the whole expression is equal to

$$\frac{1}{n}\, n \sum_x a_x\, \delta_{x,y} \quad = \quad a_y \quad \text{where} \quad \delta_{x,y} := \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases} \tag{20.1-4a}$$

Here we will call the FT with the plus in the exponent the forward transform. The choice is actually arbitrary, engineers seem to prefer the minus for the forward transform, mathematicians the plus. The sign in the exponent is called the *sign of the transform*.

The FT is a linear transform. That is, for $\alpha, \beta \in \mathbb{C}$

$$\mathcal{F}\big[\,\alpha\, a + \beta\, b\,\big] \quad = \quad \alpha\, \mathcal{F}\big[\,a\,\big] + \beta\, \mathcal{F}\big[\,b\,\big] \tag{20.1-5}$$

Further *Parseval's equation* holds, the sum of squares of the absolute values is identical for a sequence and its Fourier transform:

$$\sum_{x=0}^{n-1} |a_x|^2 \quad = \quad \sum_{k=0}^{n-1} |c_k|^2 \tag{20.1-6}$$

A straightforward implementation of the discrete Fourier transform, that is, the computation of n sums each of length n, requires $\sim n^2$ operations.

```
void slow_ft(Complex *f, long n, int is)
{
    Complex h[n];
    const double ph0 = is*2.0*M_PI/n;
    for (long w=0; w<n; ++w)
    {
        Complex t = 0.0;
        for (long k=0; k<n; ++k)
        {
            t +=  f[k] * SinCos(ph0*k*w);
        }
        h[w] = t;
    }
    copy(h, f, n);
}
```

This is [FXT: `slow_ft()` in fft/slowft.cc]. The variable `is`$= \pm 1$ is the sign of the transform, the function `SinCos(x)` returns a `Complex(cos(x), sin(x))`.

Note that the normalization factor $\frac{1}{\sqrt{n}}$ in front of the FT sums has been left out. The inverse of the transform with one sign is the transform with the opposite sign followed by a multiplication of each element by $\frac{1}{n}$. One has to keep in mind if that the sum of squares of the original sequence and its transform are equal up to a factor $1/\sqrt{n}$.

A *fast Fourier transform* (FFT) algorithm is an algorithm that improves the operation count to proportional $n \sum_{k=1}^{m}(p_k - 1)$, where $n = p_1 p_2 \cdots p_m$ is a factorization of n. In case of a power $n = p^m$ the value computes to $n\,(p-1)\,\log_p(n)$. In the special case $p = 2$ even $n/2\,\log_2(n)$ (complex) multiplications suffice. There are several different FFT algorithms with many variants.

20.2 Summary of definitions of Fourier transforms *

We summarize the definitions of the continuous, semi-continuous and the discrete Fourier transform.

The continuous Fourier transform

The (continuous) *Fourier transform* (FT) of a function $f : \mathbb{C}^n \to \mathbb{C}^n, \quad \vec{x} \mapsto f(\vec{x})$ is defined by

$$F(\vec{\omega}) \quad := \quad \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{C}^n} f(\vec{x})\, e^{+\sigma\, i\, \vec{x}\, \vec{\omega}} d^n x \tag{20.2-1}$$

where $\sigma = \pm 1$ is the sign of the transform. The FT is a unitary transform. Its inverse is the complex conjugate transform

$$f(\vec{x}) \quad = \quad \frac{1}{\left(\sqrt{2\,\pi}\right)^n} \int_{\mathbb{C}^n} F(\vec{\omega})\, e^{-\sigma\, i\, \vec{x}\, \vec{\omega}} d^n \omega \tag{20.2-2}$$

For the 1-dimensional case one has

$$F(\omega) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} f(x) e^{+\sigma\, i\, x\, \omega} dx \tag{20.2-3}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^{+\infty} F(\omega)\, e^{-\sigma\, i\, x\, \omega} d\omega \tag{20.2-4}$$

The 'frequency-form' is

$$F(\nu) \quad = \quad \int_{-\infty}^{+\infty} f(x) e^{+2\,\pi\, \sigma\, i\, x\, \nu} dx \tag{20.2-5}$$

$$f(x) \quad = \quad \int_{-\infty}^{+\infty} F(\nu)\, e^{-2\,\pi\, \sigma\, i\, x\, \nu} d\nu \tag{20.2-6}$$

The semi-continuous Fourier transform

For periodic functions defined on a interval $L \in \mathbb{R}$, $f : L \to \mathbb{R},\ x \mapsto f(x)$ one defines the *semi-continuous Fourier transform* as:

$$c_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, e^{+\sigma\, 2\,\pi\, i\, k\, x/L} dx \tag{20.2-7}$$

The inverse transform is an infinite sum with the property

$$\frac{1}{\sqrt{L}} \sum_{k=-\infty}^{k=+\infty} c_k\, e^{-\sigma\, 2\,\pi\, i\, k\, x/L} \quad = \quad \begin{cases} f(x) & \text{if } f \text{ continuous at } x \\ \frac{f(x+0)+f(x-0)}{2} & \text{else} \end{cases} \tag{20.2-8}$$

An equivalent form of the semi-continuous Fourier transform is given by

$$a_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, \cos \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 0, 1, 2, \ldots \tag{20.2-9a}$$

$$b_k \quad := \quad \frac{1}{\sqrt{L}} \int_L f(x)\, \sin \frac{2\,\pi\, k\, x}{L} dx, \qquad k = 1, 2, \ldots \tag{20.2-9b}$$

$$f(x) \quad = \quad \frac{1}{\sqrt{L}} \left[\frac{a_0}{2} + \sum_{k=1}^{\infty} \left(a_k\, \cos \frac{2\,\pi\, k\, x}{L} + b_k\, \sin \frac{2\,\pi\, k\, x}{L} \right) \right] \tag{20.2-9c}$$

The connection between the two forms is given by:

$$c_k \quad = \quad \begin{cases} \frac{a_0}{2} & (k = 0) \\ \frac{1}{2}(a_k - i b_k) & (k > 0) \\ \frac{1}{2}(a_k + i b_k) & (k < 0) \end{cases} \tag{20.2-10}$$

The discrete Fourier transform

The *discrete Fourier transform* (DFT) of a sequence f of length n with elements f_x is defined by

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} f_x \, e^{+\sigma \, 2 \, \pi \, i \, x \, k / n} \tag{20.2-11}$$

The inverse transform is

$$f_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k \, e^{-\sigma \, 2 \, \pi \, i \, x \, k / n} \tag{20.2-12}$$

Some sources define $c_k := \sum \ldots$ and $f_x = \frac{1}{n} \sum \ldots$. With this definition the transform does not preserve norms but there is a computational saving when the normalization does not matter. When a transform and its inverse are applied in succession (as with convolution algorithms) then only one normalization (factors $\frac{1}{n}$) is needed instead of two (factors $\frac{1}{\sqrt{n}}$). In the implementations of the transform we will in general omit the normalization altogether and leave it to the user to apply it where needed.

20.3 Radix-2 FFT algorithms

A little bit of notation

In what follows let a be a length-n sequence with n a power of two.

- Let $a^{(even)}$ and $a^{(odd)}$ denote the length-$n/2$ subsequences of those elements of a that have even and odd indices, respectively. That is, $a^{(even)} = [a_0, a_2, a_4, a_6, \ldots, a_{n-2}]$ and $a^{(odd)} = [a_1, a_3, \ldots, a_{n-1}]$.
- Let $a^{(left)}$ and $a^{(right)}$ denote the left and right subsequences, respectively. That is, $a^{(left)} = [a_0, a_1, \ldots, a_{n/2-1}]$ and $a^{(right)} = [a_{n/2}, a_{n/2+1}, \ldots, a_{n-1}]$.
- Let $c = \mathcal{S}^k a$ denote the sequence with elements $c_x = a_x \, e^{\sigma \, 2 \, \pi \, i \, k \, x / n}$ where $\sigma = \pm 1$ is the sign of the transform. The symbol $\mathcal{S}$ shall suggest a shift operator. With radix-2 FFT algorithms only $\mathcal{S}^{1/2}$ is needed.
- In relations between sequences we sometimes emphasize the length of the sequences on both sides as in $a^{(even)} \overset{n/2}{=} b^{(odd)} + c^{(odd)}$. In these relations the operators plus and minus are to be understood as element-wise.

20.3.1 Decimation in time (DIT) FFT

The following observation is the key to the (radix-2) *decimation in time* (DIT) FFT algorithm, also called *Cooley-Tukey* FFT algorithm: For even values of n the k-th element of the Fourier transform is

$$\mathcal{F}\big[\, a \,\big]_k = \sum_{x=0}^{n-1} a_x \, z^{x \, k} = \sum_{x=0}^{n/2-1} a_{2\,x} \, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1} \, z^{(2\,x+1)\,k} \tag{20.3-1a}$$

$$= \sum_{x=0}^{n/2-1} a_{2\,x} \, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1} \, z^{2\,x\,k} \tag{20.3-1b}$$

where $z = e^{\sigma \, i \, 2 \, \pi / n}$, $\sigma = \pm 1$ is the sign of the transform, and $k \in \{0, 1, \ldots, n-1\}$.

The identity tells us how to compute the k-th element of the length-n Fourier transform from the length-$n/2$ Fourier transforms of the even and odd indexed subsequences.

To actually rewrite the length-n FT in terms of length-$n/2$ FTs one has to distinguish whether $0 \le k < n/2$ or $n/2 \le k < n$. In the expressions we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = j + \delta\,\frac{n}{2}$ where $j \in \{0, 1, 2, \ldots, n/2-1\}$ and $\delta \in \{0, 1\}$:

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(j+\delta\,\frac{n}{2})} = \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} + z^{j+\delta\,\frac{n}{2}} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} \tag{20.3-2a}$$

$$= \begin{cases} \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} + z^{j} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \displaystyle\sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,j} - z^{j} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{20.3-2b}$$

The minus sign in the relation for $\delta = 1$ is due to the fact that $z^{j+1\cdot n/2} = z^j\, z^{n/2} = -z^j$.

Observing that z^2 is just the root of unity that appears in a length-$n/2$ transform one can rewrite the last two equations to obtain the *radix-2 DIT FFT step*:

$$\mathcal{F}\big[\,a\,\big]^{(left)} \overset{n/2}{=} \mathcal{F}\big[\,a^{(even)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(odd)}\,\big] \tag{20.3-3a}$$

$$\mathcal{F}\big[\,a\,\big]^{(right)} \overset{n/2}{=} \mathcal{F}\big[\,a^{(even)}\,\big] - \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(odd)}\,\big] \tag{20.3-3b}$$

The length-n transform has been replaced by two transforms of length $n/2$. If n is a power of 2 this scheme can be applied recursively until length-one transforms are reached which are identity ('do nothing') operations.

Thereby the operation count is improved to proportional $n \cdot \log_2(n)$: there are $\log_2(n)$ splitting steps, the work in each step is proportional to n.

Note that the operator $\mathcal{S}$ depends on the sign of the transform.

20.3.1.1 Recursive implementation

A recursive implementation of radix-2 DIT FFT given as pseudo code (C++ version in [FXT: fft/recfft2.cc]) is

```
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then  // end of recursion
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1  // copy to workspace
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[], nh, b[], is)
    rec_fft_dit2(t[], nh, c[], is)

    fourier_shift(c[], nh, is*1/2)

    for k:=0 to nh-1  // copy back from workspace
    {
```

```
        x[k]    := b[k] + c[k]
        x[k+nh] := b[k] - c[k]
    }
}
```

The parameter `is` $= \sigma = \pm 1$ is the sign of the transform. The data length `n` must be a power of 2. The result is returned in the array `x[]`. Note that normalization (multiplication of each element of `x[]` by $1/\sqrt{n}$) is not included here.

The procedure uses the subroutine `fourier_shift` which modifies the array `c[]` according to the operation $\mathcal{S}^v$: each element `c[k]` is multiplied by $e^{v\,2\,\pi\,i\,k/n}$. It is called with $v = \pm 1/2$ for the Fourier transform. The pseudo code (C++ equivalent in [FXT: fft/fouriershift.cc]) is

```
procedure fourier_shift(c[], n, v)
{
    for k:=0 to n-1
    {
        c[k] := c[k] * exp(v*2.0*PI*I*k/n)
    }
}
```

The recursive FFT-procedure involves $n \log_2(n)$ function calls, which can be avoided by rewriting it in a non-recursive way. One can even do all operations *in-place*, no temporary workspace is needed at all. The price is the necessity of an additional data reordering: the procedure `revbin_permute(a[],n)` rearranges the array `a[]` in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. Methods to do this are discussed in section 2.1.

20.3.1.2 Iterative implementation

Pseudo code for a non-recursive procedure of the radix-2 DIT algorithm (C++ implementation given in [FXT: fft/fftdit2.cc]):

```
procedure fft_depth_first_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[], n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for r:=0 to n-m step m  // n/m iterations
        {
            for j:=0 to mh-1  // m/2 iterations
            {
                e := exp(is*2*PI*I*j/m)  // log_2(n)*n/m*m/2 = log_2(n)*n/2 computations

                u := a[r+j]
                v := a[r+j+mh] * e

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in-place. But it is a bit wasteful. The (expensive) computation `e := exp(is*2*PI*I*j/m)` is done $n/2 \cdot \log_2(n)$ times.

20.3.1.3 Saving trigonometric computations

To reduce the number of sine and cosine computations, one can simply swap the two inner loops, leading to the first 'real world' FFT procedure presented here. Pseudo code for a non-recursive procedure of the radix-2 DIT algorithm (C++ version in [FXT: fft/fftdit2.cc]):

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn

    revbin_permute(a[], n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for j:=0 to mh-1  // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m)  // 1 + 2 + ... + n/8 + n/4 + n/2 == n-1 computations

            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

Swapping the two inner loops reduces the number of trigonometric computations to `n` but leads to a feature that many FFT implementations share: memory access is highly nonlocal. For each recursion stage (value of `ldm`) the array is traversed `mh` times with `n/m` accesses in strides[1] of `mh`. As `mh` is a power of two this can (on computers that use memory cache) have a very negative performance impact for large values of `n`. On computers where the memory access is very slow compared to the CPU the naive version can actually be faster.

It is a good idea to extract the `ldm==1` stage of the outermost loop, this avoids complex multiplications with the trivial factors $1 + 0\,i$ (and the computations of these quantities as trigonometric functions). Replace

```
    for ldm:=1 to ldn
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
    for ldm:=2 to ldn
    {
```

20.3.2 Decimation in frequency (DIF) FFT

Splitting of the Fourier sum into a left and right half leads to the *decimation in frequency* (DIF) FFT algorithm, also called *Sande-Tukey* FFT algorithm. For even values of n the k-th element of the Fourier transform is

$$\mathcal{F}\big[\,a\,\big]_k = \sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n-1} a_x\, z^{x\,k} \tag{20.3-4a}$$

$$= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=0}^{n/2-1} a_{x+n/2}\, z^{(x+n/2)\,k} \tag{20.3-4b}$$

$$= \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{k\,n/2}\, a_x^{(right)}\right) z^{x\,k} \tag{20.3-4c}$$

[1] Stride-n memory access: consecutive accesses to addresses that are n units apart.

where $z = e^{\sigma\, i\, 2\,\pi/n}$, $\sigma = \pm 1$ is the sign of the transform, and $k \in \{0, 1, \ldots, n-1\}$.

Here one has to distinguish whether k is even or odd. Therefore we rewrite $k \in \{0, 1, 2, \ldots, n-1\}$ as $k = 2\,j + \delta$ where $j \in \{0, 1, 2, \ldots, n/2-1\}$ and $\delta \in \{0, 1\}$:

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(2\,j+\delta)} = \sum_{x=0}^{n/2-1} (a_x^{(left)} + z^{(2\,j+\delta)\,n/2}\, a_x^{(right)})\, z^{x\,(2\,j+\delta)} \tag{20.3-5a}$$

$$= \begin{cases} \sum_{x=0}^{n/2-1} (a_x^{(left)} + a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum_{x=0}^{n/2-1} z^x (a_x^{(left)} - a_x^{(right)})\, z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{20.3-5b}$$

$z^{(2\,j+\delta)\,n/2} = e^{\pm\pi\, i\, \delta}$ is equal to plus or minus one for $\delta = 0$ or $\delta = 1$ corresponding to k even or odd. The last two equations are, more compactly written, the key to the *radix-2 DIF FFT step*:

$$\mathcal{F}\big[\, a \,\big]^{(even)} \overset{n/2}{=} \mathcal{F}\big[\, a^{(left)} + a^{(right)} \,\big] \tag{20.3-6a}$$

$$\mathcal{F}\big[\, a \,\big]^{(odd)} \overset{n/2}{=} \mathcal{F}\big[\, \mathcal{S}^{1/2} \left(a^{(left)} - a^{(right)} \right) \big] \tag{20.3-6b}$$

A recursive implementation of radix-2 DIF FFT given as pseudo code (C++ version given in [FXT: fft/recfft2.cc]) is

```
procedure rec_fft_dif2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left'  elements
        t[k] := a[k+nh]  // 'right' elements
    }

    for k:=0 to nh-1
    {
        {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
    }

    fourier_shift(t[], nh, is*0.5)

    rec_fft_dif2(s[], nh, b[], is)
    rec_fft_dif2(t[], nh, c[], is)

    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

The parameter `is` $= \sigma = \pm 1$ is the sign of the transform. The data length `n` must be a power of 2. The result is returned in the array `x[]`. Again, the routine does no normalization.

Pseudo code for a non-recursive procedure (a C++ implementation is given in [FXT: fft/fftdif2.cc]):

```
procedure fft_dif2(a[],ldn,is)
// complex a[0..2**ldn-1] input, result
{
```

```
    n := 2**ldn
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-1
        {
            e := exp(is*2*PI*I*j/m)
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := (u + v)
                a[r+j+mh] := (u - v) * e
            }
        }
    }
    revbin_permute(a[], n)
}
```

In DIF FFTs the `revbin_permute()`-procedure is called after the main loop, in the DIT code it was called before the main loop. As in the procedure for the DIT FFT (section 20.3.1.3 on page 380) the inner loops where swapped to save trigonometric computations.

Extracting the `ldm==1` stage of the outermost loop is again a good idea. Replace the line

```
    for  ldm:=ldn to 1 step -1
```

by

```
    for  ldm:=ldn to 2 step -1
```

and insert

```
    for r:=0 to n-1 step 2
    {
        {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
    }
```

before the call of `revbin_permute(a[], n)`.

20.4 Saving trigonometric computations

The sine and cosine computations are an expensive part of any FFT. There are two apparent ways for saving the involved CPU cycles, the use of lookup-tables and recursive methods. The CORDIC algorithms for sine and cosine given in section 33.2.1 on page 630 can be useful when implementing FFTs in hardware.

20.4.1 Using lookup tables

The idea is to precompute all necessary values, store them in an array, and later looking up the values needed. This is a good idea if one wants to compute many FFTs of the same (small) length. For FFTs of large sequences one gets large lookup tables that can introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a notable slowdown is possible. However, for a length-n FFT one does not need to store all the (n complex or $2\,n$ real) sine/cosine values $\exp(2\,\pi\,i\,k/n) = \cos(2\,\pi\,k/n) + i\,\sin(2\,\pi\,k/n)$ where $k = 0, 1, 2, 3, \ldots, n-1$. For the lookups one can use the symmetry relations

$$\cos(\pi + x) = -\cos(x) \tag{20.4-1a}$$

$$\sin(\pi + x) = -\sin(x) \tag{20.4-1b}$$

To reduce the interval from $0 \ldots 2\pi$ to $0 \ldots \pi$. Exploiting the relations

$$\cos(\pi/2 + x) = -\sin(x) \tag{20.4-2a}$$
$$\sin(\pi/2 + x) = +\cos(x) \tag{20.4-2b}$$

further reduces the interval to $0 \ldots \pi/2$. Finally, the relation

$$\sin(x) = \cos(\pi/2 - x) \tag{20.4-3}$$

shows that only the table of cosines is needed. That is, already a table of the $n/4$ real values $\cos(2\,\pi\,i\,k/n)$ for $k = 0, 1, 2, 3, \ldots, n/4-1$ suffices for a length-n FFT computation. The size of the table is thereby cut by a factor of 8. Possible cache problems can sometimes be mitigated by simply storing the trigonometric values in reversed order which can avoid many equidistant memory accesses.

20.4.2 Recursive generation

In FFT computations one typically needs the values

$$[\exp(i\,\varphi\,0) = 1, \quad \exp(i\,\varphi\,\gamma), \quad \exp(i\,\varphi\,2\,\gamma), \quad \exp(i\,\varphi\,3\,\gamma), \quad \ldots] \quad \text{where} \quad \varphi \in \mathbb{R}$$

in sequence. The naive idea for a recursive computation of these values is to precompute $d = \exp(i\,\varphi\,\gamma)$ and then compute the following value using the identity $\exp(i\,\varphi\,k\,\gamma) = d \cdot \exp(i\,\varphi\,(k-1)\,\gamma)$. This method, however, is of no practical value because the numerical error grows exponentially in the process.

A stable version of a trigonometric recursion for the computation of the sequence can be stated as follows. Precompute

$$c = \cos\varphi, \tag{20.4-4a}$$
$$s = \sin\varphi, \tag{20.4-4b}$$
$$\alpha = 1 - \cos\gamma \qquad \text{[Cancellation!]} \tag{20.4-4c}$$
$$= 2\left(\sin\frac{\gamma}{2}\right)^2 \qquad \text{[OK.]} \tag{20.4-4d}$$
$$\beta = \sin\gamma \tag{20.4-4e}$$

Then compute the next pair (c_+, s_+) from the previous one (c, s) via

$$c_+ = c - (\alpha\,c + \beta\,s); \tag{20.4-5a}$$
$$s_+ = s - (\alpha\,s - \beta\,c); \tag{20.4-5b}$$

The underlying idea is to use the relation $E(\varphi + \gamma) = E(\varphi) - E(\varphi) \cdot z$ where $E(x) := \exp(2\,\pi\,i\,x)$. This leads to $z = 1 - \cos\gamma - i\,\sin\gamma = 2\left(\sin\frac{\gamma}{2}\right)^2 - i\,\sin\gamma$.

Do not expect to get all the precision you would get with calls of the sine and cosine functions, but even for very long FFTs less than 3 bits of precision are lost. When working with (C-type) `double`s it might be a good idea to use the type `long double` with the trigonometric recursion: the generated values will then always be accurate within the `double`-precision, provided `long double`s are actually more precise than `double`s. With high precision multiplication routines (that is, with exact integer convolution) this can be mandatory.

A real-world example from [FXT: `fht_dif_core()` in fht/fhtdif.cc]:

```
[--snip--]
double tt = M_PI_4/kh;   // the angle increment
double s1 = 0.0,  c1 = 1.0;  // start at angle zero
double al = sin(0.5*tt);
al *= (2.0*al);
double be = sin(tt);
```

```
for (ulong i=1; i<kh; i++)
{
    double t1 = c1;
    c1 -= (al*t1+be*s1);
    s1 -= (al*s1-be*t1);

    // here c1 = cos(tt*i) and s1 = sin(tt*i)
[--snip--]
```

The variable `tt` equals γ in relations 20.4-4d and 20.4-4e on the facing page.

20.5 Higher radix FFT algorithms

Higher radix FFT algorithms save trigonometric computations. The radix-4 FFT algorithms presented in what follows replace all multiplications with complex factors $(0, \pm i)$ by the obvious simpler operations. Radix-8 algorithms also simplify the special cases where the sines and cosines equal $\pm\sqrt{1/2}$.

Further the bookkeeping overhead is reduced due to the more unrolled structure. Moreover, the number of loads and stores is reduced.

More notation

Let a be a length-n sequence where n is a multiple of m.

- Let $a^{(r\%m)}$ denote the subsequence of the elements with index x where $x \equiv r \bmod m$. For example, $a^{(0\%2)} = a^{(even)}$ and $a^{(3\%4)} = [a_3, a_7, a_{11}, a_{15}, \ldots]$. The length of $a^{(r\%m)}$ is n/m.
- Let $a^{(r/m)}$ denote the subsequence of elements with indices $\frac{r\,n}{m} \ldots \frac{(r+1)\,n}{m} - 1$. For example $a^{(1/2)} = a^{(right)}$ and $a^{(2/3)}$ is the last third of a. The length of $a^{(r/m)}$ is also n/m.

20.5.1 Decimation in time algorithms

First rewrite the radix-2 DIT step (relations 20.3-3a and 20.3-3b on page 379) in the new notation:

$$\mathcal{F}\big[\,a\,\big]^{(0/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\big[\,a^{(0\%2)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big] \tag{20.5-1a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/2)} \overset{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\big[\,a^{(0\%2)}\,\big] - \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big] \tag{20.5-1b}$$

The operator $\mathcal{S}$ is defined on section 20.3 on page 378, note that $\mathcal{S}^{0/2} = \mathcal{S}^0$ is the identity operator.

The derivation of the radix-4 step is analogous to the radix-2 step, it just involves more writing and does not give additional insights. So we just state the *radix-4 DIT FFT step* which can be applied when n is divisible by 4:

$$\mathcal{F}\big[\,a\,\big]^{(0/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + \mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] + \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + \mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{20.5-2a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + i\sigma\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] - i\sigma\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{20.5-2b}$$

$$\mathcal{F}\big[\,a\,\big]^{(2/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] + \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] - \mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{20.5-2c}$$

$$\mathcal{F}\big[\,a\,\big]^{(3/4)} \overset{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] - i\sigma\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + i\sigma\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big] \tag{20.5-2d}$$

In contrast to the radix-2 step that happens to be identical for forward and backward transform the sign of the transform $\sigma = \pm 1$ appears explicitly. The relations, written more compactly:

$$\begin{aligned}\mathcal{F}\big[\,a\,\big]^{(j/4)} \overset{n/4}{=}\ & +e^{\sigma\,2\,i\,\pi\,0\,j/4}\cdot\mathcal{S}^{0/4}\mathcal{F}\big[\,a^{(0\%4)}\,\big] + e^{\sigma\,2\,i\,\pi\,1\,j/4}\cdot\mathcal{S}^{1/4}\mathcal{F}\big[\,a^{(1\%4)}\,\big] \\ & +e^{\sigma\,2\,i\,\pi\,2\,j/4}\cdot\mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big] + e^{\sigma\,2\,i\,\pi\,3\,j/4}\cdot\mathcal{S}^{3/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\end{aligned} \tag{20.5-3}$$

where $j \in \{0,1,2,3\}$ and n is a multiple of 4. Still more compactly:

$$\mathcal{F}\big[\,a\,\big]^{(j/4)} \quad \overset{n/4}{=} \quad \sum_{k=0}^{3} e^{\sigma\, 2\, i\, \pi\, k\, j/4} \cdot \mathcal{S}^{k/4} \mathcal{F}\big[\,a^{(k\%4)}\,\big] \qquad j \in \{0,1,2,3\} \tag{20.5-4}$$

where the summation symbol denotes *element-wise* summation of the sequences. The dot indicates multiplication of all elements of the sequence by the exponential.

The general *radix-r DIT FFT step*, applicable when n is a multiple of r, is:

$$\mathcal{F}\big[\,a\,\big]^{(j/r)} \quad \overset{n/r}{=} \quad \sum_{k=0}^{r-1} e^{\sigma\, 2\, i\, \pi\, k\, j/r} \cdot \mathcal{S}^{k/r} \mathcal{F}\big[\,a^{(k\%r)}\,\big] \qquad j = 0,1,2,\ldots,r-1 \tag{20.5-5}$$

Our notation turned out to be useful indeed.

20.5.2 Decimation in frequency algorithms

The radix-2 DIF step (relations 20.3-6a and 20.3-6b on page 382), in the new notation:

$$\mathcal{F}\big[\,a\,\big]^{(0\%2)} \quad \overset{n/2}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{0/2}\Big(a^{(0/2)} + a^{(1/2)}\Big)\,\big] \tag{20.5-6a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%2)} \quad \overset{n/2}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{1/2}\Big(a^{(0/2)} - a^{(1/2)}\Big)\,\big] \tag{20.5-6b}$$

The *radix-4 DIF FFT step*, applicable for n divisible by 4, is

$$\mathcal{F}\big[\,a\,\big]^{(0\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{0/4}\Big(a^{(0/4)} + \quad a^{(1/4)} + a^{(2/4)} + \quad a^{(3/4)}\Big)\,\big] \tag{20.5-7a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{1/4}\Big(a^{(0/4)} + i\,\sigma\, a^{(1/4)} - a^{(2/4)} - i\,\sigma\, a^{(3/4)}\Big)\,\big] \tag{20.5-7b}$$

$$\mathcal{F}\big[\,a\,\big]^{(2\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{2/4}\Big(a^{(0/4)} - \quad a^{(1/4)} + a^{(2/4)} - \quad a^{(3/4)}\Big)\,\big] \tag{20.5-7c}$$

$$\mathcal{F}\big[\,a\,\big]^{(3\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{3/4}\Big(a^{(0/4)} - i\,\sigma\, a^{(1/4)} - a^{(2/4)} + i\,\sigma\, a^{(3/4)}\Big)\,\big] \tag{20.5-7d}$$

Again, $\sigma = \pm 1$ is the sign of the transform. More compactly:

$$\mathcal{F}\big[\,a\,\big]^{(j\%4)} \quad \overset{n/4}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{j/4} \sum_{k=0}^{3} e^{\sigma\, 2\, i\, \pi\, k\, j/4} \cdot a^{(k/4)}\,\big] \qquad j \in \{0,1,2,3\} \tag{20.5-8}$$

The general *radix-r DIF FFT step* is

$$\mathcal{F}\big[\,a\,\big]^{(j\%r)} \quad \overset{n/r}{=} \quad \mathcal{F}\big[\,\mathcal{S}^{j/r} \sum_{k=0}^{r-1} e^{\sigma\, 2\, i\, \pi\, k\, j/r} \cdot a^{(k/r)}\,\big] \qquad j \in \{0,1,2,\ldots,r-1\} \tag{20.5-9}$$

20.5.3 Implementation of radix-r FFTs

For the implementation of a radix-r FFT with $r \neq 2$ the `revbin_permute` routine has to be replaced by its radix-r version `radix_permute`. The reordering now swaps elements a_x with $a_{\tilde{x}}$ where $\tilde{x}$ is obtained from x by reversing its radix-r expansion (see section 2.2 on page 89). In most practical cases one considers $r = p^x$ where p is prime. Pseudo code for a radix $r = p^x$ DIT FFT:

```
procedure fftdit_r(a[], n, is)
// complex a[0..n-1] input, result
// p (hardcoded)
// r == power of p (hardcoded)
```

```
// n == power of p (not necessarily a power of r)
{
    radix_permute(a[], n, p)
    lx := log(r) / log(p)  // r == p ** lx
    ln := log(n) / log(p)
    ldm := (log(n)/log(p)) % lx
    if ( ldm != 0 )  // n is not a power of p
    {
        xx := p**lx
        for z:=0 to n-xx step xx
        {
            fft_dit_xx(a[z..z+xx-1], is)  // inlined length-xx dit fft
        }
    }
    for ldm:=ldm+lx to ln step lx
    {
        m  := p**ldm
        mr := m/r
        for j := 0 to mr-1
        {
            e := exp(is*2*PI*I*j/m)
            for k:=0 to n-m step m
            {
                // all code in this block should be
                // inlined, unrolled and combined:
                // temporary  u[0..r-1]
                for z:=0 to r-1
                {
                    u[z] := a[k+j+mr*z]
                }
                radix_permute(u[], r, p)
                for z:=1 to r-1  // e**0 == 1
                {
                    u[z] := u[z] * e**z
                }
                r_point_fft(u[], is)
                for z:=0 to r-1
                {
                    a[k+j+mr*z] := u[z]
                }
            }
        }
    }
}
```

Of course the loops that use the variable `z` have to be unrolled, the (length-p^x) scratch space `u[]` has to be replaced by explicit variables (for example, `u0, u1, ...`), and the `r_point_fft(u[],is)` shall be an inlined p^x-point FFT.

There is one pitfall: if one uses the radix-p permutation instead of a radix-p^x permutation (for example, the radix-2 `revbin_permute()` for a radix-4 FFT), then some additional reordering is necessary in the innermost loop. In the given pseudo code this is indicated by the `radix_permute(u[],p)` just before the `p_point_fft(u[],is)` line.

20.5.4 Radix-4 DIT FFT

C++ code for a radix-4 DIT FFT is given in [FXT: fft/fftdit4l.cc]:

```
static const ulong RX = 4;  // == r
static const ulong LX = 2;  // == log(r)/log(p) == log_2(r)

void
fft_dit4l(Complex *f, ulong ldn, int is)
// Decimation in time radix-4 FFT.
{
    double s2pi = ( is>0 ? 2.0*M_PI : -2.0*M_PI );
```

```
    const ulong n = (1UL<<ldn);
    revbin_permute(f, n);
    ulong ldm = (ldn&1);
    if ( ldm!=0 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong r=0; r<n; r+=2)
        {
            Complex a0 = f[r];
            Complex a1 = f[r+1];
            f[r]   = a0 + a1;
            f[r+1] = a0 - a1;
        }
    }
    ldm += LX;
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);
        double ph0 = s2pi/m;
        for (ulong j=0; j<m4; j++)
        {
            double phi = j*ph0;
            Complex e  = SinCos(phi);
            Complex e2 = SinCos(2.0*phi);
            Complex e3 = SinCos(3.0*phi);
            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;
                Complex a0 = f[i0];
                Complex a1 = f[i2]; // (!)
                Complex a2 = f[i1]; // (!)
                Complex a3 = f[i3];
                a1 *= e;
                a2 *= e2;
                a3 *= e3;
                Complex t0 = (a0+a2) + (a1+a3);
                Complex t2 = (a0+a2) - (a1+a3);
                Complex t1 = (a0-a2) + Complex(0,is) * (a1-a3);
                Complex t3 = (a0-a2) - Complex(0,is) * (a1-a3);
                f[i0] = t0;
                f[i1] = t1;
                f[i2] = t2;
                f[i3] = t3;
            }
        }
    }
}
```

For reasonable performance the call to the procedure `radix_permute(u[],p)` of the pseudo code has been replaced by changing indices in the loops where the `a[z]` are read. The respective lines are marked with the comment '`// (!)`'.

In order not to restrict the possible array sizes to powers of $p^x = 4$ but only to powers of $p = 2$ an additional radix-2 step has been prepended that is used when n is an odd power of two.

The routine [FXT: `fft_dit4_core_p1()` in fft/fftdit4.cc] is a reasonably optimized radix-4 DIT FFT implementation. It starts with an radix-2 or radix-8 step for the initial pass with trivial exp()-values. The core routine is hardcoded for $\sigma = +1$ and called with swapped real and imaginary part for the inverse transform as explained in section 20.8 on page 397.

The routine, however, uses separate arrays for real and imaginary parts which is very problematic with large transforms: the memory access pattern in skips that are a power of two *will* lead to cache problems.

A routine that uses the C++ type `complex` is given in [FXT: fft/cfftdit4.cc]. The core routine is hardcoded for $\sigma = -1$ (therefore the name suffix `_m1`):

```
void
fft_dit4_core_m1(Complex *f, ulong ldn)
// Auxiliary routine for fft_dit4()
// Radix-4 decimation in frequency FFT
// ldn := base-2 logarithm of the array length
// Fixed isign = -1
// Input data must be in revbin_permuted order
{
    const ulong n = (1UL<<ldn);

    if ( n<=2 )
    {
        if ( n==2 )  sumdiff(f[0], f[1]);
        return;
    }
    ulong ldm = ldn & 1;
    if ( ldm!=0 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  fft8_dit_core_m1(f+i0);  // isign
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)
        {
            ulong i1 = i0 + 1;
            ulong i2 = i1 + 1;
            ulong i3 = i2 + 1;

            Complex x, y, u, v;
            sumdiff(f[i0], f[i1], x, u);
            sumdiff(f[i2], f[i3], y, v);
            v *= Complex(0, -1);  // isign
            sumdiff(u, v, f[i1], f[i3]);
            sumdiff(x, y, f[i0], f[i2]);
        }
    }
    ldm += 2 * LX;

    for ( ; ldm<=ldn; ldm+=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);
        const double ph0 = -2.0*M_PI/m;  // isign

        for (ulong j=0; j<m4; j++)
        {
            double phi = j * ph0;
            Complex e  = SinCos(phi);
            Complex e2 = e * e;
            Complex e3 = e2 * e;

            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;

                Complex x = f[i1] * e2;
                Complex u;
                sumdiff3_r(x, f[i0], u);

                Complex v = f[i3] * e3;
                Complex y = f[i2] * e;
                sumdiff(y, v);
                v *= Complex(0, -1);  // isign

                sumdiff(u, v, f[i1], f[i3]);
                sumdiff(x, y, f[i0], f[i2]);
            }
        }
    }
}
```

The `sumdiff()` function is defined in [FXT: aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b}  <--| {a+b, a-b}
{ Type t=a-b; a+=b; b=t; }
```

The routine `fft8_dit_core_m1()` is an unrolled size-8 DIT FFT (hardcoded for $\sigma = -1$) given in [FXT: fft/fft8ditcore.cc]. We further need a version of the routine for the positive sign. It uses a routine `fft8_dit_core_p1()` for the computation of length-8 DIT FFTs (with $\sigma = -1$). The following changes need to be made in the core routine [FXT: fft/cfftdit4.cc]:

```
void
fft_dit4_core_p1(Complex *f, ulong ldn)
// Fixed isign = +1
{
    [--snip--]
        for (ulong i0=0; i0<n; i0+=8)  fft8_dit_core_p1(f+i0);  // isign
    [--snip--]
            v *= Complex(0, +1);  // isign
    [--snip--]
        const double ph0 = +2.0*M_PI/m;  // isign
    [--snip--]
                v *= Complex(0, +1);  // isign
    [--snip--]
}
```

The routine to be called by the user is

```
void
fft_dit4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// Radix-4 decimation in time algorithm
{
    revbin_permute(f, 1UL<<ld);
    if ( is>0 )  fft_dit4_core_p1(f, ldn);
    else         fft_dit4_core_m1(f, ldn);
}
```

A version that uses the separate arrays for real and imaginary part is given in [FXT: fft/fftdit4.cc]. The type `complex` version, should be preferred for large transforms.

20.5.5 Radix-4 DIF FFT

Pseudo code for a radix-4 DIF FFT:

```
procedure fftdif4(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm := ldn to 2 step -2
    {
        m  := 2**ldm
        mr := m/4
        for j := 0 to mr-1
        {
            e  := exp(is*2*PI*I*j/m)
            e2 := e * e
            e3 := e2 * e
            for r := 0 to n-m step m
            {
                u0 := a[r+j]
                u1 := a[r+j+mr]
                u2 := a[r+j+mr*2]
                u3 := a[r+j+mr*3]
                x := u0 + u2
                y := u1 + u3
                t0 := x + y  // == (u0+u2) + (u1+u3)
                t2 := x - y  // == (u0+u2) - (u1+u3)
                x := u0 - u2
```

```
                y := (u1 - u3)*I*is
                t1 := x + y  // == (u0-u2) + (u1-u3)*I*is
                t3 := x - y  // == (u0-u2) - (u1-u3)*I*is

                t1 := t1 * e
                t2 := t2 * e2
                t3 := t3 * e3

                a[r+j]      := t0
                a[r+j+mr]   := t2  // (!)
                a[r+j+mr*2] := t1  // (!)
                a[r+j+mr*3] := t3
            }
        }
    }

    if is_odd(ldn) then  // n not a power of 4
    {
        for r:=0 to n-2 step 2
        {
            {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
        }
    }

    revbin_permute(a[],n)
}
```

The C++ equivalent is [FXT: `fft_dif4l()` in fft/fftdif4l.cc]. A reasonably optimized implementation is given in [FXT: fft/cfftdif4.cc], it is hardcoded for $\sigma = +1$ (therefore the name suffix `_p1`):

```
static const ulong RX = 4;
static const ulong LX = 2;

void
fft_dif4_core_p1(Complex *f, ulong ldn)
// Auxiliary routine for fft_dif4().
// Radix-4 decimation in frequency FFT.
// Output data is in revbin_permuted order.
// ldn := base-2 logarithm of the array length.
// Fixed isign = +1
{
    const ulong n = (1UL<<ldn);

    if ( n<=2 )
    {
        if ( n==2 )  sumdiff(f[0], f[1]);
        return;
    }

    for (ulong ldm=ldn; ldm>=(LX<<1); ldm-=LX)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>LX);

        const double ph0 = 2.0*M_PI/m;  // isign

        for (ulong j=0; j<m4; j++)
        {
            double phi = j * ph0;
            Complex e  = SinCos(phi);
            Complex e2 = e * e;
            Complex e3 = e2 * e;

            for (ulong r=0; r<n; r+=m)
            {
                ulong i0 = j + r;
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;

                Complex x, y, u, v;
                sumdiff(f[i0], f[i2], x, u);
                sumdiff(f[i1], f[i3], y, v);
                v *= Complex(0, +1);  // isign

                diffsum3(x, y, f[i0]);
                f[i1] = y * e2;

                sumdiff(u, v, x, y);
                f[i3] = y * e3;
                f[i2] = x * e;
```

```
            }
        }
    }

    if ( ldn & 1 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  fft8_dif_core_p1(f+i0);  // isign
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)
        {
            ulong i1 = i0 + 1;
            ulong i2 = i1 + 1;
            ulong i3 = i2 + 1;
            Complex x, y, u, v;
            sumdiff(f[i0], f[i2], x, u);
            sumdiff(f[i1], f[i3], y, v);
            v *= Complex(0, +1);  // isign
            sumdiff(x, y, f[i0], f[i1]);
            sumdiff(u, v, f[i2], f[i3]);
        }
    }
}
```

The routine for $\sigma = -1$ needs changes where the comment `isign` appears [FXT: fft/cfftdif4.cc]::

```
void
fft_dif4_core_m1(Complex *f, ulong ldn)
// Fixed isign = -1
{
    [--snip--]
        const double ph0 = -2.0*M_PI/m;  // isign
    [--snip--]
                v *= Complex(0, -1);  // isign
    [--snip--]
        for (ulong i0=0; i0<n; i0+=8)  fft8_dif_core_m1(f+i0);  // isign
    [--snip--]
            v *= Complex(0, -1);  // isign
    [--snip--]
}
```

The routine to be called by the user is

```
void
fft_dif4(Complex *f, ulong ldn, int is)
// Fast Fourier Transform
// ldn := base-2 logarithm of the array length
// is := sign of the transform (+1 or -1)
// radix-4 decimation in frequency algorithm
{
    if ( is>0 )  fft_dif4_core_p1(f, ldn);
    else         fft_dif4_core_m1(f, ldn);
    revbin_permute(f, 1UL<<ldn);
}
```

A version that uses the separate arrays for real and imaginary part is given in [FXT: fft/fftdif4.cc]. Again, the type `complex` version, should be preferred for large transforms.

20.6 Split-radix Fourier transforms

The idea underlying the *split-radix FFT algorithm* is to use both radix-2 and radix-4 decompositions at the same time. We use one relation from the radix-2 (DIF) decomposition (relation 20.3-6a on page 382, the one for the even indices), and for the odd indices we use the radix-4 splitting (relations 20.5-7b and 20.5-7d on page 386) in a slightly reordered form. The radix-4 decimation in frequency (DIF) step

for the split-radix FFT is

$$\mathcal{F}\big[\,a\,\big]^{(0\%2)} \overset{n/2}{=} \mathcal{F}\big[\,\big(a^{(0/2)} + a^{(1/2)}\big)\,\big] \tag{20.6-1a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1\%4)} \overset{n/4}{=} \mathcal{F}\big[\,\mathcal{S}^{1/4}\Big(\big(a^{(0/4)} - a^{(2/4)}\big) + i\,\sigma\,\big(a^{(1/4)} - a^{(3/4)}\big)\Big)\,\big] \tag{20.6-1b}$$

$$\mathcal{F}\big[\,a\,\big]^{(3\%4)} \overset{n/4}{=} \mathcal{F}\big[\,\mathcal{S}^{3/4}\Big(\big(a^{(0/4)} - a^{(2/4)}\big) - i\,\sigma\,\big(a^{(1/4)} - a^{(3/4)}\big)\Big)\,\big] \tag{20.6-1c}$$

Now we have expressed the length-$N = 2^n$ FFT as one length-$N/2$ and two length-$N/4$ FFTs. A nice feature is that the operation count of the split-radix FFT is actually lower than that of the radix-4 FFT. Using the introduced notation it is almost trivial to write down the DIT version of the algorithm: The radix-4 decimation in time (DIT) step for the split-radix FFT is

$$\mathcal{F}\big[\,a\,\big]^{(0/2)} \overset{n/2}{=} \Big(\mathcal{F}\big[\,a^{(0\%2)}\,\big] + \mathcal{S}^{1/2}\mathcal{F}\big[\,a^{(1\%2)}\,\big]\Big) \tag{20.6-2a}$$

$$\mathcal{F}\big[\,a\,\big]^{(1/4)} \overset{n/4}{=} \Big(\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big]\Big) + i\sigma\mathcal{S}^{1/4}\Big(\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\Big) \tag{20.6-2b}$$

$$\mathcal{F}\big[\,a\,\big]^{(3/4)} \overset{n/4}{=} \Big(\mathcal{F}\big[\,a^{(0\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(2\%4)}\,\big]\Big) - i\sigma\mathcal{S}^{1/4}\Big(\mathcal{F}\big[\,a^{(1\%4)}\,\big] - \mathcal{S}^{2/4}\mathcal{F}\big[\,a^{(3\%4)}\,\big]\Big) \tag{20.6-2c}$$

Pseudo code for the split-radix DIF algorithm:

```
procedure fft_splitradix_dif(x[], y[], ldn, is)
{
    n := 2**ldn
    if n<=1  return
    n2 := 2*n
    for k:=1 to ldn
    {
        n2 := n2 / 2
        n4 := n2 / 4
        e := 2 * PI / n2
        for j:=0 to n4-1
        {
            a := j * e
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
            ix := j
            id := 2*n2
            while ix<n-1
            {
                i0 := ix
                while i0 < n
                {
                    i1 := i0 + n4
                    i2 := i1 + n4
                    i3 := i2 + n4
                    {x[i0], r1} := {x[i0] + x[i2], x[i0] - x[i2]}
                    {x[i1], r2} := {x[i1] + x[i3], x[i1] - x[i3]}
                    {y[i0], s1} := {y[i0] + y[i2], y[i0] - y[i2]}
                    {y[i1], s2} := {y[i1] + y[i3], y[i1] - y[i3]}
                    {r1, s3} := {r1+s2, r1-s2}
                    {r2, s2} := {r2+s1, r2-s1}
                    // complex mult: (x[i2],y[i2]) := -(s2,r1) * (ss1,cc1)
                    x[i2] :=  r1*cc1 - s2*ss1
                    y[i2] := -s2*cc1 - r1*ss1
                    // complex mult: (y[i3],x[i3]) := (r2,s3) * (cc3,ss3)
                    x[i3] :=  s3*cc3 + r2*ss3
                    y[i3] :=  r2*cc3 - s3*ss3
                    i0 := i0 + id
                }
```

```
                ix := 2 * id - n2 + j
                id := 4 * id
            }
        }
    }

    ix := 1
    id := 4

    while ix<n
    {
        for i0:=ix-1 to n-id step id
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            {y[i0], y[i1]} := {y[i0]+y[i1], y[i0]-y[i1]}
        }

        ix := 2 * id - 1
        id := 4 * id
    }

    revbin_permute(x[],n)
    revbin_permute(y[],n)

    if is>0
    {
        for j:=1 to n/2-1
        {
            swap(x[j], x[n-j])
        }

        for j:=1 to n/2-1
        {
            swap(y[j], y[n-j])
        }
    }
}
```

The C++ implementation given in [FXT: fft/fftsplitradix.cc] uses a DIF core as above which was given in [100]. The C++ type `complex` version of the split-radix FFT given in [FXT: fft/cfftsplitradix.cc] uses a DIF or DIT core, depending on the sign of the transform. Here we just give the DIF version:

```
void
split_radix_dif_fft_core(Complex *f, ulong ldn)
// Split-radix decimation in frequency (DIF) FFT.
// ldn := base-2 logarithm of the array length.
// Fixed isign = +1
// Output data is in revbin_permuted order.
{
    if ( ldn==0 )  return;

    const ulong n = (1UL<<ldn);

    double s2pi = 2.0*M_PI;  // pi*2*isign
    ulong n2 = 2*n;
    for (ulong k=1; k<ldn; k++)
    {
        n2 >>= 1;  // == n>>(k-1) == n, n/2, n/4, ..., 4
        const ulong n4 = n2 >> 2;  // == n/4, n/8, ..., 1
        const double e = s2pi / n2;

        {  // j==0:
            const ulong j = 0;
            ulong ix = j;
            ulong id = (n2<<1);
            while ( ix<n )
            {
                for (ulong i0=ix; i0<n; i0+=id)
                {
                    ulong i1 = i0 + n4;
                    ulong i2 = i1 + n4;
                    ulong i3 = i2 + n4;

                    Complex t0, t1;
                    sumdiff3(f[i0], f[i2], t0);
                    sumdiff3(f[i1], f[i3], t1);

                    // t1 *= Complex(0, 1);  // +isign
                    t1 = Complex(-t1.imag(), t1.real());

                    sumdiff(t0, t1);
```

```
                    f[i2] = t0; // * Complex(cc1, ss1);
                    f[i3] = t1; // * Complex(cc3, ss3);
                }

                ix = (id<<1) - n2 + j;
                id <<= 2;
            }
        }

        for (ulong j=1; j<n4; j++)
        {
            double a = j * e;
            double cc1,ss1, cc3,ss3;
            SinCos(a, &ss1, &cc1);
            SinCos(3.0*a, &ss3, &cc3);

            ulong ix = j;
            ulong id = (n2<<1);
            while ( ix<n )
            {
                for (ulong i0=ix; i0<n; i0+=id)
                {
                    ulong i1 = i0 + n4;
                    ulong i2 = i1 + n4;
                    ulong i3 = i2 + n4;

                    Complex t0, t1;
                    sumdiff3(f[i0], f[i2], t0);
                    sumdiff3(f[i1], f[i3], t1);

                    t1 = Complex(-t1.imag(), t1.real());

                    sumdiff(t0, t1);
                    f[i2] = t0 * Complex(cc1, ss1);
                    f[i3] = t1 * Complex(cc3, ss3);
                }

                ix = (id<<1) - n2 + j;
                id <<= 2;
            }
        }
    }

    for (ulong ix=0, id=4;  ix<n;  id*=4)
    {
        for (ulong i0=ix; i0<n; i0+=id)  sumdiff(f[i0], f[i0+1]);
        ix = 2*(id-1);
    }
}
```

The function `sumdiff3()` is defined in [FXT: aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff3(Type &a, Type b, Type &d)
// {a, b, d} <--| {a+b, b, a-b}  (used in split-radix fft)
{ d=a-b; a+=b; }
```

20.7 Symmetries of the Fourier transform

A bit of notation again. Let $\overline{a}$ be the length-n sequence a reversed around the element with index 0:

$$\overline{a}_0 := a_0 \tag{20.7-1a}$$

$$\overline{a}_{n/2} := a_{n/2} \quad \text{if } n \text{ even} \tag{20.7-1b}$$

$$\overline{a}_k := a_{n-k} = a_{-k} \tag{20.7-1c}$$

That is, we consider the indices modulo n and $\overline{a}$ is the sequence a with negated indices. Element zero stays in its place and for even n there is also an element with index $n/2$ that stays in place.
Example one, length-4: $a := [0, 1, 2, 3]$ then $\overline{a} = [0, 3, 2, 1]$ (zero and two stay).
Example two, length-5: $a := [0, 1, 2, 3, 4]$ then $\overline{a} = [0, 4, 3, 2, 1]$ (only zero stays).

Let a_S and a_A denote the symmetric and antisymmetric part of the sequence a, respectively:

$$a_S := \frac{1}{2}\,(a + \overline{a}) \tag{20.7-2a}$$

$$a_A := \frac{1}{2}\,(a - \overline{a}) \tag{20.7-2b}$$

The elements with index 0 (and $n/2$ for even n) a_A are zero. One has

$$a = a_S + a_A \tag{20.7-3a}$$

$$\overline{a} = a_S - a_A \tag{20.7-3b}$$

Let $c + i\,d$ be the FT of the sequence $a + i\,b$. Then

$$\mathcal{F}\big[\,(a_S + a_A) + i\,(b_S + b_A)\,\big] = (c_S + c_A) + i\,(d_S + d_A) \quad \text{where} \tag{20.7-4a}$$

$$\mathcal{F}\big[\,a_S\,\big] = c_S \quad \in \mathbb{R} \tag{20.7-4b}$$

$$\mathcal{F}\big[\,a_A\,\big] = i\,d_A \quad \in i\,\mathbb{R} \tag{20.7-4c}$$

$$\mathcal{F}\big[\,i\,b_S\,\big] = i\,d_S \quad \in i\,\mathbb{R} \tag{20.7-4d}$$

$$\mathcal{F}\big[\,i\,b_A\,\big] = c_A \quad \in \mathbb{R} \tag{20.7-4e}$$

Where we write $a \in \mathbb{R}$ as a short form for a purely real sequence a. Equivalently, write $a \in i\,\mathbb{R}$ for purely imaginary sequences. Thereby the FT of a complex symmetric or antisymmetric sequence is symmetric or antisymmetric, respectively:

$$\mathcal{F}\big[\,a_S + i\,b_S\,\big] = c_S + i\,d_S \tag{20.7-5a}$$

$$\mathcal{F}\big[\,a_A + i\,b_A\,\big] = c_A + i\,d_A \tag{20.7-5b}$$

The real and imaginary part of the transform of a symmetric sequence correspond to the real and imaginary part of the original sequence. With an antisymmetric sequence the transform of the real and imaginary part correspond to the imaginary and real part of the original sequence.

$$\mathcal{F}\big[\,(a_S + a_A)\,\big] = c_S + i\,d_A \tag{20.7-6a}$$

$$\mathcal{F}\big[\,i\,(b_S + b_A)\,\big] = c_A + i\,d_S \tag{20.7-6b}$$

Now let the sequence a be purely real. Then

$$\mathcal{F}\big[\,a_S\,\big] = +\overline{\mathcal{F}\big[\,a_S\,\big]} \quad \in \mathbb{R} \tag{20.7-7a}$$

$$\mathcal{F}\big[\,a_A\,\big] = -\overline{\mathcal{F}\big[\,a_A\,\big]} \quad \in i\,\mathbb{R} \tag{20.7-7b}$$

That is, the FT of a real symmetric sequence is real and symmetric and the FT of a real antisymmetric sequence is purely imaginary and antisymmetric. Thereby the FT of a general real sequence is the complex conjugate of its reversed:

$$\mathcal{F}\big[\,a\,\big] = \overline{\mathcal{F}\big[\,a\,\big]}^{\,*} \quad \textit{for} \quad a \in \mathbb{R} \tag{20.7-8}$$

Similarly, for a purely imaginary sequence $b \in i\mathbb{R}$:

$$\mathcal{F}\big[\,b_S\,\big] = +\overline{\mathcal{F}\big[\,b_S\,\big]} \quad \in i\,\mathbb{R} \tag{20.7-9a}$$

$$\mathcal{F}\big[\,b_A\,\big] = -\overline{\mathcal{F}\big[\,b_A\,\big]} \quad \in \mathbb{R} \tag{20.7-9b}$$

We compare the results of the Fourier transform and its inverse (the transform with negated sign σ) by symbolically writing the transforms as a complex multiplication with the trigonometric term (using C for cosine, S for sine):

$$\mathcal{F}\big[\,a + i\,b\,\big] : (a + i\,b)\,(C + i\,S) = (a\,C - b\,S) + i\,(b\,C + a\,S) \tag{20.7-10a}$$

$$\mathcal{F}^{-1}\big[\,a + i\,b\,\big] : (a + i\,b)\,(C - i\,S) = (a\,C + b\,S) + i\,(b\,C - a\,S) \tag{20.7-10b}$$

The terms on the right side can be identified with those in relation 20.7-4a. We see that changing the sign of the transform leads to a result where the components due to the antisymmetric components of the input are negated.

Now write $\mathcal{F}$ for the Fourier operator, and $\mathcal{R}$ for the reversal. We have $\mathcal{F}^4 = \mathrm{id}$, $\mathcal{F}^3 = \mathcal{F}^{-1}$, and $\mathcal{F}^2 = \mathcal{R}$. Thereby the inverse transform can be computed as either

$$\mathcal{F}^{-1} \;=\; \mathcal{R}\,\mathcal{F} \;=\; \mathcal{F}\,\mathcal{R} \tag{20.7-11}$$

20.8 Inverse FFT for free

Some FFT implementations are hardcoded for a fixed sign of the transform. If one cannot easily modify the implementation into the transform with the other sign (the inverse transform), then how can one compute the inverse FFT?

If the implementation uses separate arrays for real and imaginary part of the complex sequences to be transformed, as in

```
procedure my_fft(ar[], ai[], ldn)  // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
    // incredibly complicated code
    // that you cannot see how to modify
    // for is==-1
}
```

then do a follows: with the forward transform being

```
    my_fft(ar[], ai[], ldn)  // forward FFT
```

compute the inverse transform as

```
    my_fft(ai[], ar[], ldn)  // backward FFT
```

Note the swapped real and imaginary parts! The same trick works for a procedure coded for fixed `is`$= -1$.

To see why this works, we first note that

$$\begin{aligned} \mathcal{F}\big[\,a + i\,b\,\big] &= \mathcal{F}\big[\,a_S\,\big] + i\,\sigma\,\mathcal{F}\big[\,a_A\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] + \sigma\,\mathcal{F}\big[\,b_A\,\big] && \text{(20.8-1a)} \\ &= \mathcal{F}\big[\,a_S\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] + i\,\sigma\,\big(\mathcal{F}\big[\,a_A\,\big] - i\,\mathcal{F}\big[\,b_A\,\big]\big) && \text{(20.8-1b)} \end{aligned}$$

and the computation with swapped real- and imaginary parts gives

$$\mathcal{F}\big[\,b + i\,a\,\big] \;=\; \mathcal{F}\big[\,b_S\,\big] + i\,\mathcal{F}\big[\,a_S\,\big] + i\,\sigma\,\big(\mathcal{F}\big[\,b_A\,\big] - i\,\mathcal{F}\big[\,a_A\,\big]\big) \tag{20.8-2a}$$

... but the real and imaginary parts are implicitly swapped at the end of the computation, giving

$$\mathcal{F}\big[\,a_S\,\big] + i\,\mathcal{F}\big[\,b_S\,\big] - i\,\sigma\,\big(\mathcal{F}\big[\,a_A\,\big] - i\,\mathcal{F}\big[\,b_A\,\big]\big) \;=\; \mathcal{F}^{-1}\big[\,a + i\,b\,\big] \tag{20.8-2b}$$

When a complex type is used then the best way to achieve the inverse transform may be to reverse the sequence according to the symmetry of the FT according to relation 20.7-11: the transform with negated sign can be computed by reversing the order of the result (use [FXT: `reverse_0()` in perm/reverse.h]). The reversal can also happen with the input data before the transform, which is advantageous if the data has to be copied anyway (use [FXT: `copy_reverse_0()` in aux1/copy.h]). While not really 'free' the additional work will usually not matter.

A mechanical way to obtain a routine for the inverse FFT from a given FFT routine for length n is to replace all reads and writes at nonzero array indices i by the operations at indices $n - i$.

20.9 Real valued Fourier transforms

The Fourier transform of a purely real sequence $c = \mathcal{F}[\,a\,]$ where $a \in \mathbb{R}$ has a symmetric real part ($\mathfrak{Re}\,\overline{c} = \mathfrak{Re}\,c$, relation 20.7-8) and an antisymmetric imaginary part ($\mathfrak{Im}\,\overline{c} = -\,\mathfrak{Im}\,c$). The symmetric and antisymmetric part of the original sequence correspond to the symmetric (and purely real) and antisymmetric (and purely imaginary) part of the transform, respectively:

$$\mathcal{F}[\,a\,] \quad = \quad \mathcal{F}[\,a_S\,] + i\,\sigma\,\mathcal{F}[\,a_A\,] \tag{20.9-1}$$

Simply using a complex FFT for real input is basically a waste by a factor 2 of memory and CPU cycles. There are several ways out:

- wrapper routines for complex FFTs (section 20.9.3 on the facing page)
- usage of the fast Hartley transform (section 24.5 on page 491)
- special versions of the split-radix algorithm (section 20.9.4 on page 401)

All techniques have in common that they store only half of the complex result to avoid the redundancy due to the symmetries of a complex FT of purely real input. The result of a real to (half-) complex FT (abbreviated R2CFT) contains the purely real components c_0 (the 'DC-part' of the input signal) and, in case n is even, $c_{n/2}$ (the Nyquist frequency part). The inverse procedure, the (half-) complex to real transform (abbreviated C2RFT) must be compatible to the ordering of the R2CFT.

20.9.1 Sign of the transforms

The sign of the transform can be chosen arbitrarily to be either $+1$ or -1. Note that transform with the 'other sign' is *not* the inverse transform. The R2CFT and its inverse C2RFT must use the same sign.

Some R2CFT and C2RFT implementations are hardcoded for a fixed sign. In order to obtain the R2CFT with the other sign the trick (in the spirit of section 20.8) is to negate the imaginary part after the transform. This of course is not much of a trick at all. In case one has to copy the data anyway before the transform one can exploit the relation

$$\mathcal{F}[\,\overline{a}\,] \quad = \quad \mathcal{F}[\,a_S\,] - i\,\sigma\,\mathcal{F}[\,a_A\,] \tag{20.9-2}$$

That is, copy the real data in reversed order to get the transform with the other sign. This technique does not involve an extra pass and should be virtually for free.

For the complex-to-real FTs (C2RFT) one has to negate the imaginary part before the transform in order to obtain the inverse transform for the other sign.

20.9.2 Data ordering

Let c be the Fourier transform of the purely real sequence, it is stored in the array `a[]`. The procedures presented in what follows use one of the following schemes for storing the transformed sequence.

A scheme that interleaves real and imaginary parts ('complex ordering') is

$$\begin{aligned}
\mathtt{a}[0] &= \mathfrak{Re}\, c_0 \\
\mathtt{a}[1] &= \mathfrak{Re}\, c_{n/2} \\
\mathtt{a}[2] &= \mathfrak{Re}\, c_1 \\
\mathtt{a}[3] &= \mathfrak{Im}\, c_1 \\
\mathtt{a}[4] &= \mathfrak{Re}\, c_2 \\
\mathtt{a}[5] &= \mathfrak{Im}\, c_2 \\
&\vdots \\
\mathtt{a}[\mathtt{n}-2] &= \mathfrak{Re}\, c_{n/2-1} \\
\mathtt{a}[\mathtt{n}-1] &= \mathfrak{Im}\, c_{n/2-1}
\end{aligned} \tag{20.9-3}$$

Note the absence of the elements $\mathfrak{Im}\, c_0$ and $\mathfrak{Im}\, c_{n/2}$ which are always zero.

Some routines store the real parts in the lower half, and imaginary parts in upper half. The data in the lower half will always be ordered as follows:

$$\begin{aligned}
\mathtt{a}[0] &= \mathfrak{Re}\, c_0 \\
\mathtt{a}[1] &= \mathfrak{Re}\, c_1 \\
\mathtt{a}[2] &= \mathfrak{Re}\, c_2 \\
&\vdots \\
\mathtt{a}[\mathtt{n}/2] &= \mathfrak{Re}\, c_{n/2}
\end{aligned} \tag{20.9-4}$$

For the imaginary part of the result there are two schemes:
Scheme 1 ('parallel ordering') is

$$\begin{aligned}
\mathtt{a}[\mathtt{n}/2+1] &= \mathfrak{Im}\, c_1 \\
\mathtt{a}[\mathtt{n}/2+2] &= \mathfrak{Im}\, c_2 \\
\mathtt{a}[\mathtt{n}/2+3] &= \mathfrak{Im}\, c_3 \\
&\vdots \\
\mathtt{a}[\mathtt{n}-1] &= \mathfrak{Im}\, c_{n/2-1}
\end{aligned} \tag{20.9-5}$$

Scheme 2 ('antiparallel ordering') is

$$\begin{aligned}
\mathtt{a}[\mathtt{n}/2+1] &= \mathfrak{Im}\, c_{n/2-1} \\
\mathtt{a}[\mathtt{n}/2+2] &= \mathfrak{Im}\, c_{n/2-2} \\
\mathtt{a}[\mathtt{n}/2+3] &= \mathfrak{Im}\, c_{n/2-3} \\
&\vdots \\
\mathtt{a}[\mathtt{n}-1] &= \mathfrak{Im}\, c_1
\end{aligned} \tag{20.9-6}$$

20.9.3 Real valued FT via wrapper routines

A simple way to use a complex length-$n/2$ FFT for a real length-n FFT (n even) is to use some post- and preprocessing routines. For a real sequence a one feeds the (half length) complex sequence $f = a^{(even)} + i\, a^{(odd)}$ into a complex FFT. Some post-processing is necessary. This is not the most elegant real FFT available, but it is directly usable to turn complex FFTs of any (even) length into a real-valued FFT.

A C++ implementation of the real to complex FFT (R2CFT) is given in [FXT: realfft/realfftwrap.cc], the sign of the transform is hardcoded to $\sigma = +1$:

```
void
wrap_real_complex_fft(double *f, ulong ldn)
// Real to complex FFT (R2CFT)
{
    if ( ldn==0 )  return;

    fht_fft((Complex *)f, ldn-1, +1);  // cast

    const ulong n = 1UL<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]

        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]

        double f1r, f2i;
        sumdiff05(f[i3], f[i1], f1r, f2i);

        double f2r, f1i;
        sumdiff05(f[i2], f[i4], f2r, f1i);

        double c, s;
        double phi = i*phi0;
        SinCos(phi, &s, &c);

        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);

        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);

        // f[i4] = is * (ti + f1i);  // im hi
        // f[i2] = is * (ti - f1i);  // im low
        // =^=
        sumdiff( ti,  f1i, f[i4], f[i2]);
    }
    sumdiff(f[0], f[1]);
}
```

The output is ordered according to relations 20.9-3 on the previous page. The same ordering must be used for the input for the inverse routine, the complex to real FFT (C2RFT). Again the sign of the transform is hardcoded to $\sigma = +1$:

```
void
wrap_complex_real_fft(double *f, ulong ldn)
// Complex to real FFT (C2RFT).
{
    if ( ldn==0 )  return;

    const ulong n = 1UL<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = -M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]

        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]

        double f1r, f2i;
        // double f1r =  f[i1] + f[i3];  // re symm
        // double f2i =  f[i1] - f[i3];  // re asymm
        // =^=
        sumdiff(f[i1], f[i3], f1r, f2i);

        double f2r, f1i;
        // double f2r = -f[i2] - f[i4];  // im symm
        // double f1i =  f[i2] - f[i4];  // im asymm
        // =^=
        sumdiff(-f[i4], f[i2], f1i, f2r);
```

```
        double c, s;
        double phi = i*phi0;
        SinCos(phi, &s, &c);
        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);
        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);
        // f[i2] = ti - f1i;  // im low
        // f[i4] = ti + f1i;  // im hi
        // =^=
        sumdiff(ti, f1i, f[i4], f[i2]);
    }
    sumdiff(f[0], f[1]);
    if ( nh>=2 )  { f[nh] *= 2.0; f[nh+1] *= 2.0; }
    fht_fft((Complex *)f, ldn-1, -1);  // cast
}
```

20.9.4 Real valued split-radix Fourier transforms

We give pseudo code for the split-radix real to complex FFT and its inverse. The C++ implementations are given in [FXT: realfft/realfftsplitradix.cc]. The code given here follows [101], see also [219] (erratum for page 859 of [219]: at the start of the `DO 32` loop replace assignments by `CC1=COS(A)`, `SS1=SIN(A)`, `CC3=COS(A3)`, `SS3=SIN(A3)`).

Recall that the following pseudo code

```
{ a, b } := { c, d }
```

is a parallel assignment. For example,

```
{x0, x1} := {x0+x1, x0-x1 }
```

would translate to the C code

```
double s = x0 + x1,  d = x0 - x1;
x0 = s;  x1 = d;
```

Use the function [FXT: `sumdiff()` in aux0/sumdiff.h] for the translation of the parallel assignments.

20.9.5 Real to complex split-radix FFT

Pseudo code for the split-radix R2CFT algorithm, the sign of the transform is hardcoded to $\sigma = -1$.

```
procedure r2cft_splitradix_dit(x[], ldn)
{
    n := 2**ldn
    revbin_permute(x[], n);
    ix := 1;
    id := 4;
    do
    {
        i0 := ix-1
        while i0<n
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            i0 := i0 + id
        }
        ix := 2*id-1
        id := 4 * id
    }
    while ix<n
    n2 := 2
    nn := n/4
    while nn!=0
    {
        ix := 0
```

```
n2 := 2*n2
id := 2*n2
n4 := n2/4
n8 := n2/8
do  // ix loop
{
    i0 := ix
    while i0<n
    {
        i1 := i0
        i2 := i1 + n4
        i3 := i2 + n4
        i4 := i3 + n4

        {t1, x[i4]} := {x[i4]+x[i3], x[i4]-x[i3]}
        {x[i1], x[i3]} := {x[i1]+t1, x[i1]-t1}

        if  n4!=1
        {
            i1 := i1 + n8
            i2 := i2 + n8
            i3 := i3 + n8
            i4 := i4 + n8

            t1 := (x[i3]+x[i4]) * sqrt(1/2)
            t2 := (x[i3]-x[i4]) * sqrt(1/2)

            {x[i4], x[i3]} := {x[i2]-t1, -x[i2]-t1}
            {x[i1], x[i2]} := {x[i1]+t2,  x[i1]-t2}
        }

        i0 := i0 + id
    }

    ix := 2*id - n2
    id := 2*id
}
while ix<n

e := 2.0*PI/n2
a := e

for j:=2 to n8
{
    cc1 := cos(a)
    ss1 := sin(a)
    cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
    ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

    a := j*e

    ix := 0
    id := 2*n2

    do  // ix-loop
    {
        i0 := ix
        while  i0<n
        {
            i1 := i0 + j - 1
            i2 := i1 + n4
            i3 := i2 + n4
            i4 := i3 + n4
            i5 := i0 + n4 - j + 1
            i6 := i5 + n4
            i7 := i6 + n4
            i8 := i7 + n4

            // complex mult: (t2,t1) := (x[i7],x[i3]) * (cc1,ss1)
            t1 := x[i3]*cc1 + x[i7]*ss1
            t2 := x[i7]*cc1 - x[i3]*ss1

            // complex mult: (t4,t3) := (x[i8],x[i4]) * (cc3,ss3)
            t3 := x[i4]*cc3 + x[i8]*ss3
            t4 := x[i8]*cc3 - x[i4]*ss3

            t5 := t1 + t3
            t6 := t2 + t4
            t3 := t1 - t3
            t4 := t2 - t4

            {t2, x[i3]} := {t6+x[i6], t6-x[i6]}
            x[i8] := t2
            {t2, x[i7]} := {x[i2]-t3, -x[i2]-t3}
            x[i4] := t2
            {t1, x[i6]} := {x[i1]+t5, x[i1]-t5}
            x[i1] := t1
            {t1, x[i5]} := {x[i5]+t4, x[i5]-t4}
            x[i2] := t1

            i0 := i0 + id
```

```
                }

                ix := 2*id - n2
                id := 2*id

            }
            while ix<n
        }
        nn := nn/2
    }
}
```

The ordering of the output is given as relations 20.9-4 on page 399 for the real part, and relation 20.9-6 for the imaginary part.

20.9.6 Complex to real split-radix FFT

The following routine is the inverse of `r2cft_splitradix_dit()`. The imaginary part of the input data must be ordered according to relation 20.9-6 on page 399. We give pseudo code for the split-radix C2RFT algorithm, the sign of the transform is hardcoded to $\sigma = -1$:

```
procedure c2rft_splitradix_dif(x[], ldn)
{
    n := 2**ldn

    n2 := n/2
    nn := n/4
    while nn!=0
    {
        ix := 0
        id := n2
        n2 := n2/2
        n4 := n2/4
        n8 := n2/8
        do  // ix loop
        {
            i0 := ix
            while i0<n
            {
                i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                {x[i1], t1} := {x[i1]+x[i3], x[i1]-x[i3]}
                x[i2] := 2*x[i2]
                x[i4] := 2*x[i4]
                {x[i3], x[i4]} := {t1+x[i4], t1-x[i4]}

                if  n4!=1
                {
                    i1 := i1 + n8
                    i2 := i2 + n8
                    i3 := i3 + n8
                    i4 := i4 + n8

                    {x[i1], t1} := {x[i2]+x[i1], x[i2]-x[i1]}
                    {t2, x[i2]} := {x[i4]+x[i3], x[i4]-x[i3]}
                    x[i3] := -sqrt(2)*(t2+t1)
                    x[i4] :=  sqrt(2)*(t1-t2)
                }

                i0 := i0 + id
            }

            ix := 2*id - n2
            id := 2*id
        }
        while ix<n

        e := 2.0*PI/n2
        a := e

        for j:=2 to n8
        {
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
            a := j*e

            ix := 0
            id := 2*n2
            do  // ix-loop
```

```
            {
                i0 := ix
                while  i0<n
                {
                    i1 := i0 + j - 1
                    i2 := i1 + n4
                    i3 := i2 + n4
                    i4 := i3 + n4
                    i5 := i0 + n4 - j + 1
                    i6 := i5 + n4
                    i7 := i6 + n4
                    i8 := i7 + n4

                    {x[i1], t1} := {x[i1]+x[i6], x[i1]-x[i6]}
                    {x[i5], t2} := {x[i5]+x[i2], x[i5]-x[i2]}
                    {t3, x[i6]} := {x[i8]+x[i3], x[i8]-x[i3]}
                    {t4, x[i2]} := {x[i4]+x[i7], x[i4]-x[i7]}
                    {t1, t5} := {t1+t4, t1-t4}
                    {t2, t4} := {t2+t3, t2-t3}

                    // complex mult: (x[i7],x[i3]) := (t5,t4)  * (ss1,cc1)
                    x[i3] := t5*cc1 + t4*ss1
                    x[i7] := -t4*cc1 + t5*ss1

                    // complex mult: (x[i4],x[i8]) := (t1,t2)  * (cc3,ss3)
                    x[i4] := t1*cc3 - t2*ss3
                    x[i8] := t2*cc3 + t1*ss3

                    i0 := i0 + id
                }

                ix := 2*id - n2
                id := 2*id
            }
            while ix<n
        }

        nn := nn/2
    }

    ix := 1;
    id := 4;
    do
    {
        i0 := ix-1
        while i0<n
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            i0 := i0 + id
        }
        ix := 2*id-1
        id := 4 * id
    }
    while ix<n

    revbin_permute(x[], n);
}
```

20.10 Multidimensional Fourier transforms

20.10.1 Definition

Let $a_{x,y}$ ($x = 0, 1, 2, \ldots, C-1$ and $y = 0, 1, 2, \ldots, R-1$) be a 2-dimensional array. That is, a $R \times C$ 'matrix' of R rows (of length C) and C columns (of length R). Its 2-dimensional Fourier transform is defined by:

$$c = \mathcal{F}\big[\, a \,\big] \tag{20.10-1a}$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{y=0}^{R-1} a_{x,y}\, z^{+(x\,k/C + y\,h/R)} \quad \text{where} \quad z = e^{\sigma\, 2\,\pi\, i} \tag{20.10-1b}$$

where $k \in \{0, 1, 2, \ldots, C-1\}$, $h \in \{0, 1, 2, \ldots, R-1\}$, and $n = R \cdot C$. The inverse transform is

$$a = \mathcal{F}^{-1}\big[\, c \,\big] \tag{20.10-2a}$$

$$a_{x,y} = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h}\, z^{-(x\,k/C + y\,h/R)} \tag{20.10-2b}$$

For a m-dimensional array $a_{\vec{x}}$ (where $\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$ and $x_i \in 0, 1, 2, \ldots, S_i$) the m-dimensional Fourier transform $c_{\vec{k}}$ (where $\vec{k} = (k_1, k_2, k_3, \ldots, k_m)$ and $k_i \in 0, 1, 2, \ldots, S_i$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \cdots \sum_{x_m=0}^{S_m-1} a_{\vec{x}}\, z^{(x_1\,k_1/S_1 + x_2\,k_2/S_2 + \ldots + x_m\,k_m/S_m)} \tag{20.10-3a}$$

The inverse transform is, like in the 1-dimensional case, the complex conjugate transform.

20.10.2 The row-column algorithm

The equation of the definition of the two dimensional FT (relation 20.10-1a on the facing page) can be recast as

$$c_{k,h} = \frac{1}{\sqrt{n}} \sum_{y=0}^{R-1} \left[\exp(y\,h/R) \sum_{x=0}^{C-1} a_{x,y}\, \exp(x\,k/C) \right] \tag{20.10-4}$$

which shows that the 2-dimensional FT can be obtained by first applying 1-dimensional transforms on the rows and then applying 1-dimensional transforms on the columns. The same result is obtained when the columns are transformed first and then the rows.

This leads us directly to the *row-column algorithm* for 2-dimensional FFTs. Pseudo code to compute the two dimensional FT of `a[][]` using the row-column method:

```
procedure rowcol_ft(a[][], R, C, is)
{
    complex a[R][C]  // R (length-C) rows, C (length-R) columns
    for r:=0 to R-1  // FFT rows
    {
        fft(a[r][], C, is)
    }
    complex t[R]     // scratch array for columns
    for c:=0 to C-1  // FFT columns
    {
        copy a[0,1,...,R-1][c] to t[]  // get column
        fft(t[], R, is)
        copy t[] to a[0,1,...,R-1][c]  // write back column
    }
}
```

Here it is assumed that the rows lie in contiguous memory (as in the C language). The equivalent C++ code is given in [FXT: fft/twodimfft.cc].

Transposing the array before the column pass avoids the notorious problem with memory-cache and will do good for the performance in most cases. That is:

```
procedure rowcol_fft2d(a[][], R, C, is)
{
    complex a[R][C]  // R (length-C) rows, C (length-R) columns
    for r:=0 to R-1  // FFT rows
    {
        fft(a[r][], C, is)
    }
    transpose( a[R][C] )  // in-place
    for c:=0 to C-1  // FFT columns (which are rows now)
```

```
    {
        fft(a[c][], R, is)
    }
    transpose( a[C][R] )  // transpose back (note swapped R,C)
}
```

The transposing back at the end of the routine can be avoided if a back-transform will follow immediately as typical for convolution. The back-transform must then be called with `R` and `C` swapped.

The generalization to higher dimensions is straightforward, C++ code is given in [FXT: fft/ndimfft.cc].

20.11 The matrix Fourier algorithm (MFA)

The matrix Fourier algorithm (MFA) is an algorithm for 1-dimensional FFTs that works for data lengths $n = RC$. It is quite similar to the row-column algorithm (relation 20.10-4 on the previous page) for 2-dimensional FFTs. The only differences are n multiplications with trigonometric factors and a final matrix transposition.

Consider the input array as a $R \times C$-matrix (R rows, C columns), the rows shall be contiguous in memory. Then the *matrix Fourier algorithm* (MFA) can be stated as follows:

1. Apply a (length R) FFT on each column.
2. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$
3. Apply a (length C) FFT on each row.
4. Transpose the matrix.

Note the elegance! A variant of the MFA is called *four step FFT* in [23]. A trivial modification is obtained if the steps are executed in reversed order. The *transposed matrix Fourier algorithm* (TMFA) for the FFT:

1. Transpose the matrix.
2. Apply a (length C) FFT on each row of the matrix.
3. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$.
4. Apply a (length R) FFT on each column of the matrix.

A variant of the MFA that, apart from the transpositions, accesses the memory only in consecutive address ranges can be stated as

1. Transpose the matrix.
2. Apply a (length C) FFT on each row of the transposed matrix.
3. Multiply each matrix element (index r, c) by $\exp(\sigma\, 2\,\pi\, i\, r\, c/n)$.
4. Transpose the matrix back.
5. Apply a (length R) FFT on each row of the matrix.
6. Transpose the matrix (if the order of the transformed data matters).

The 'transposed' version of this algorithm is identical. The performance will depend critically on the performance of the transposition routine.

It is usually a good idea to use factors of the data length n that are close to $\sqrt{n}$. Of course one can apply the same algorithm for the row (or column) FFTs again: it can be an improvement to split n into 3 factors (as close to $n^{1/3}$ as possible) if a length-$n^{1/3}$ FFT fits completely into the cache. Especially for systems where CPU clock speed is much higher than memory clock speed the performance may increase

drastically, a speedup by a factor of three (even when compared to otherwise very well optimized FFTs) can sometimes be observed. Another algorithm that is efficient with large arrays is the localized transform as described in section 24.9 on page 497 for the Hartley transform.

Chapter 21

Algorithms for fast convolution

This chapter gives several FFT based algorithms for fast convolution. These are in practice the most important applications of the FFT. An efficient algorithm for the convolution of arrays that do not fit into the main memory (mass storage convolution) is given for both complex and real data. Further, weighted convolutions and their algorithms are introduced.

We describe how fast convolution can be used for computing the z-transform of sequences of arbitrary length. Another convolution based algorithm for the Fourier transform of arrays of prime length, Rader's algorithm, is described at the end of the chapter.

Convolution algorithms based on the fast Hartley transform are described in section 24.7. The dyadic convolution, which is computed via the Walsh transform is treated in section 22.7.

21.1 Convolution

The *cyclic convolution* (or *circular convolution*) of two length-n sequences $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b1, \ldots, b_{n-1}]$ is defined as the length-n sequence h with elements h_τ as:

$$h = a \circledast b \tag{21.1-1a}$$

$$h_\tau := \sum_{x+y \equiv \tau \pmod n} a_x \, b_y \tag{21.1-1b}$$

The last equation may be rewritten as

$$h_\tau := \sum_{x=0}^{n-1} a_x \, b_{(\tau - x) \mod n} \tag{21.1-2}$$

That is, indices $\tau - x$ wrap around, it is a cyclic convolution. A convenient way to illustrate the cyclic convolution of two sequences is shown in figure 21.1-A.

21.1.1 Direct computation

A C++ implementation of the computation by definition is [FXT: `slow_convolution()` in convolution/slowcnvl.h]:

```
template <typename Type>
void slow_convolution(const Type *f, const Type *g, Type *h, ulong n)
// (cyclic) convolution:  h[] :=  f[] (*) g[]
```

```
          +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          |
          0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
          1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15  0
          2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15  0  1
          3:   3  4  5  6   7  8  9 10  11 12 13 14  15  0  1  2

          4:   4  5  6  7   8  9 10 11  12 13 14 15   0  1  2  3
          5:   5  6  7  8   9 10 11 12  13 14 15  0   1  2  3  4
          6:   6  7  8  9  10 11 12 13  14 15  0  1   2  3  4  5
          7:   7  8  9 10  11 12 13 14  15  0  1  2   3  4  5  6

          8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
          9:   9 10 11 12  13 14 15  0   1  2  3  4   5  6  7  8
         10:  10 11 12 13  14 15  0  1   2  3  4  5   6  7  8  9
         11:  11 12 13 14  15  0  1  2   3  4  5  6   7  8  9 10

         12:  12 13 14 15   0  1  2  3   4  5  6  7   8  9 10 11
         13:  13 14 15  0   1  2  3  4   5  6  7  8   9 10 11 12
         14:  14 15  0  1   2  3  4  5   6  7  8  9  10 11 12 13
         15:  15  0  1  2   3  4  5  6   7  8  9 10  11 12 13 14


    +--   0   1   2   3   (a)
    |
    0:    0   1   2   4    ...
    1:    1   3   5 <--= h[5] contains a[2]*b[1]
    2:    4   8   9 <--= h[9] contains a[2]*b[2]
    3:    ...
  (b):
```

Figure 21.1-A: Semi-symbolic table of the cyclic convolution of two sequences (top). The entries denote where in the convolution the products of the input elements can be found (bottom).

```
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        for (ulong k=0; k<n; ++k)
        {
            ulong k2 = tau - k;
            if ( (long)k2<0 )  k2 += n;  // modulo n
            s += (f[k]*g[k2]);
        }
        h[tau] = s;
    }
}
```

The following version avoids the `if` statement in the inner loop:

```
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        ulong k = 0;
        for (ulong k2=tau;  k<=tau;  ++k, --k2)  s += (f[k]*g[k2]);
        for (ulong k2=n-1;  k<n;     ++k, --k2)  s += (f[k]*g[k2]);  // wrapped around
        h[tau] = s;
    }
```

For length-n sequences this procedure involves proportional n^2 operations, therefore it is slow for large values of n. For short lengths the algorithm is just fine. Unrolled routines will offer good performance, especially for convolutions of fixed length. For medium length convolutions the splitting schemes given in section 27.2 on page 524 and section 38.2 on page 799 are applicable.

21.1.2 Computation via FFT

The Fourier transform provides us with an efficient way to compute convolutions that only uses proportional $n \log(n)$ operations. The *convolution property* of the Fourier transform is

$$\mathcal{F}\big[\, a \circledast b \,\big] \;=\; \mathcal{F}\big[\, a \,\big]\, \mathcal{F}\big[\, b \,\big] \tag{21.1-3}$$

That is, convolution in original space is element-wise multiplication in Fourier space. The statement can be motivated as follows:

$$\mathcal{F}\big[\,a\,\big]_k\,\mathcal{F}\big[\,b\,\big]_k \;=\; \sum_x a_x\,z^{k\,x}\sum_y b_y\,z^{k\,y} \tag{21.1-4a}$$

$$=\; \sum_x a_x\,z^{k\,x}\sum_{\tau-x} b_{\tau-x}\,z^{k\,(\tau-x)} \quad \text{where} \quad y=\tau-x \tag{21.1-4b}$$

$$=\; \sum_x\sum_{\tau-x} a_x\,z^{k\,x}b_{\tau-x}\,z^{k\,(\tau-x)} \;=\; \sum_\tau\left(\sum_x a_x\,b_{\tau-x}\right)z^{k\,\tau} \tag{21.1-4c}$$

$$=\; \left(\mathcal{F}\big[\sum_x a_x\,b_{\tau-x}\,\big]\right)_k \;=\; \big(\mathcal{F}\big[\,a\circledast b\,\big]\big)_k \tag{21.1-4d}$$

Rewriting relation 21.1-3 on the preceding page as

$$a\circledast b \;=\; \mathcal{F}^{-1}\big[\,\mathcal{F}\big[\,a\,\big]\,\mathcal{F}\big[\,b\,\big]\,\big] \tag{21.1-5}$$

tells us how to proceed. We give pseudo code for the cyclic convolution of two complex valued sequences `x[]` and `y[]`, result is returned in `y[]`:

```
procedure fft_cyclic_convolution(x[], y[], n)
{
    complex x[0..n-1], y[0..n-1]

    // transform data:
    fft(x[], n, +1)
    fft(y[], n, +1)

    // convolution in transformed domain:
    for i:=0 to n-1
    {
        y[i] := y[i] * x[i]
    }

     // transform back:
    fft(y[], n, -1)

    // normalize:
    n1 := 1 / n
    for i:=0 to n-1
    {
        y[i] := y[i] * n1
    }
}
```

It is assumed that the procedure `fft()` does no normalization. For the normalization loop we precompute `1/n` and multiply as divisions are usually much slower than multiplications.

Relation 21.1-3 also holds for the more general z-transform (see section 21.5 on page 422). However, there is no (efficient) algorithm for the back-transform, so we cannot turn the relation

$$\mathcal{Z}\,[\,a\circledast b\,] \;=\; \mathcal{Z}\,[\,a\,]\;\mathcal{Z}\,[\,b\,] \tag{21.1-6}$$

into a practical algorithm for convolution.

21.1.3 Avoiding the revbin permutations

One can save the revbin permutations by observing that any DIF FFT is of the form

```
DIF_FFT_CORE(f, n);
revbin_permute(f, n);
```

and any DIT FFT is of the form

```
revbin_permute(f, n);
DIT_FFT_CORE(f, n);
```

Thereby a convolution routine that uses DIF FFTs for the forward transform and DIT FFTs as backward transform can omit the revbin permutations as demonstrated in the C++ implementation for the cyclic convolution of complex sequences [FXT: `fft_complex_convolution()` in convolution/fftcocnvl.cc]:

```
#define  DIT_FFT_CORE  fft_dit4_core_m1  // isign = -1
#define  DIF_FFT_CORE  fft_dif4_core_p1  // isign = +1
void
fft_complex_convolution(Complex * restrict f, Complex * restrict g,
                        ulong ldn, double v/*=0.0*/)
// (complex, cyclic) convolution:  g[] :=  f[] (*) g[]
// (use zero padded data for usual convolution)
// ldn := base-2 logarithm of the array length
// Supply a value for v for a normalization factor != 1/n
{
    const ulong n = (1UL<<ldn);

    DIF_FFT_CORE(f, ldn);
    DIF_FFT_CORE(g, ldn);
    if ( v==0.0 )  v = 1.0/n;
    for (ulong i=0; i<n; ++i)
    {
        Complex t = g[i] * f[i];
        g[i] = t * v;
    }
    DIT_FFT_CORE(g, ldn);
}
```

The signs of the two FFTs must be different but are else immaterial.

The so-called *auto convolution* (or *self convolution*) of a sequence is defined as the convolution of a sequence with itself: $h = a \circledast a$. The corresponding procedure needs only two instead of three FFTs.

21.1.4 Linear (acyclic) convolution

In the definition of the cyclic convolution (relations 21.1-1a and 21.1-1b) one can distinguish between those summands where the $x+y$ 'wrapped around' (i.e. $x+y = n+\tau$) and those where simply $x+y = \tau$ holds. These are (following the notation in [90]) denoted by $h^{(1)}$ and $h^{(0)}$ respectively. We have

$$h = h^{(0)} + h^{(1)} \qquad \text{where} \tag{21.1-7a}$$

$$h^{(0)} = \sum_{x \le \tau} a_x \, b_{\tau - x} \tag{21.1-7b}$$

$$h^{(1)} = \sum_{x > \tau} a_x \, b_{n+\tau - x} \tag{21.1-7c}$$

There is a simple way to separate $h^{(0)}$ and $h^{(1)}$ as the left and right half of a length-$2\,n$ sequence. This is just what the *linear convolution* (or *acyclic convolution*) does: Acyclic convolution of two (length-n) sequences a and b can be defined as that length-$2\,n$ sequence h which is the cyclic convolution of the *zero padded* sequences A and B:

$$A := [a_0, a_1, a_2, \ldots, a_{n-1}, 0, 0, \ldots, 0] \tag{21.1-8}$$

Same for B. Then the linear convolution is defined as

$$h = a \circledast_{lin} b \tag{21.1-9a}$$

$$h_\tau := \sum_{x=0}^{2\,n-1} A_x \, B_{\tau - x} \qquad \tau = 0, 1, 2, \ldots, 2\,n-1 \tag{21.1-9b}$$

As an illustration consider the convolution of the sequence $[1, 1, 1, 1]$ with itself: its linear self convolution is the length-8 sequence $[h_0][h_1] = [1, 2, 3, 4][3, 2, 1, 0]$, its cyclic self convolution is $[h_0 + h_1] = [4, 4, 4, 4]$.

```
+--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
|
0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
1:   1  2  3  4   5  6  7  8   9 10 11 12  13 14 15 16
2:   2  3  4  5   6  7  8  9  10 11 12 13  14 15 16 17
3:   3  4  5  6   7  8  9 10  11 12 13 14  15 16 17 18

4:   4  5  6  7   8  9 10 11  12 13 14 15  16 17 18 19
5:   5  6  7  8   9 10 11 12  13 14 15 16  17 18 19 20
6:   6  7  8  9  10 11 12 13  14 15 16 17  18 19 20 21
7:   7  8  9 10  11 12 13 14  15 16 17 18  19 20 21 22

8:   8  9 10 11  12 13 14 15  16 17 18 19  20 21 22 23
9:   9 10 11 12  13 14 15 16  17 18 19 20  21 22 23 24
10: 10 11 12 13  14 15 16 17  18 19 20 21  22 23 24 25
11: 11 12 13 14  15 16 17 18  19 20 21 22  23 24 25 26

12: 12 13 14 15  16 17 18 19  20 21 22 23  24 25 26 27
13: 13 14 15 16  17 18 19 20  21 22 23 24  25 26 27 28
14: 14 15 16 17  18 19 20 21  22 23 24 25  26 27 28 29
15: 15 16 17 18  19 20 21 22  23 24 25 26  27 28 29 30
```

Figure 21.1-B: Semi-symbolic table for the (length-31) linear convolution of two length-16 sequences.

The semi-symbolic table for the acyclic convolution is given in figure 21.1-B. The elements in the lower right triangle do not 'wrap around' anymore, they go to extra buckets. Note there are 31 buckets labeled $0 \ldots 30$.

Linear convolution is *polynomial multiplication*: let $A = a_0 + a_1\,x + a_2\,x^2 + \ldots$, $B = b_0 + b_1\,x + b_2\,x^2 + \ldots$ and $C = A\,B = c_0 + c_1\,x + c_2\,x^2 + \ldots$ then

$$c_k \quad = \quad \sum_{i+j=k} a_i\,b_j \tag{21.1-10}$$

Chapter 27 on page 523 explains how fast convolution algorithms can be used for fast multiplication of multiprecision numbers.

The direct (slow) algorithm can be modified to compute just $h^{(0)}$ or $h^{(1)}$ [FXT: convolution/slowcnvlhalf.h]:

```
template <typename Type>
void slow_half_convolution(const Type *f, const Type *g, Type *h, ulong n, int h01)
// Half cyclic convolution.
// Part determined by h01 which must be 0 or 1.
// n := array length
{
    if ( 0==h01 )  // compute h0:
    {
        for (ulong tau=0;  tau<n;  ++tau)
        {
            Type s0 = 0.0;
            for (ulong k=0, k2=tau;  k<=tau;  ++k, --k2)  s0 += (f[k]*g[k2]);
            h[tau] = s0;
        }
    }
    else  // compute h1 (wrapped part):
    {
        for (ulong tau=0;  tau<n;  ++tau)
        {
            Type s1 = 0.0;
            for (ulong k2=n-1, k=tau+1;  k<n;  ++k, --k2)  s1 += (f[k]*g[k2]);
            h[tau] = s1;
        }
    }
}
```

The cost is half of what it is for the linear convolution. With the FFT-based computation of the convolution the parts $h^{(0)}$ and $h^{(1)}$ can be isolated using weighted convolution algorithms that are given in section 21.3 on page 417. The cost is that of a linear convolution. No algorithm to compute just one of $h^{(0)}$ or $h^{(1)}$ that is significantly cheaper is known.

21.2 Correlation

+--	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
|																
0:	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1:	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2
2:	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3
3:	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4
4:	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5
5:	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6
6:	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7
7:	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
8:	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9
9:	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10
10:	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11
11:	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12
12:	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13
13:	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14
14:	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15
15:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 21.2-A: Semi-symbolic table for the (cyclic) correlation of two length-16 sequences.

+--	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
|																
0:	0	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
1:	1	0	31	30	29	28	27	26	25	24	23	22	21	20	19	18
2:	2	1	0	31	30	29	28	27	26	25	24	23	22	21	20	19
3:	3	2	1	0	31	30	29	28	27	26	25	24	23	22	21	20
4:	4	3	2	1	0	31	30	29	28	27	26	25	24	23	22	21
5:	5	4	3	2	1	0	31	30	29	28	27	26	25	24	23	22
6:	6	5	4	3	2	1	0	31	30	29	28	27	26	25	24	23
7:	7	6	5	4	3	2	1	0	31	30	29	28	27	26	25	24
8:	8	7	6	5	4	3	2	1	0	31	30	29	28	27	26	25
9:	9	8	7	6	5	4	3	2	1	0	31	30	29	28	27	26
10:	10	9	8	7	6	5	4	3	2	1	0	31	30	29	28	27
11:	11	10	9	8	7	6	5	4	3	2	1	0	31	30	29	28
12:	12	11	10	9	8	7	6	5	4	3	2	1	0	31	30	29
13:	13	12	11	10	9	8	7	6	5	4	3	2	1	0	31	30
14:	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	31
15:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 21.2-B: Semi-symbolic table for the linear (acyclic) correlation of two length-16 sequences.

The *cyclic correlation* (or *circular correlation*) of two *real* length-n sequences $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b_1, \ldots, b_{n-1}]$ is defined as the length-n sequence h with elements h_τ as:

$$h_\tau \quad := \sum_{x-y \equiv \tau \bmod n} a_x \, b_y \tag{21.2-1}$$

The relation can be recast as

$$h_\tau \quad = \quad \sum_{x=0}^{n-1} a_x \, b_{(\tau+x) \bmod n} \tag{21.2-2}$$

The semi-symbolic table for the (cyclic) correlation is shown in figure 21.2-A. For the computation of the *linear* (or *acyclic*) correlation the sequences have to be zero-padded as in the algorithm for the linear convolution. The semi-symbolic table is shown in figure 21.2-B.

The *auto correlation* (or *self-correlation*) is the correlation of a sequence with itself, the correlation of two distinct sequences is also called *cross correlation*. The term *auto correlation function* (ACF) is often used for the auto correlation sequence.

21.2.1 Direct computation

A C++ implementation of the computation by the definition is [FXT: correlation/slowcorr.h]:

```
template <typename Type>
void slow_correlation(const Type *f, const Type *g, Type * restrict h, ulong n)
// Cyclic correlation of f[], g[], both real-valued sequences.
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        for (ulong k=0; k<n; ++k)
        {
             ulong k2 = k + tau;
             if ( k2>=n )  k2 -= n;
             s += (g[k]*f[k2]);
        }
        h[tau] = s;
    }
}
```

The `if` statement in the inner loop is avoided by the following version:

```
    for (ulong tau=0;  tau<n;  ++tau)
    {
        Type s = 0.0;
        ulong k = 0;
        for (ulong k2=tau;  k2<n;  ++k, ++k2)  s += (g[k]*f[k2]);
        for (ulong k2=0;     k<n;  ++k, ++k2)  s += (g[k]*f[k2]);
        h[tau] = s;
    }
```

For the linear correlation one can avoid zero products:

```
template <typename Type>
void slow_correlation0(const Type *f, const Type *g, Type * restrict h, ulong n)
// Linear correlation of f[], g[], both real-valued sequences.
// n := array length
// Version for zero padded data:
//   f[k],g[k] == 0 for k=n/2 ... n-1
// n must be >=2
{
    const ulong nh = n/2;
    for (ulong tau=0; tau<nh; ++tau)  // k2 == tau + k
    {
        Type s = 0;
        for (ulong k=0, k2=tau;  k2<nh;  ++k, ++k2)  s += (f[k]*g[k2]);
        h[tau] = s;
    }

    for (ulong tau=nh; tau<n; ++tau)  // k2 == tau + k - n
    {
        Type s = 0;
        for (ulong k=n-tau, k2=0;  k<nh;  ++k, ++k2)  s += (f[k]*g[k2]);
        h[tau] = s;
    }
}
```

The algorithm involves proportional n^2 operations and is therefore slow with very long arrays.

21.2.2 Computation via FFT

A simple algorithm for fast correlation follows from the relation

$$h_\tau \;=\; \mathcal{F}^{-1}\big[\,\mathcal{F}\big[\,\overline{a}\,\big]\,\mathcal{F}\big[\,b\,\big]\,\big] \tag{21.2-3}$$

That is, use a convolution algorithm with one of the input sequences reversed (indices negated modulo n). For purely real sequences the relation is equivalent to complex conjugation of one of the inner transforms:

$$h_\tau \;=\; \mathcal{F}^{-1}\big[\,\mathcal{F}\big[\,a\,\big]^*\,\mathcal{F}\big[\,b\,\big]\,\big] \tag{21.2-4}$$

For the computation of self-correlation the latter relation is the only reasonable way to go: first transform the input sequence, then multiply each element by its complex conjugate and finally transform back. A C++ implementation is [FXT: correlation/fftcorr.cc]:

```
void
fft_correlation(double *f, double *g, ulong ldn)
// Cyclic correlation of f[], g[], both real-valued sequences.
// Result is written to g[].
// ldn := base-2 logarithm of the array length
{
    const ulong n=(1UL<<ldn);
    const ulong nh=(n>>1);

    fht_real_complex_fft(f, ldn);  // real, imag part in lower, upper half
    fht_real_complex_fft(g, ldn);

    const double v = 1.0/n;
    g[0]  *= f[0] * v;
    g[nh] *= f[nh] * v;
    for (ulong i=1,j=n-1; i<nh;  ++i,--j)  // real at index i, imag at index j
    {
        cmult_n(f[i], -f[j], g[i], g[j], v);
    }

    fht_complex_real_fft(g, ldn);
}
```

The function `cmult_n()` is defined in [FXT: aux0/cmult.h]:

```
static inline void
cmult_n(double c, double s, double &u, double &v, double dn)
// {u,v} <--| {dn*(u*c-v*s), dn*(u*s+v*c)}
{ double t = u*s+v*c;  u *= c;  u -= v*s;  u *= dn;  v = t*dn; }
```

We note that relation 21.2-4 also holds for complex sequences.

21.2.3 Correlation and difference sets *

The linear auto-correlation of a sequence that contains zeros and ones only (a *delta sequence*) is the set of mutual differences of the positions of the ones, including multiplicity. An example:

```
[1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0]  <--= delta array  R
[4, 2, 1, 2, 1, 0, 0, 1, 2, 1, 2]  <--= linear auto correlation ACF
 0, 1, 2, 3, 4, 5,-5,-4,-3,-2,-1   <--= index
```

Element zero of the ACF tells us that there are four elements in R (each element has difference zero to just itself). Element one tells us that there are two pairs of consecutive elements, it is identical to the last element (element minus one). There is just one pair of elements in R whose indices differ by two (elements two and minus two of the ACF), and so on. Note that the ACF does not tell us where the elements with a certain difference are.

The delta array with ones at the 7 positions 0, 3, 4, 12, 18, 23, and 25 has the ACF

```
[7, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, (+symm.)]
 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,       ...                                  26, <--= index
```

That is, a ruler of length 26 with marks only at the 7 given positions can be used to measure most of the distances up to 26 (the smallest missing distance is 10). Further, no distance appears more than once. Sequences with this property are called *Golomb rulers* and they are very hard to find.

If we allow for two rulers then the set of mutual differences in positions is the cross correlation. For this setting analogues of Golomb rulers (that do not have any missing differences) can be found. We use dots for zeros:

```
11..11..........11..11.....................................  <--= R1
1.1.....1.1.....1.1.....1.1.....1.1.....1.1.....1.1.....1.  <--= R2
1111111111111111111111111111111111111111111111111111111111  <--= cross correlation
```

The rulers are binary representations of the evaluations $F(1/2)$ and $F(1/4)$ of a curious function given in section 36.10.1 on page 715.

21.3 Weighted Fourier transforms and convolutions

21.3.1 The weighted Fourier transform

We define a new kind of transform by slightly modifying the definition of the FT (formula 20.1-1a on page 375):

$$c = \mathcal{W}_v[a] \tag{21.3-1a}$$

$$c_k := \sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k} \qquad v_x \neq 0 \quad \forall x \tag{21.3-1b}$$

where $z := e^{\sigma\, 2\,\pi\, i/n}$. The sequence c shall be called (discrete) *weighted transform* of the sequence a with the weight (sequence) v. Note the v_x that entered: the weighted transform with $v_x = \frac{1}{\sqrt{n}}\ \forall x$ is just the usual Fourier transform. The inverse transform is

$$a = \mathcal{W}_v^{-1}[c] \tag{21.3-2a}$$

$$a_x = \frac{1}{n\, v_x} \sum_{k=0}^{n-1} c_k\, z^{-x\,k} \tag{21.3-2b}$$

This can be easily seen:

$$\mathcal{W}_v^{-1}\left[\mathcal{W}_v[a]\right]_y = \frac{1}{n\, v_y} \sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k}\, z^{-y\,k} \tag{21.3-3a}$$

$$= \frac{1}{n} \sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\, \frac{1}{v_y}\, a_x\, z^{x\,k}\, z^{-y\,k} \tag{21.3-3b}$$

$$= \frac{1}{n} \sum_{x=0}^{n-1} v_x\, \frac{1}{v_y}\, a_x\, \delta_{x,y}\, n \quad = a_y \tag{21.3-3c}$$

Obviously all v_x have to be invertible. That $\mathcal{W}_v\left[\mathcal{W}_v^{-1}[a]\right]$ is also identity is apparent from the definitions.

Given an FFT routine it is trivial to set up a weighted Fourier transform. Pseudo code for the discrete weighted Fourier transform:

```
procedure weighted_ft(a[], v[], n, is)
{
    for x:=0 to n-1
    {
        a[x] := a[x] * v[x]
    }
    fft(a[], n, is)
}
```

The inverse is essentially identical. Pseudo code for the inverse discrete weighted Fourier transform:

```
procedure inverse_weighted_ft(a[], v[], n, is)
{
    fft(a[], n, -is)
    for x:=0 to n-1
    {
        a[x] := a[x] / v[x]
    }
}
```

The C++ implementations are given in [FXT: fft/weightedfft.cc].

21.3.2 Weighted convolution

Define the *weighted (cyclic) convolution* h_v by

$$h_v = a \circledast_{\{v\}} b \tag{21.3-4a}$$
$$= \mathcal{W}_v^{-1}\left[\mathcal{W}_v\left[a\right]\,\mathcal{W}_v\left[b\right]\right] \tag{21.3-4b}$$

Then, for the special case $v_x = V^x$, one has

$$h_v = h^{(0)} + V^n\, h^{(1)} \tag{21.3-5}$$

Here $h^{(0)}$ and $h^{(1)}$ are defined as in relation 21.1-7a on page 412. It is not hard to see why this is: up to the final division by the weight sequence, the weighted convolution is just the cyclic convolution of the two weighted sequences, which is for the element with index τ equal to

$$\sum_{x+y\equiv\tau \mod n} (a_x\, V^x)\,(b_y\, V^y) = \sum_{x\le\tau} a_x\, b_{\tau-x}\, V^\tau + \sum_{x>\tau} a_x\, b_{n+\tau-x}\, V^{n+\tau} \tag{21.3-6}$$

Final division of this element (by V^τ) gives $h^{(0)} + V^n\, h^{(1)}$ as stated.

The cases when V^n is some root of unity are particularly interesting: For $V^n = \pm i = \pm\sqrt{-1}$ one obtains the so-called *right-angle convolution*:

$$h_v = h^{(0)} \mp i\, h^{(1)} \tag{21.3-7}$$

This gives a nice possibility to directly use complex FFTs for the computation of a linear (acyclic) convolution of two real sequences: For length-n sequences the elements of the linear convolution with indices $0, 1, \ldots, n-1$ are the real part of the result, the elements $n, n+1, \ldots, 2\,n-1$ are the imaginary part. Choosing $V^n = -1$ leads to the *negacyclic convolution* (or *skew circular convolution*):

$$h_v = h^{(0)} - h^{(1)} \tag{21.3-8}$$

Cyclic, negacyclic and right-angle convolution can be understood as polynomial products modulo the polynomials $z^n - 1$, $z^n + 1$ and $z^n \pm i$, respectively (see [186]).

C++ implementations of the weighted-, negacyclic- and right-angle (self) convolution are given in [FXT: convolution/weightedconv.cc].

+--	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
|																
0:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0-
2:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0-	1-
3:	3	4	5	6	7	8	9	10	11	12	13	14	15	0-	1-	2-
4:	4	5	6	7	8	9	10	11	12	13	14	15	0-	1-	2-	3-
5:	5	6	7	8	9	10	11	12	13	14	15	0-	1-	2-	3-	4-
6:	6	7	8	9	10	11	12	13	14	15	0-	1-	2-	3-	4-	5-
7:	7	8	9	10	11	12	13	14	15	0-	1-	2-	3-	4-	5-	6-
8:	8	9	10	11	12	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-
9:	9	10	11	12	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-
10:	10	11	12	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-
11:	11	12	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	10-
12:	12	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	10-	11-
13:	13	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	10-	11-	12-
14:	14	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	10-	11-	12-	13-
15:	15	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	10-	11-	12-	13-	14-

Figure 21.3-A: Semi-symbolic table for the negacyclic convolution. The products that enter with negative sign are indicated with a postfix minus at the corresponding entry.

The semi-symbolic table for the negacyclic convolution is shown in figure 21.3-A. With right-angle convolution the minuses have to be replaced by $i = \sqrt{-1}$ which means the wrap-around (i.e. $h^{(1)}$) elements

go to the imaginary part. With real input one thereby effectively separates $h^{(0)}$ and $h^{(1)}$. Thereby the linear convolution of real sequences can be computed using the complex right-angle convolution.

With routines for both cyclic and negacyclic convolution the parts $h^{(0)}$ and $h^{(1)}$ can be computed as sum and difference, respectively. Thereby all expressions of the form $\alpha\, h^{(0)} + \beta\, h^{(1)}$ where $\alpha, \beta \in \mathbb{C}$ can be computed.

The direct (slow, proportional n^2) computation can be obtained by a minimal modification of the non-weighted convolution algorithm [FXT: convolution/slowweightedcnvl.h]:

```
template <typename Type>
void slow_weighted_convolution(const Type *f, const Type *g, Type *h, ulong n, Type w)
// weighted (cyclic) convolution:  h[] :=  f[] (*)_w g[]
// n := array length
{
    for (ulong tau=0;  tau<n;  ++tau)
    {
        ulong k = 0;
        Type s0 = 0.0;
        for (ulong k2=tau;  k<=tau;  ++k, --k2)  s0 += (f[k]*g[k2]);
        Type s1 = 0.0;
        for (ulong k2=n-1;  k<n;     ++k, --k2)  s1 += (f[k]*g[k2]);  // wrapped around
        h[tau] = s0 + s1*w;
    }
}
```

21.4 Convolution using the MFA

We give an algorithm for convolution that use the matrix Fourier algorithm (MFA, see section 20.11 on page 406). The MFA is used for the forward transform and the transposed algorithm (TMFA) for the backward transform. The elements of each row are assumed to lie contiguous in memory. For the sake of simplicity auto convolution is considered. The matrix FFT convolution algorithm:

1. Apply a (length R) FFT on each column.
   (*memory access with C-skips*)
2. Multiply each matrix element (index r, c) by $\exp(+\sigma\, 2\,\pi\, i\, r\, c/n)$.
3. Apply a (length C) FFT on each row.
   (*memory access without skips*)
4. Complex square row (element-wise).
5. Apply a (length C) FFT on each row (of the transposed matrix).
   (*memory access is without skips*)
6. Multiply each matrix element (index r, c) by $\exp(-\sigma\, 2\,\pi\, i\, r\, c/n)$.
7. Apply a (length R) FFT on each column (of the transposed matrix).
   (*memory access with C-skips*)

Note that steps 3, 4, and 5 constitute a length-C convolution on each row.

C++ implementations of the cyclic and linear convolution using the described algorithm are given in [FXT: convolution/matrixfftcnvl.cc]. The code for self-convolution can be found in [FXT: convolution/matrixfftcnvla.cc].

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm given on page 419:

1. Apply a FFT on each column.

2. On each row apply the weighted convolution with $V^C = e^{2\,\pi\,i\,r/R} = 1^{r/R}$ where R is the total number of rows, $r = 0..R-1$ the index of the row, C the length of each row (equivalently, the total number columns)
3. Apply a FFT on each column (of the transposed matrix).

We first consider the special cases of two and three rows and then formulate an MFA-based algorithm for the convolution of real sequences.

The case $R = 2$

Define s and d as the sums and differences of the lower and higher halves of a given sequence x:

$$s := x^{(0/2)} + x^{(1/2)} \tag{21.4-1a}$$
$$d := x^{(0/2)} - x^{(1/2)} \tag{21.4-1b}$$

Then the cyclic auto convolution of the sequence x can be obtained by two half-length convolutions of s and d as

$$x \circledast x = \frac{1}{2}\,[s \circledast s + d \circledast_- d, \quad s \circledast s - d \circledast_- d] \tag{21.4-2}$$

where the symbols $\circledast$ and $\circledast_-$ stand for cyclic and negacyclic convolution, respectively (see section 21.3 on page 417). The equivalent formula for the cyclic convolution of two sequences x and y is

$$x \circledast y = \frac{1}{2}\,[s_x \circledast s_y + d_x \circledast_- d_y, \quad s_x \circledast s_y - d_x \circledast_- d_y] \tag{21.4-3}$$

where

$$s_x := x^{(0/2)} + x^{(1/2)} \tag{21.4-4a}$$
$$d_x := x^{(0/2)} - x^{(1/2)} \tag{21.4-4b}$$
$$s_y := y^{(0/2)} + y^{(1/2)} \tag{21.4-4c}$$
$$d_y := y^{(0/2)} - y^{(1/2)} \tag{21.4-4d}$$

Now use the fact that an linear convolution is computed by a cyclic convolution of zero-padded sequences whose upper halves are simply zero, so $s_x = d_x = x$ and $s_y = d_y = y$. Then relation 21.4-3 reads:

$$x \circledast_{lin} y = \frac{1}{2}\,[x \circledast y + x \circledast_- y, \quad x \circledast y - x \circledast_- y] \tag{21.4-5}$$

And for the acyclic auto convolution:

$$x \circledast_{lin} x = \frac{1}{2}\,[x \circledast x + x \circledast_- x, \quad x \circledast x - x \circledast_- x] \tag{21.4-6}$$

The lower and upper halves of the linear convolution can be obtained from the sum and difference of the cyclic and the negacyclic convolution.

The case $R = 3$

Let $\omega = \frac{1}{2}\,(1 + i\,\sqrt{3})$ and define

$$A := x^{(0/3)} + x^{(1/3)} + x^{(2/3)} \tag{21.4-7a}$$
$$B := x^{(0/3)} + \omega\ x^{(1/3)} + \omega^2\,x^{(2/3)} \tag{21.4-7b}$$
$$C := x^{(0/3)} + \omega^2\,x^{(1/3)} + \omega\ x^{(2/3)} \tag{21.4-7c}$$

Then, if $h := x \circledast x$, one has

$$h^{(0/3)} = A \circledast A + \quad B \circledast_{\{\omega\}} B + \quad C \circledast_{\{\omega^2\}} C \tag{21.4-8a}$$

$$h^{(1/3)} = A \circledast A + \omega^2 B \circledast_{\{\omega\}} B + \omega \; C \circledast_{\{\omega^2\}} C \tag{21.4-8b}$$

$$h^{(2/3)} = A \circledast A + \omega \; B \circledast_{\{\omega\}} B + \omega^2 C \circledast_{\{\omega^2\}} C \tag{21.4-8c}$$

For real valued data C is the complex conjugate (*cc.*) of B and (with $\omega^2 = cc.\omega$) $B \circledast_{\{\omega\}} B$ is the *cc.* of $C \circledast_{\{\omega^2\}} C$ and therefore every $B \circledast_{\{.\}} B$-term is the cc. of the $C \circledast_{\{.\}} C$-term in the same line. Is there a nice and general scheme for real valued convolutions based on the MFA? Read on for the positive answer.

21.4.1 Convolution of real valued data using the MFA

Consider the MFA-algorithm for the cyclic convolution as given on page 419 but with real input data: for row 0 which is real after the column FFTs one needs to compute the usual cyclic convolution; for row $R/2$ which is also purely real after the column FFTs a negacyclic convolution is needed[1], the code for negacyclic convolution is given on page 496.

All other weighted convolutions involve complex computations, but it is easy to see how to reduce the work by 50 percent: as the result must be real the data in row number $R - r$ must, because of the symmetries of the real and imaginary part of the (inverse) Fourier transform of real data, be the complex conjugate of the data in row r. Therefore one can use real FFTs (R2CFTs) for all column-transforms for step 1 and half-complex to real FFTs (C2RFTs) for step 3.

Let the computational cost of a cyclic (real) convolution be q, then

- For even values of R one must perform one cyclic (row 0), one negacyclic (row $R/2$) and $R/2 - 2$ complex weighted convolutions (rows $1, 2, \ldots, R/2 - 1$)
- For R odd one must perform 1 cyclic (row 0) and $(R - 1)/2$ complex weighted convolutions (rows $1, 2, \ldots, (R - 1)/2$)

Now assume, slightly simplifying, that the cyclic and the negacyclic real convolution involve the same number of computations and that the cost of a weighted complex convolution is twice that. Then in both cases above the total work is exactly half of that for the complex case, which is what one expects from a real world real valued convolution algorithm.

For the computation of the linear convolution one can use the right angle convolution (and complex FFTs in the column passes), see section 21.3 on page 417.

21.4.2 Mass storage convolution using the MFA

Algorithms on data sets that do not fit into physical RAM are sometimes called *external* or *out of core* algorithms. Simply using the virtual memory mechanism of the operating system is not an option, eternal hard disk activity would be the consequence in most cases. We give a method for the *mass storage convolution*. It is based on the matrix FFT convolution algorithm given on page 419. The number of disk seeks has to be kept minimal because these are slow operations which degrade performance unacceptably if they occur too often,

The crucial modification of the use of the MFA is to *not* choose R and C as close as possible to $\sqrt{n}$ as is usually done. Instead one chooses R to be minimal so that the row length C corresponds to the biggest data set that fits into the available RAM. We now analyze how the number of seeks depends on the choice of R and C: In what follows it is assumed that the data lies in memory as row_0, row_1, ..., row_{R-1}. In other words, the elements of each rows lie contiguous in memory. Further let $\alpha \geq 2$ be the number of times the data set exceeds the available RAM size.

[1] For odd values of R there is no such row and no negacyclic convolution is needed.

In step 1 and 3 of the convolution algorithm given on page 419 one reads from disk (row by row, involving R seeks) the number of columns that just fit into RAM, does the (many, short) column-FFTs, writes back (again R seeks), and proceeds to the next block; this happens for α of these blocks, giving a total of $4\,\alpha\,R$ seeks for steps 1 and 3.

In step 2 one reads (α times) blocks of one or more rows, which lie in contiguous portions of the disk, perform the FFT on the rows, and write back to disk. This gives a total of $2\,\alpha$ seeks for step 2.

Thereby there are $2\,\alpha + 4\,\alpha\,R$ seeks during the whole computation, which is minimized by the choice of maximal C. This means that one chooses a shape of the matrix so that the rows are as big as possible subject to the constraint that they have to fit into main memory, which in turn means there are $R = \alpha$ rows, leading to an optimal seek count of $K = 2\,\alpha + 4\,\alpha^2$.

Let S_D be the seek time of the hard disk in seconds, W_D be the disk transfer rate in megabyte per second. Further let A be the amount of available RAM in megabytes. Then the total time spent for disk operations is the sums of the time spent in seeks and the time for reading and writing:

$$T = K\,S_D + 6\,\frac{\alpha\,A}{W_D} \tag{21.4-9a}$$

$$= \left(2\,\alpha + 4\,\alpha^2\right)\,S_D + 6\,\frac{\alpha\,A}{R_D} \tag{21.4-9b}$$

We give two examples for an FFT whose size that exceeds the available RAM by a factor of $\alpha = 16$ (so the number of seeks is $k = 1056$).

With a machine where the disk seek takes 10 milliseconds ($S_D = 0.010$) about 10 seconds are needed for the seeks. Further assume we have $A = 64$ Megabytes of RAM available so the transform size is 1 GB corresponding to 128 million (double precision) floats. The disk transfer rate shall be $W_D = 10$ MB/sec. Then the overhead for the read and write would amount to $6 \cdot 1024MB/(10MB/sec) \approx 615sec$ or approximately 10 minutes. So the total disk activity takes approximately 10.5 minutes.

With a workstation machine with $S_D = 0.005$ sec (5 milliseconds), $W_d = 50$ MB/sec, and 512 MB of RAM: the transform size is 8 GB (1 G doubles) and we obtain $T = 5.28{+}983.04 = 988.32$ or approximately 16.5 minutes.

In both cases the CPU will be busy for several minutes which is in the same order as the time for the disk activity. Therefore, when multi-threading is available, one may want to use a *double buffer* variant: Choose the row length so that it fits twice into the RAM workspace; then let always one (CPU-intensive) thread do the FFTs in one of the scratch spaces and the other (hard disk intensive) thread write back the data from the other scratch space and read the next data to be processed. With not too small main memory (and not too slow hard disk) and some fine tuning double buffering can to keep the CPU busy during much of the hard disk operations. Thereby most of the disk time can be 'hidden' as the CPU activity does not stall waiting for data.

The mass storage convolution as described was used for the calculation of the number

$$9^{9^9} \approx 0.4281247 \cdot 10^{369,693,100} \tag{21.4-10}$$

on a 32-bit machine in 1999. The computation used two files of size 2 Gigabytes each and took less than eight hours on a system with a AMD K6/2 CPU at 366 MHz with 66 MHz memory. The log-file of the computation is [hfloat: examples/run1-pow999.txt]. The computation did not use the double-buffer technique.

21.5 The z-transform (ZT)

In this section we will learn a technique to compute the Fourier transform by a cyclic convolution. In fact, the transform computed is the z-transform, a more general transform that in a special case is identical

to the Fourier transform.

The discrete z-transform (ZT) of a length-n sequence a is a length-n sequence c defined by

$$c = \mathcal{Z}[a] \tag{21.5-1a}$$

$$c_k := \sum_{x=0}^{n-1} a_x \, z^{k\,x} \tag{21.5-1b}$$

The z-transform is a linear transformation. It is not an orthogonal transformation unless z is a root of unity. For $z = e^{\pm 2\,\pi\,i/n}$ the z-transform specializes to the discrete Fourier transform. An important property is the convolution property:

$$\mathcal{Z}[a \circledast b] = \mathcal{Z}[a]\;\mathcal{Z}[b] \tag{21.5-2}$$

Convolution in original space corresponds to ordinary (element-wise) multiplication in z-space. This can be turned into an efficient convolution algorithm for the special case of the Fourier transform but not in general because no efficient algorithm for the inverse transform is known.

21.5.1 Computation via convolution (Bluestein's algorithm)

Noting that

$$x\,k = \frac{1}{2}\left(x^2 + k^2 - (k-x)^2\right) \tag{21.5-3}$$

we find, for element c_k of the Fourier transform of the sequence a,

$$c_k = \sum_{x=0}^{n-1} a_x \, z^{x\,k} = z^{k^2/2}\left[\sum_{x=0}^{n-1}\left(a_x \, z^{x^2/2}\right)\, z^{-(k-x)^2/2}\right] \tag{21.5-4}$$

The expression in brackets is a cyclic convolution of the sequence $a_x \, z^{x^2/2}$ with the sequence $z^{-x^2/2}$.

This leads to the algorithm for the *chirp z-transform*:

1. Multiply the sequence a element-wise with $z^{x^2/2}$.
2. Convolve the resulting sequence with the sequence $z^{-x^2/2}$.
3. Multiply element-wise with the sequence $z^{k^2/2}$.

The above algorithm constitutes a fast algorithm for the ZT because fast convolution is possible via FFT. The idea is due to Bluestein [44], a detailed description of the algorithm for computing the Bluestein FFT is given in [224].

21.5.2 Arbitrary length FFT by ZT

The length n of the input sequence a for the fast z-transform is not limited to highly composite values: For values of n where a FFT is not feasible pad the sequence with zeros up to a length L with $L >= 2\,n$ such that a length-L FFT can be computed (highly composite L, for example a power of two).

As the Fourier transform is the special case $z = e^{\pm 2\,\pi\,i/n}$ of the ZT the chirp-ZT algorithm constitutes an FFT algorithm for sequences of arbitrary length.

The transform takes a few times more than an direct FFT. The worst case (if only FFTs for n a power of 2 are available) is $n = 2^p + 1$: one must perform three FFTs of length $L = 2^{p+2} \approx 4\,n$ for the computation of the convolution. So the total work amounts to about 12 times the work a FFT of length $n = 2^p$ would

cost. It is possible to lower this 'worst case factor' to 6 by using highly composite L slightly greater than $2\,n$.

For multiple computations of z-transforms of the same length one may want to store the Fourier transform of the sequence $z^{k^2/2}$ as it does not change. Thereby the worst case is reduced to a factor 4 with highly composite FFTs and 8 if FFTs are available for powers of two only.

A C++ implementation of the Fourier transform for sequences of arbitrary length is given in [FXT: chirpzt/fftarblen.cc]. For even length, the routine is

```
static void
fft_arblen_even(Complex *x, ulong n, int is)
// Arbitrary length FFT for even lengths n.
{
    ulong ldnn = ld(n);
    if ( n==(1UL<<ldnn) )  ldnn += 1;
    else                   ldnn += 2;
    ulong nn = (1UL<<ldnn);

    Complex *f = new Complex[nn];
    ::copy0(x, n, f, nn);

    Complex *w = new Complex[nn];
    make_fft_chirp(w, n, nn, is);
    multiply(f, n, w);

    double *dw = (double *)w;
    for (ulong k=1; k<2*n; k+=2)  dw[k] = -dw[k];  // =^= make_fft_chirp(w, n, nn, -is);

    fft_complex_convolution(w, f, ldnn);
    add(f, n, f+n);  // need cyclic convolution

    make_fft_chirp(w, n, nn, is);
    multiply(w, n, f);

    ::copy(w, x, n);
    delete [] w;
    delete [] f;
}
```

The auxiliary routine `make_fft_chirp()` is defined in [FXT: chirpzt/makechirp.cc]:

```
void
make_fft_chirp(Complex *w, ulong n, ulong nn, int is)
// For k=0..n-1:   w[k] := exp( is * k*k * (i*2*PI/n)/2 )
//   where i = sqrt(-1)
// For k=n..nn-1:  w[k] = 0
{
    double phi = 1.0*is*M_PI/n;  // == (i*2*Pi/n)/2
    ulong k2 = 0,  n2 = 2*n;
    for (ulong k=0; k<n; ++k)
    {
        w[k] = SinCos(phi*k2);
        k2 += (2*k+1);
        if ( k2>n2 )  k2 -= n2;
        // here:  k2 == (k*k) mod 2*n;
    }
    null(w+n, nn-n);
}
```

For transform of odd length, use

```
static void
fft_arblen_odd(Complex *x, ulong n, int is)
// Arbitrary length FFT for odd lengths n.
{
    ulong ldnn = ld(n);  // == floor(log2(n))
    ldnn += 3;
    ulong nn = (1UL<<ldnn);  // == 4 * next_2_pow(n)

    Complex *f = new Complex[nn];
    ::copy0(x, n, f, nn);

    Complex *w = new Complex[nn];
    make_fft_chirp_n2(w, n, nn, is);
    multiply(f, n, w);

    double *dw = (double *)w;  // cast
```

```
    for (ulong k=1; k<nn; k+=2)  dw[k] = -dw[k];  // =^= make_fft_chirp_n2(w, n, nn, -is);
    fft_complex_convolution(w, f, ldnn);
    ulong n2 = n*2;
    ulong dn = nn-n2;
    if ( dn>n )  dn = n;
    add(f, dn, f+n2);  // need cyclic convolution
    make_fft_chirp(w, n, nn, is);
    multiply(w, n, f);
    ::copy(w, x, n);
    delete [] w;
    delete [] f;
}
```

The auxiliary routine `make_fft_chirp_n2()` is

```
void
make_fft_chirp_n2(Complex *w, ulong n, ulong nn, int is)
// For k=0..2*n-1:   w[k] = exp( is * k*k * (i*2*PI/n)/2 )
//   where i = sqrt(-1)
// For k=2*n..nn-1:  w[k] = 0
{
    double phi = 1.0*is*M_PI/n;  // == (i*2*Pi/n)/2
    ulong n2 = n*2;
    ulong k2 = 0;
    for (ulong k=0; k<n2; ++k)
    {
        w[k] = SinCos(phi*k2);
        k2 += (2*k+1);
        if ( k2>n2 )  k2 -= n2;
        // here:  k2 == (k*k) mod 2*n;
    }
    null(w+n2, nn-n2);
}
```

The routine to be called by the user is

```
void
fft_arblen(Complex *x, ulong n, int is)
// Arbitrary length FFT.
{
    if ( n&1 )  fft_arblen_odd(x, n, is);
    else        fft_arblen_even(x, n, is);
}
```

21.5.3 Fractional Fourier transform by ZT

The z-transform with $z = e^{\alpha\, 2\,\pi\, i/n}$ is called the *fractional Fourier transform*. For $\alpha = \pm 1$ one again obtains the usual Fourier transform. The fractional Fourier transform can be used for the computation of the Fourier transform of sequences with only few nonzero elements and for the exact detection of frequencies that are not integer multiples of the lowest frequency of the DFT. A discussion of the fractional Fourier transform can be found in [24].

A C++ implementation of the fractional Fourier transform for sequences of arbitrary length is given in [FXT: chirpzt/fftfract.cc]:

```
void
fft_fract(Complex *x, ulong n, double v)
// Fractional (fast) Fourier transform.
{
    ulong ldnn = ld(n);
    if ( n==(1UL<<ldnn) )  ldnn += 1;
    else                   ldnn += 2;
    ulong nn = (1UL<<ldnn); // smallest power of 2 >= 2*n
    Complex *f = new Complex[nn];
    copy(x, f, n);
    null(f+n, nn-n);
    Complex *w = new Complex[nn];
    make_fft_fract_chirp(w, v, n, nn);
    for (ulong j=0; j<n; ++j)  f[j] *= w[j];
    for (ulong j=0; j<nn; ++j)  w[j] = conj(w[j]);
```

```
    fft_complex_convolution(w, f, ldnn);
    make_fft_fract_chirp(w, v, n, nn);
    for (ulong j=0; j<n; ++j)  w[j] *= f[j];
    copy(w+n, x, n);
    delete [] w;
    delete [] f;
}
```

The auxiliary routine `make_fft_fract_chirp()` is defined in [FXT: chirpzt/makechirp.cc]:

```
void
make_fft_fract_chirp(Complex *w, double v, ulong n, ulong nn)
// For k=0..nn-1:   w[k] == exp(v*sqrt(-1)*k*k*2*pi*/n/2)
{
    const double phi = v*2.0*M_PI/n/2;
    ulong n2 = 2*n;
    ulong np=0;
    for (ulong k=0; k<nn; ++k)
    {
        w[k] = SinCos(phi*np);
        np += ((k<<1)+1);  // np == (k*k)%n2
        if ( np>=n2 )  np -= n2;
    }
}
```

21.6 Prime length FFTs

For the computation of FFTs for sequences whose length is prime we can exploit the existence of primitive roots. We will be able to express the transform of all but the first element as a cyclic convolution of two sequences whose length is reduced by one.

Let p be prime, then an element g exists so that the least positive exponent e so that $g^e \equiv 1 \bmod p$ is $e = p-1$. The element g is called a *generator* (or *primitive root*) modulo p (see section 37.5 on page 741). Every nonzero element modulo p can uniquely be expressed as a power g^e where $0 \le e < p-1$. For example, a generator modulo $p = 11$ is $g = 2$, its powers are:

$$g^0 \equiv 1 \quad g^1 \equiv 2 \quad g^2 \equiv 4 \quad g^3 \equiv 8 \quad g^4 \equiv 5 \quad g^5 \equiv 10 \equiv -1 \quad g^6 \equiv 9 \quad g^7 \equiv 7 \quad g^8 \equiv 3 \quad g^9 \equiv 6 \quad g^{p-1} \equiv 1$$

Likewise, we can express any nonzero element as a negative power of g. Let $h = g^{-1}$, then with our example, $h \equiv 6$ and

$$h^0 \equiv 1 \quad h^1 \equiv 6 \quad h^2 \equiv 3 \quad h^3 \equiv 7 \quad h^4 \equiv 9 \quad h^5 \equiv 10 \equiv -1 \quad h^6 \equiv 5 \quad h^7 \equiv 8 \quad h^8 \equiv 4 \quad h^9 \equiv 2 \quad h^{p-1} \equiv 1$$

This is just the reversed sequence of values. Let the c be the Fourier transform of length-p sequence a:

$$c_k \;=\; \sum_{x=0}^{p-1} a_x \, z^{\sigma\, x\, k} \tag{21.6-1}$$

where $z = \exp(2\,i\,\pi/p)$ and $\sigma = \pm 1$ is the sign of the transform. We split the computation of the Fourier transform in two parts, we compute the first element of the transform as

$$c_0 \;=\; \sum_{x=0}^{p-1} a_x \tag{21.6-2}$$

Now it remains to compute, for $1 \le k \le p-1$,

$$c_k \;=\; a_0 + \sum_{x=1}^{p-1} a_x \, z^{\sigma\, x\, k} \tag{21.6-3}$$

Note the lower index of the sum. We write $k \equiv g^e$ and $x \equiv g^{-f}$ (modulo p), so

$$c_{(g^e)} - a_0 \;=\; \sum_{f=0}^{p-2} a_{(g^{-f})} \, z^{\sigma\,(g^{-f})\,(g^e)} \;=\; \sum_{f=0}^{p-2} a_{(g^{-f})} \, z^{\sigma\,(g^{e-f})} \tag{21.6-4}$$

The sum is a cyclic convolution of the sequences $z^{(g^w)}$ and $a_{(g^{-w})}$ where $0 \le w \le p-2$. That is, if we permute the sequences a_1, a_2, ..., a_{p-1} and z^1, z^2, ..., z^{p-1} and compute their cyclic convolution then we obtain a permutation of the sequence c_1, c_2, ..., c_{p-1}.

The method was given in [196], it is called *Rader's algorithm*. We implement it in pari/gp:

```
ft_rader(a, is=+1)=
\\ Fourier transform for prime lengths (Rader's algorithm)
{
    local(n, a0, f0, g, w);
    local(f, ixp, ixm, pa, pw, t);
    n = length(a);
    a0 = a[1];  f0 = sum(j=1, n, a[j]);
    g = znprimroot(n);  ixp = vector(n, j, component( g^(j-1), 2) );
    w = is*2*I*Pi/n;  pw = vector(n-1, j, exp(w*ixp[j]) );
    pa = vector(n-1);  for (j=1, n-1, pa[j]=a[1+ixp[1+n-j]] );
    t = cconv(pa, pw);  \\ cyclic convolution
    f = vector(n);  f[1] = f0;  for (k=1, n-1, f[1+k]=t[k]+a0);
    t = vector(n);  t[1] = f[1];  for (k=2, n, t[1+ixp[k-1]]=f[k]);
    return( t );
}
```

With a (slow) implementation of the cyclic convolution and DFT we can check whether the method works by comparing the results:

```
cconv(a, b, w=+1)=
\\ weighted cyclic convolution (by definition, n^2 operations)
\\ w==+1 ==> usual cyclic convolution
{
    local(n, f, s, k, k2);
    n = length(a);
    f = vector(n);
    for (tau=0, n-1,  \\ tau = k + k2
        s0 = 0;  k = 0;  k2 = tau;
        while (k<=tau, s0 += (a[k+1]*b[k2+1]);  k++; k2--);
        s1 = 0;  k2 = n-1; \\ k=tau+1
        while (k<n,    s1 += (a[k+1]*b[k2+1]);  k++; k2--);
        f[tau+1] = s0 + w * s1;
    );
    return( f );
}
```

```
dft(a, is=+1)=
\\ Fourier transform (by definition, n^2 operations)
{
    local(n, f, s, ph0, ph);
    n = length(a);
    f = vector(n);
    ph0 = is*2*Pi*I/n;
    for (k=0, n-1,
        ph = ph0 * k;
        f[k+1] = sum (x=0, n-1,  a[x+1] * exp(ph*x) );
    );
    return( f );
}
```

In order to turn the algorithm into a fast Fourier transform we need to compute the convolution via length-$(p-1)$ (fast) transforms. This is trivially possible when $p-1 = 2^q$, for example when $p = 5$ or $p = 17$. As $p-1$ is always divisible by two, we can split at least once. For $p = 11$ we have $(p-1)/2 = 5$ so we can again use Rader's algorithm and length-4 transforms.

The method can be used to generate code for short (prime) length FFTs. It is advisable to create the permuted and transformed sequence of the powers of z. Thereby only two length-$(p-1)$ FFTs will be needed for a length-p transform.

Chapter 22

The Walsh transform and its relatives

We describe several variants of the *Walsh transform*, sometimes called *Walsh-Hadamard transform* or just *Hadamard transform*. The Walsh transform has the same complexity as the Fourier transform but does not involve any multiplications. In fact, one can obtain a Walsh transform routine by removing all multiplications (with sines and cosines) in a given FFT routine.

We also give related transforms like the slant transform and the Reed-Muller transform. The dyadic convolution that can be computed efficiently by the Walsh transform is introduced.

22.1 The Walsh transform: Walsh-Kronecker basis

How to make a Walsh transform out of your FFT:
'*Replace exp(something) by* 1*, done.*'

Removing all *exp*(something) from the radix-2, decimation in time Fourier transform we obtain

```
void slow_walsh_wak_dit2(double *f, ulong ldn)
// (this routine has a problem)
{
    ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1<<ldm);
        const ulong mh = (m>>1);
        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r+j;
                const ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u+v;
                f[t2] = u-v;
            }
        }
    }
}
```

The transform involves proportional $n \log_2(n)$ additions (and subtractions) and no multiplication at all. The transform, as given, is its own inverse up to a factor $1/n$. The Walsh transform of integer input is integral.

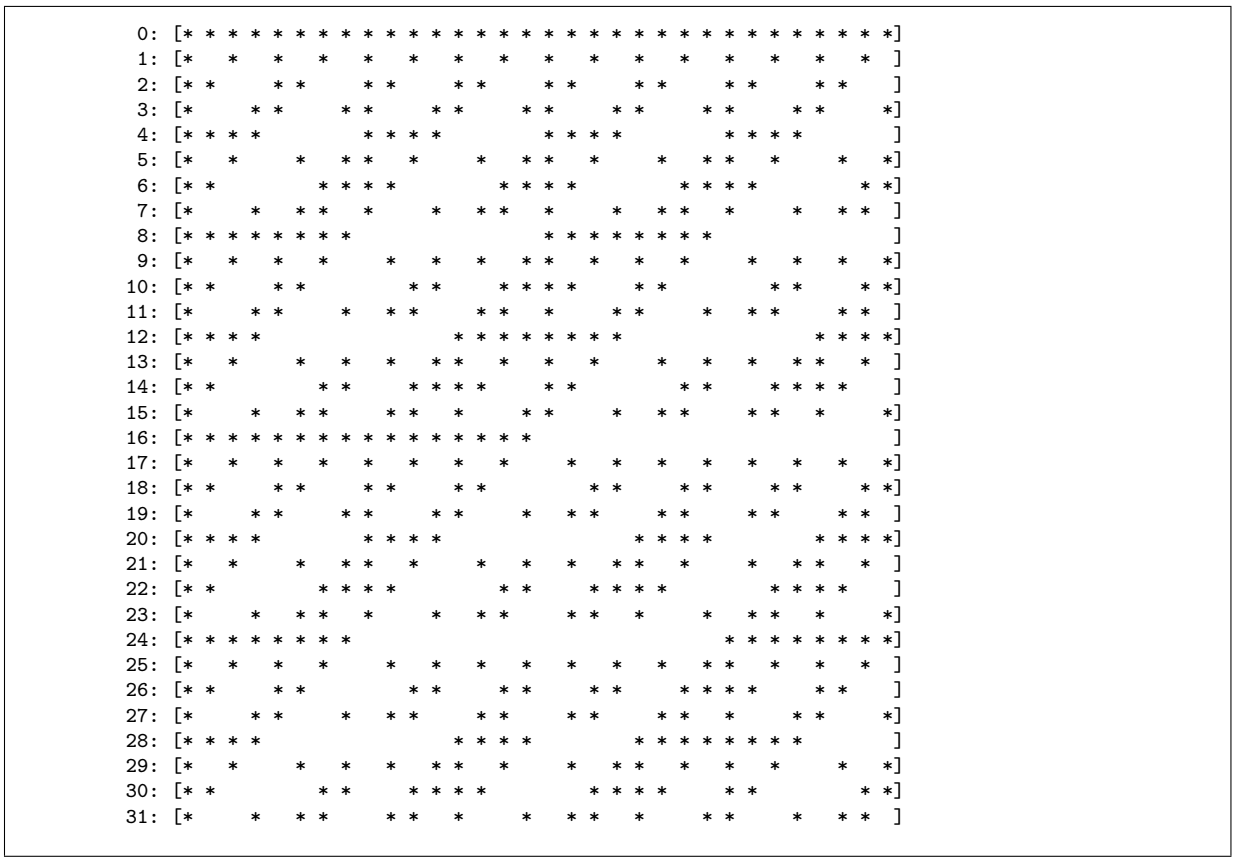


Figure 22.1-A: Basis functions for the Walsh transform (Walsh-Kronecker basis). Asterisks denote the value $+1$, blank entries denote -1.

As the `slow` in the name shall suggest, the implementation has a problem as given. The memory access pattern is highly non-local. Let's make a slight improvement: here we just took the radix-2 DIT FFT code from section 20.3.1.3 on page 380 and threw away all trigonometric computations (and multiplications). But the swapping of the inner loops, that we did for the FFT in order to save trigonometric computations is now of no advantage anymore. So we try the following [FXT: `walsh_wak_dit2()` in walsh/walshwak2.h]:

```
template <typename Type>
void walsh_wak_dit2(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-2 decimation in frequency (DIF) algorithm.
{
    ulong n = (1UL<<ldn);
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

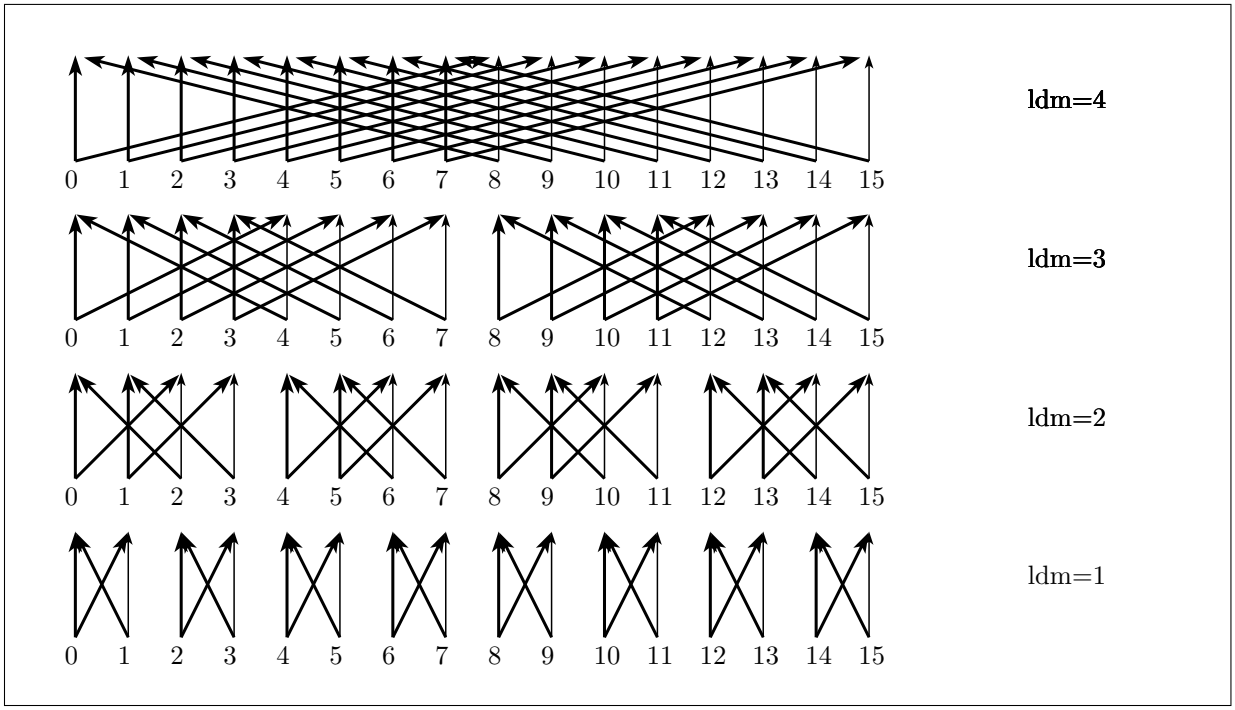


Figure 22.1-B: Data flow for the length-16, radix-2, decimation in time (DIT) transform. The stages are from bottom to top. Thin lines indicate a factor of minus one.

The performance impact is quite drastic. For $n = 2^{21}$ (and type double, 16 MByte of memory) it gives a speedup by a factor of about *eight*. For smaller lengths the ratio approaches one.

The data flow diagram (*butterfly diagram*) for the radix-2 decimation in time (DIT) algorithm is shown in figure 22.1-B. The figure was created with the program [FXT: fft/butterfly-texpic-demo.cc]. The diagram for the decimation in frequency (DIF) algorithm is obtained by reversing the order of the steps. In the code, only the outermost loop has to be changed [FXT: `walsh_wak_dif2()` in walsh/walshwak2.h]:

```
template <typename Type>
void walsh_wak_dif2(Type *f, ulong ldn)
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

A function that computes the k-th base function of the transform is [FXT: `walsh_wak_basefunc()` in walsh/walshbasefunc.h]:

```
template <typename Type>
void walsh_wak_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
```

```
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

The basis functions are shown in figure 22.1-A. Note that the lowest row is (the signed version of) the *Thue-Morse sequence*, see section 1.15.1 on page 37.

Multi-dimensional Walsh transform

If one applies the row-column algorithm (see section 20.10.2 on page 405) to compute a two-dimensional $n \times m$ Walsh transform then the result is exactly the same as with a 1-dimensional $n \cdot m$ transform. That is, algorithmically nothing needs to be done for multidimensional Walsh transforms: a k-dimensional $n_1 \times n_2 \times \ldots \times n_k$-transform is identical to a 1-dimensional $n_1 \cdot n_2 \cdot \ldots \cdot n_k$-transform. The length-$2^n$ Walsh transform is identical to a n-dimensional length-2 Fourier transform.

22.2 Eigenvectors of the Walsh transform *

```
 0:  [ +5 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 ]
 1:  [ +1 +3 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 ]
 2:  [ +1 +1 +3 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 ]
 3:  [ +1 -1 -1 +5 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 ]
 4:  [ +1 +1 +1 +1 +3 -1 -1 -1 +1 +1 +1 +1 -1 -1 -1 -1 ]
 5:  [ +1 -1 +1 -1 -1 +5 -1 +1 +1 -1 +1 -1 -1 +1 -1 +1 ]
 6:  [ +1 +1 -1 -1 -1 -1 +5 +1 +1 +1 -1 -1 -1 -1 +1 +1 ]
 7:  [ +1 -1 -1 +1 -1 +1 +1 +3 +1 -1 -1 +1 -1 +1 +1 -1 ]
 8:  [ +1 +1 +1 +1 +1 +1 +1 +1 -5 -1 -1 -1 -1 -1 -1 -1 ]
 9:  [ +1 -1 +1 -1 +1 -1 +1 -1 -1 -3 -1 +1 -1 +1 -1 +1 ]
10:  [ +1 +1 -1 -1 +1 +1 -1 -1 -1 -1 -3 +1 -1 -1 +1 +1 ]
11:  [ +1 -1 -1 +1 +1 -1 -1 +1 -1 +1 +1 -5 -1 +1 +1 -1 ]
12:  [ +1 +1 +1 +1 -1 -1 -1 -1 -1 -1 -1 -1 -3 +1 +1 +1 ]
13:  [ +1 -1 +1 -1 -1 +1 -1 +1 -1 +1 -1 +1 +1 -5 +1 -1 ]
14:  [ +1 +1 -1 -1 -1 -1 +1 +1 -1 -1 +1 +1 +1 +1 -5 -1 ]
15:  [ +1 -1 -1 +1 -1 +1 +1 -1 -1 +1 +1 -1 +1 -1 -1 -3 ]
```

Figure 22.2-A: Eigenvectors of the length-16 Walsh transform (Walsh-Kronecker basis) as row vectors. The eigenvalues are $+1$ for the vectors $0 \ldots 7$ and -1 for the vectors $8 \ldots 16$. Linear combinations of vectors with the same eigenvalue e are again eigenvectors with eigenvalue e.

The Walsh transforms are self-inverse, so their eigenvalues can only be plus or minus one. Let a be a sequence and let $W(a)$ denote the Walsh transform of a. Set

$$u_+ \quad := \quad W(a) + a \tag{22.2-1}$$

Then

$$W(u_+) \quad = \quad W(W(a)) + W(a) \;=\; a + W(a) \;=\; +1 \cdot u_+ \tag{22.2-2}$$

That is, u_+ is an eigenvector of W with eigenvalue $+1$. Equivalently, $u_- := W(a) - a$ is an eigenvector with eigenvalue -1. Thereby, two eigenvectors can be obtained from an arbitrary nonzero sequence.

We are interested in a simple routine that for a Walsh transform of length n gives a set of n eigenvectors that span the n-dimensional space. With a routine that computes the k-th basis function of the transform we can obtain an eigenvector efficiently by simply adding a delta peak at position k to the basis function. The delta peak has to be scaled according to whether a positive or negative eigenvalue is desired and according to the normalization of the transform.

A suitable routine for the Walsh-Kronecker basis (whose basis functions are given in figure 22.1-A on page 430) is

```
void
walsh_wak_eigen(double *v, ulong ldn, ulong k)
// Eigenvectors of the Walsh transform (walsh_wak).
// Eigenvalues are +1 if k<n/2, else -1
{
    ulong n = 1UL << ldn;
    walsh_wak_basefunc(v, n, k);
    double d = sqrt(n);
    v[k] += (k<n/2 ? +d : -d);
}
```

This routine is given in [FXT: walsh/walsheigen.cc]. Figure 22.2-A was created with the program [FXT: fft/walsh-eigenvec-demo.cc].

Note that with the unnormalized transforms the eigenvalues are $\pm\sqrt{n}$.

22.3 The Kronecker product

The length-2 Walsh transform is equivalent to the multiplication of a 2-component vector by the matrix

$$\mathbf{W}_2 \;=\; \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \tag{22.3-1}$$

The length-4 Walsh transform corresponds to

$$\mathbf{W}_4 \;=\; \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} \tag{22.3-2}$$

One might be tempted to write

$$\mathbf{W}_4 \;=\; \begin{bmatrix} +\mathbf{W}_2 & +\mathbf{W}_2 \\ +\mathbf{W}_2 & -\mathbf{W}_2 \end{bmatrix} \tag{22.3-3}$$

This idea can indeed be turned into a well-defined notation which is quite powerful when dealing with orthogonal transforms and their fast algorithms. Let $\mathbf{A}$ be an $m \times n$ matrix

$$\mathbf{A} \;=\; \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \tag{22.3-4}$$

then the (right) *Kronecker product* (or *tensor product*) with a matrix $\mathbf{B}$ is

$$\mathbf{A} \otimes \mathbf{B} \;:=\; \begin{bmatrix} a_{0,0}\mathbf{B} & a_{0,1}\mathbf{B} & \cdots & a_{0,n-1}\mathbf{B} \\ a_{1,0}\mathbf{B} & a_{1,1}\mathbf{B} & \cdots & a_{1,n-1}\mathbf{B} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0}\mathbf{B} & a_{m-1,1}\mathbf{B} & \cdots & a_{m-1,n-1}\mathbf{B} \end{bmatrix} \tag{22.3-5}$$

There is no restriction on the dimensions of $\mathbf{B}$. If $\mathbf{B}$ is a $r \times s$ matrix then the dimensions of the given Kronecker product is $(mr) \times (ns)$, and $c_{k+ir,l+js} = a_{i,j} b_{k,l}$. The Kronecker product is not commutative, that is, $\mathbf{A} \otimes \mathbf{B} \neq \mathbf{B} \otimes \mathbf{A}$ in general.

For a scalar factor α the following relations are immediate:

$$(\alpha \mathbf{A}) \otimes \mathbf{B} = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{22.3-6a}$$
$$\mathbf{A} \otimes (\alpha \mathbf{B}) = \alpha(\mathbf{A} \otimes \mathbf{B}) \tag{22.3-6b}$$

The next relations are the same as for the ordinary matrix product. Distributivity (the matrices on both sides of a plus sign must be of the same dimensions):

$$(\mathbf{A} + \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes \mathbf{C} + \mathbf{B} \otimes \mathbf{C} \tag{22.3-7a}$$
$$\mathbf{A} \otimes (\mathbf{B} + \mathbf{C}) = \mathbf{A} \otimes \mathbf{B} + \mathbf{A} \otimes \mathbf{C} \tag{22.3-7b}$$

Associativity:

$$\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} \tag{22.3-8}$$

The matrix product (indicated by a dot) of Kronecker products can be rewritten as

$$(\mathbf{A} \otimes \mathbf{B}) \cdot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A} \cdot \mathbf{C}) \otimes (\mathbf{B} \cdot \mathbf{D}) \tag{22.3-9a}$$
$$(\mathbf{L}_1 \otimes \mathbf{R}_1) \cdot (\mathbf{L}_2 \otimes \mathbf{R}_2) \cdot \ldots \cdot (\mathbf{L}_n \otimes \mathbf{R}_n) = (\mathbf{L}_1 \cdot \mathbf{L}_2 \cdot \ldots \cdot \mathbf{L}_n) \otimes (\mathbf{R}_1 \cdot \mathbf{R}_2 \cdot \ldots \cdot \mathbf{R}_n) \tag{22.3-9b}$$

Set $\mathbf{L}_1 = \mathbf{L}_2 = \ldots = \mathbf{L}_n =: \mathbf{L}$ and $\mathbf{R}_1 = \mathbf{R}_2 = \ldots = \mathbf{R}_n =: \mathbf{R}$ in the latter relation to obtain

$$(\mathbf{L} \otimes \mathbf{R})^n = \mathbf{L}^n \otimes \mathbf{R}^n \tag{22.3-9c}$$

The Kronecker product of matrix products can be rewritten as

$$(\mathbf{A} \cdot \mathbf{B}) \otimes (\mathbf{C} \cdot \mathbf{D}) = (\mathbf{A} \otimes \mathbf{C}) \cdot (\mathbf{B} \otimes \mathbf{D}) \tag{22.3-10a}$$
$$(\mathbf{L}_1 \cdot \mathbf{R}_1) \otimes (\mathbf{L}_2 \cdot \mathbf{R}_2) \otimes \ldots \otimes (\mathbf{L}_n \cdot \mathbf{R}_n) = (\mathbf{L}_1 \otimes \mathbf{L}_2 \otimes \ldots \otimes \mathbf{L}_n) \cdot (\mathbf{R}_1 \otimes \mathbf{R}_2 \otimes \ldots \otimes \mathbf{R}_n) \tag{22.3-10b}$$

Here the matrices left and right from a dot must be compatible for ordinary matrix multiplication.

One has

$$(\mathbf{A} \otimes \mathbf{B})^T = \mathbf{A}^T \otimes \mathbf{B}^T \tag{22.3-11a}$$
$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{B}^{-1} \tag{22.3-11b}$$

If $\mathbf{A}$ and $\mathbf{B}$ are respectively $m \times n$ and $r \times s$ matrices then

$$\mathbf{A} \otimes \mathbf{B} = (\mathbf{I}_m \otimes \mathbf{B}) \cdot (\mathbf{A} \otimes \mathbf{I}_s) \tag{22.3-12a}$$
$$= (\mathbf{A} \otimes \mathbf{I}_r) \cdot (\mathbf{I}_n \otimes \mathbf{B}) \tag{22.3-12b}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix. If $\mathbf{A}$ is $n \times n$ and $\mathbf{B}$ is $t \times t$ then

$$\det(\mathbf{A} \otimes \mathbf{B}) = \det(\mathbf{A})^t \det(\mathbf{B})^n \tag{22.3-13}$$

Back to the Walsh transform, we have $\mathbf{W}_1 = [1]$ and for $n = 2^k$, $n > 1$:

$$\mathbf{W}_n = \begin{bmatrix} +\mathbf{W}_{n/2} & +\mathbf{W}_{n/2} \\ +\mathbf{W}_{n/2} & -\mathbf{W}_{n/2} \end{bmatrix} = \mathbf{W}_2 \otimes \mathbf{W}_{n/2} \tag{22.3-14}$$

In order to see that this relation is the statement of a fast algorithm split the (to be transformed) vector x into halves

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \tag{22.3-15}$$

and write out the matrix-vector product

$$\mathbf{W}_n\, x \quad = \quad \begin{bmatrix} \mathbf{W}_{n/2}\, x_0 + \mathbf{W}_{n/2}\, x_1 \\ \mathbf{W}_{n/2}\, x_0 - \mathbf{W}_{n/2}\, x_1 \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{W}_{n/2}\, (x_0 + x_1) \\ \mathbf{W}_{n/2}\, (x_0 - x_1) \end{bmatrix} \tag{22.3-16}$$

That is, a length-n transform can be computed by two length-$n/2$ transforms of the sum and difference of the first and second half of x.

We define a notation equivalent to the product sign,

$$\bigotimes_{k=1}^{n} \mathbf{M}_k \quad := \quad \mathbf{M}_1 \otimes \mathbf{M}_2 \otimes \mathbf{M}_3 \otimes \ldots \otimes \mathbf{M}_n \tag{22.3-17}$$

where the empty product equals a 1×1 matrix with entry 1. When $\mathbf{A} = \mathbf{B}$ in relation 22.3-11b we have $(\mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$, $(\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})^{-1} = \mathbf{A}^{-1} \otimes \mathbf{A}^{-1} \otimes \mathbf{A}^{-1}$ and so on. That is,

$$\left(\bigotimes_{k=1}^{n} \mathbf{A} \right)^{-1} \quad = \quad \bigotimes_{k=1}^{n} \mathbf{A}^{-1} \tag{22.3-18}$$

For the Walsh transform:

$$\mathbf{W}_n \quad = \quad \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2 \tag{22.3-19}$$

and

$$\mathbf{W}_n^{-1} \quad = \quad \bigotimes_{k=1}^{\log_2(n)} \mathbf{W}_2^{-1} \tag{22.3-20}$$

The latter relation isn't that exciting as $\mathbf{W}_2^{-1} = \mathbf{W}_2$ for the Walsh transform. However, it also holds when the inverse transform is different from the forward transform. Thereby, given a fast algorithm for some transform in form of a Kronecker product, the fast algorithm for the backward transform is immediate.

Computation of the Kronecker product of two matrices is implemented as a method in [FXT: `class matrix` in matrix/matrix.h]:

```
bool kronecker(const matrix<Type> A, const matrix<Type> B)
{
    ulong nra = A.nr_,  nca = A.nc_;
    ulong nrb = B.nr_,  ncb = B.nc_;
    if ( nra * nrb != nr_ )  return false;
    if ( nca * ncb != nc_ )  return false;
    for (ulong ra=0; ra<nra; ++ra)
    {
        ulong ro = ra * nrb;
        for (ulong ca=0; ca<nca; ++ca)
        {
            ulong co = ca * ncb;
            Type ea = A.get(ra,ca);
            for (ulong rb=0; rb<nrb; ++rb)
            {
                ulong r = ro + rb;
                for (ulong cb=0; cb<ncb; ++cb)
                {
                    Type eb = B.get(rb,cb);
                    ulong c = co + cb;
                    Type p = ea * eb;
                    set(r, c, p);
                }
            }
        }
    }
    return true;
}
```

Here `get(r,c)` returns the entry in row `r` and column `c`, and `set(r,c,x)` assigns the value `x` to the entry.

The *direct sum* of two matrices is defined as

$$\mathbf{A} \oplus \mathbf{B} \quad := \quad \begin{bmatrix} \mathbf{A} & 0 \\ 0 & \mathbf{B} \end{bmatrix} \tag{22.3-21}$$

In general $\mathbf{A} \oplus \mathbf{B} \neq \mathbf{B} \oplus \mathbf{A}$. As an analogue to the sum sign we have

$$\bigoplus_{k=1}^{n} \mathbf{A} \quad := \quad \mathbf{I}_n \otimes \mathbf{A} \tag{22.3-22}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix. The matrix $\mathbf{I}_n \otimes \mathbf{A}$ consists of n copies of $\mathbf{A}$ that lie on the diagonal. The Kronecker product can be used to derive properties of unitary transforms, see [199].

22.4 A variant of the Walsh transform *

All operations necessary for the Walsh transform are cheap: loads, stores, additions and subtractions. The memory access pattern is a major concern with direct mapped cache, as we have verified comparing the first two implementations in this chapter. Even the one found to be superior due to its more localized access is guaranteed to have a performance problem as soon as the array is long enough: all accesses are separated by a power-of-two distance and cache misses will occur beyond a certain limit. Rather bizarre attempts like inserting 'pad data' have been reported in order to mitigate the problem. The Gray code permutation described in section 2.8 on page 97 suggests an interesting solution where the sub-arrays are always accessed in mutually reversed order [FXT: walsh/walshgray.h]:

```
template <typename Type>
void walsh_gray(Type *f, ulong ldn)
// Gray variant of the Walsh transform.
// Radix-2 decimation in frequency (DIF) algorithm
{
    const ulong n = (1UL<<ldn);

    for (ulong ldm=ldn; ldm>0; --ldm) // dif
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
            for (  ;  t1<t2;  ++t1,--t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The transform is not self-inverse, however, the inverse transform can be implemented easily be reversing the steps:

```
template <typename Type>
void inverse_walsh_gray(Type *f, ulong ldn)
// Inverse of walsh_gray().
// Radix-2 decimation in time (DIT) algorithm.
{
    const ulong n = (1UL<<ldn);

    for (ulong ldm=1; ldm<=ldn; ++ldm) // dit
    {
        const ulong m = (1UL<<ldm);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r + m - 1;
```

```
            for (  ;  t1<t2;  ++t1,--t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

Using Q for the `grs_negate()` routine (described below), W_k for `walsh_wak()`, W_g for `walsh_gray()` and G for `gray_permute()` then

$$W_k \;=\; Q\,W_g\,G^{-1} \;=\; G\,W_g^{-1}\,Q \tag{22.4-1}$$

That is, the following two sequences of statements

```
inverse_gray_permute(f, n);  walsh_gray(f, ldn);  grs_negate(f, n);
```

```
grs_negate(f, n);  inverse_walsh_gray(f, ldn);  gray_permute(f, n);
```

are both equivalent to the call `walsh_wak(f, ldn)`. The function [FXT: `grs_negate()` in aux1/grsnegate.h] changes signs for certain elements:

```
template <typename Type>
void grs_negate(Type *f, ulong n)
// Negate elements at indices where the Golay-Rudin-Shapiro is negative.
{
    for (ulong k=0; k<n; ++k)
    {
        if ( grs_negative_q(k) )  f[k] = -f[k];
    }
}
```

The function `grs_negative_q()` is described in section 1.15.5 on page 40.

It turns out that the Gray-variant only wins on machines where the memory clock speed is significantly lower than the CPU. While a call to `walsh_gray()` alone is never slower than the `walsh_wak()` routine the additional steps often cause too much overhead.

22.5 Higher radix Walsh transforms

A generator for short-length Walsh (wak) transforms is given as [FXT: fft/gen-walsh-demo.cc]. It can create code for DIF and DIT transforms. For example, the code for the 4-point DIF transforms is

```
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f)
{
    Type t0, t1, t2, t3;
    t0 = f[0];
    t1 = f[1];
    t2 = f[2];
    t3 = f[3];
    sumdiff( t0, t2 );
    sumdiff( t1, t3 );
    sumdiff( t0, t1 );
    sumdiff( t2, t3 );
    f[0] = t0;
    f[1] = t1;
    f[2] = t2;
    f[3] = t3;
}
```

To keep the code readable, we use the `sumdiff()` function from [FXT: aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b}  <--| {a+b, a-b}
```

```
{ Type t=a-b; a+=b; b=t; }
```

We further need a variant that transforms elements which are not contiguous but lie apart by a distance s:

```
template <typename Type>
inline void
short_walsh_wak_dif_4(Type *f, ulong s)
{
    Type t0, t1, t2, t3;
    {
    ulong x = 0;
    t0 = f[x];  x += s;
    t1 = f[x];  x += s;
    t2 = f[x];  x += s;
    t3 = f[x];
    }
    sumdiff( t0, t2 );
    sumdiff( t1, t3 );
    sumdiff( t0, t1 );
    sumdiff( t2, t3 );
    {
    ulong x = 0;
    f[x] = t0;  x += s;
    f[x] = t1;  x += s;
    f[x] = t2;  x += s;
    f[x] = t3;
    }
}
```

The short DIF transforms are given in [FXT: walsh/shortwalshwakdif.h], DIT variants in [FXT: walsh/shortwalshwakdit.h]. A radix-4 DIF transform using these ingredients is [FXT: walsh/walshwak4.h]:

```
template <typename Type>
void walsh_wak_dif4(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-4 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);

    if ( n<=2 )
    {
        if ( n==2 )  short_walsh_wak_dif_2(f);
        return;
    }

    for (ulong ldm=ldn; ldm>3; ldm-=2)
    {
        ulong m = (1UL<<ldm);
        ulong m4 = (m>>2);
        for (ulong r=0;  r<n;  r+=m)
        {
            for (ulong j=0;  j<m4;  j++)  short_walsh_wak_dif_4(f+j+r, m4);
        }
    }

    if ( ldn & 1 )  // n is not a power of 4, need a radix-8 step
    {
        for (ulong i0=0; i0<n; i0+=8)  short_walsh_wak_dif_8(f+i0);
    }
    else
    {
        for (ulong i0=0; i0<n; i0+=4)  short_walsh_wak_dif_4(f+i0);
    }
}
```

With the implementation radix-8 DIF transform some care must be taken to choose the correct final step size [FXT: walsh/walshwak8.h]:

```
template <typename Type>
void walsh_wak_dif8(Type *f, ulong ldn)
// Transform wrt. to Walsh-Kronecker basis (wak-functions).
// Radix-8 decimation in frequency (DIF) algorithm.
// Self-inverse.
{
    const ulong n = (1UL<<ldn);
```

```
    if ( n<=4 )
    {
        switch (n )
        {
        case 4:  short_walsh_wak_dif_4(f);  break;
        case 2:  short_walsh_wak_dif_2(f);  break;
        }
        return;
    }
    const ulong xx = 4;
    ulong ldm;
    for (ldm=ldn; ldm>xx; ldm-=3)
    {
        ulong m = (1UL<<ldm);
        ulong m8 = (m>>3);
        for (ulong r=0;  r<n;  r+=m)
        {
            for (ulong j=0;  j<m8;  j++)  short_walsh_wak_dif_8(f+j+r, m8);
        }
    }
    switch ( ldm )
    {
    case 4:
        for (ulong i0=0; i0<n; i0+=16)  short_walsh_wak_dif_16(f+i0);
        break;
    case 3:
        for (ulong i0=0; i0<n; i0+=8)  short_walsh_wak_dif_8(f+i0);
        break;
    case 2:
        for (ulong i0=0; i0<n; i0+=4)  short_walsh_wak_dif_4(f+i0);
        break;
    }
}
```

Performance

For the performance comparison we include a matrix variant of the Walsh transform [FXT: walsh/walshwakmatrix.h]:

```
template <typename Type>
void walsh_wak_matrix(Type *f, ulong ldn)
{
    ulong ldc = (ldn>>1);
    ulong ldr = ldn-ldc;  // ldr>=ldc
    ulong nc = (1UL<<ldc);
    ulong nr = (1UL<<ldr);  // nrow >= ncol
    for (ulong r=0; r<nr; ++r)  walsh_wak_dif4(f+r*nc, ldc);
    transpose2(f, nr, nc);
    for (ulong c=0; c<nc; ++c)  walsh_wak_dif4(f+c*nr, ldr);
    transpose2(f, nc, nr);
}
```

The transposition routine is given in [FXT: aux2/transpose2.h]. We only use even powers of two so the transposition is that of a square matrix.

As for dyadic convolutions we do not need the data in a particular order so we also include a version of the matrix algorithm that omits the final transposition:

```
template <typename Type>
void walsh_wak_matrix_1(Type *f, ulong ldn, int is)
{
    ulong ldc = (ldn>>1);
    ulong ldr = ldn-ldc;  // ldr>=ldc
    if ( is<0 )  swap2(ldr, ldc);  // inverse
    ulong nc = (1UL<<ldc);
    ulong nr = (1UL<<ldr);  // nrow >= ncol
    for (ulong r=0; r<nr; ++r)  walsh_wak_dif4(f+r*nc, ldc);
    transpose2(f, nr, nc);
    for (ulong c=0; c<nc; ++c)  walsh_wak_dif4(f+c*nr, ldr);
}
```

The following calls give (up to normalization) the mutually inverse transforms:

```
walsh_wak_matrix_1(f, ldn, +1);
walsh_wak_matrix_1(f, ldn, -1);
```

We do not consider the range of transform lengths $n < 128$ where unrolled routines and the radix-4 algorithm consistently win. Figure 22.5-A shows a comparison of the routines given so far. There are clearly two regions to distinguish: firstly, the region were the transforms fit into the first-level data cache (which is 64 kilobyte, corresponding to `ldn` = 13). Secondly, the region where `ldn` > 13 and the performance becomes more and more memory bound.

In the first region the radix-4 routine is the fastest. The radix-8 routine comes close but, somewhat surprisingly, never wins.

In the second region the matrix version is the best. However, for very large sizes its performance could be better. Note that with odd `ldn` (not shown) its performance drops significantly due to the more expensive transposition operation. The transposition is clearly the bottleneck. One can use (machine specific) optimizations for the transposition to further improve the performance.

In the next section we give an algorithm that avoids the transposition completely and consistently outperforms the matrix algorithm.

22.6 Localized Walsh transforms

A decimation in time (DIT) algorithm combines the two halves of the array, then the halves of the two halves, the halves of each quarter, and so on. With each step the whole array is accessed which leads to the performance drop as soon as the array does not fit into the cache.

One can reorganize the algorithm as follows: combine the two halves of the array and postpone further processing of the upper half, then combine the halves of the lowers half and again postpone processing of its upper half. Repeat until size two is reached. Then use the algorithm at the postponed parts, starting with the smallest (last postponed).

The scheme can be sketched for size 16, as follows:

```
hhhhhhhhhhhhhhhh
hhhhhhhh44444444
hhhh333344444444
hh22333344444444
```

The letters '`h`' denote places processed before any recursive call. The blocks of twos, threes and fours denote postponed blocks. The Walsh transform is thereby decomposed into a sequence of Haar transforms (see figure 23.6-A on page 476). The algorithm described is most easily implemented via recursion:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
{
    if ( ldn<1 )  return;

    // Recursion:
    for (ulong ldm=1; ldm<ldn; ++ldm)  walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

Optimizations are obtained by avoiding recursions for small sizes. We use a radix-4 algorithm as soon as the transform size is smaller or equal to the cache size and we avoid recursion for tiny transforms:

```
template <typename Type>
void walsh_wak_loc_dit2(Type *f, ulong ldn)
```

```
 8 == ldn;   MemSize ==     2 kB  ==  256 doubles;   rep == 976563
              walsh_wak_dif2(f,ldn);    dt=    2.49551    MB/s= 6114   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.56806    MB/s= 9731   rel=   0.628352 *
              walsh_wak_dif8(f,ldn);    dt=    1.57419    MB/s= 9693   rel=    0.63081
            walsh_wak_matrix(f,ldn);    dt=    2.28047    MB/s= 6691   rel=    0.91383
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.94357    MB/s= 7851   rel=   0.778827
                 walsh_gray(f,ldn);     dt=    2.39791    MB/s= 6363   rel=   0.960888

10 == ldn;   MemSize ==     8 kB  ==  1024 doubles;   rep == 195313
              walsh_wak_dif2(f,ldn);    dt=    2.26683    MB/s= 6731   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.47338    MB/s=10356   rel=   0.649977 *
              walsh_wak_dif8(f,ldn);    dt=    1.65262    MB/s= 9233   rel=   0.729044
            walsh_wak_matrix(f,ldn);    dt=    1.91859    MB/s= 7953   rel=   0.846378
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.69215    MB/s= 9017   rel=   0.746485
                 walsh_gray(f,ldn);     dt=    2.18454    MB/s= 6985   rel=     0.9637

12 == ldn;   MemSize ==    32 kB  ==  4096 doubles;   rep == 20345
              walsh_wak_dif2(f,ldn);    dt=     1.0884    MB/s= 7010   rel=          1
              walsh_wak_dif4(f,ldn);    dt=   0.723136    MB/s=10550   rel=   0.664403 *
              walsh_wak_dif8(f,ldn);    dt=   0.790313    MB/s= 9654   rel=   0.726124
            walsh_wak_matrix(f,ldn);    dt=    1.01233    MB/s= 7536   rel=   0.930112
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.926387    MB/s= 8236   rel=   0.851146
                 walsh_gray(f,ldn);     dt=    1.05364    MB/s= 7241   rel=   0.968062

14 == ldn;   MemSize ==   128 kB  ==  16384 doubles;   rep == 2180
              walsh_wak_dif2(f,ldn);    dt=    1.17042    MB/s= 3260   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.14861    MB/s= 3321   rel=   0.981368
              walsh_wak_dif8(f,ldn);    dt=    1.08501    MB/s= 3516   rel=   0.927026
            walsh_wak_matrix(f,ldn);    dt=   0.669182    MB/s= 5701   rel=   0.571747
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.552063    MB/s= 6910   rel=   0.471681 *
                 walsh_gray(f,ldn);     dt=    1.00794    MB/s= 3785   rel=    0.86118
16 == ldn;   MemSize ==   512 kB  ==  65536 doubles;   rep == 477
              walsh_wak_dif2(f,ldn);    dt=    1.40004    MB/s= 2726   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    1.70347    MB/s= 2240   rel=    1.21673
              walsh_wak_dif8(f,ldn);    dt=    1.12997    MB/s= 3377   rel=   0.807095
            walsh_wak_matrix(f,ldn);    dt=   0.801902    MB/s= 4759   rel=   0.572769
      walsh_wak_matrix_1(f,ldn,+1);     dt=   0.628073    MB/s= 6076   rel=   0.448609 *
                 walsh_gray(f,ldn);     dt=    1.18693    MB/s= 3215   rel=   0.847779
18 == ldn;   MemSize ==     2 MB  ==  256 K doubles;   rep == 106
              walsh_wak_dif2(f,ldn);    dt=    2.61599    MB/s= 1459   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.55153    MB/s= 1496   rel=   0.975359
              walsh_wak_dif8(f,ldn);    dt=     1.9791    MB/s= 1928   rel=   0.756538
            walsh_wak_matrix(f,ldn);    dt=    1.77306    MB/s= 2152   rel=   0.677776
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.14735    MB/s= 3326   rel=   0.438591 *
                 walsh_gray(f,ldn);     dt=    2.51539    MB/s= 1517   rel=   0.961545
20 == ldn;   MemSize ==     8 MB  ==  1024 K doubles;   rep == 24
              walsh_wak_dif2(f,ldn);    dt=    2.64158    MB/s= 1454   rel=          1
              walsh_wak_dif4(f,ldn);    dt=     2.8532    MB/s= 1346   rel=    1.08011
              walsh_wak_dif8(f,ldn);    dt=    2.34867    MB/s= 1635   rel=   0.889113
            walsh_wak_matrix(f,ldn);    dt=    1.88431    MB/s= 2038   rel=   0.713327
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.21084    MB/s= 3171   rel=   0.458376 *
                 walsh_gray(f,ldn);     dt=    2.58747    MB/s= 1484   rel=   0.979514
22 == ldn;   MemSize ==    32 MB  ==  4096 K doubles;   rep == 5
              walsh_wak_dif2(f,ldn);    dt=    2.43537    MB/s= 1445   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.82337    MB/s= 1247   rel=    1.15932
              walsh_wak_dif8(f,ldn);    dt=    2.07422    MB/s= 1697   rel=   0.851708
            walsh_wak_matrix(f,ldn);    dt=    1.99251    MB/s= 1767   rel=   0.818155
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.22719    MB/s= 2868   rel=   0.503901 *
                 walsh_gray(f,ldn);     dt=    2.38611    MB/s= 1475   rel=   0.979776
24 == ldn;   MemSize ==   128 MB  ==  16384 K doubles;   rep == 1
              walsh_wak_dif2(f,ldn);    dt=    2.10939    MB/s= 1456   rel=          1
              walsh_wak_dif4(f,ldn);    dt=    2.61517    MB/s= 1175   rel=    1.23977
              walsh_wak_dif8(f,ldn);    dt=    2.11508    MB/s= 1452   rel=     1.0027
            walsh_wak_matrix(f,ldn);    dt=    2.16597    MB/s= 1418   rel=    1.02683
      walsh_wak_matrix_1(f,ldn,+1);     dt=    1.28349    MB/s= 2393   rel=   0.608466 *
                 walsh_gray(f,ldn);     dt=    2.04744    MB/s= 1500   rel=   0.970635
```

Figure 22.5-A: Relative speed of different implementations of the Walsh (wak) transform. The transforms were run '`rep`' times for each measurement, the quantity '`dt`' gives the elapsed time for `rep` transforms of the given type. The quantity '`MB/s`' gives the memory transfer rate as if a radix-2 algorithm was used, it equals '`Memsize`' times '`ldn`' divided by the time elapsed for a single transform. The '`rel`' gives the performance relative to the radix-2 version, smaller values mean better performance.

```
{
    if ( ldn<=13 )  // parameter: (2**13)*sizeof(Type) <= L1-cache
    {
        walsh_wak_dif4(f,ldn);  // note: DIF version, result is the same
        return;
    }

    // Recursion:
    short_walsh_wak_dit_2(f+2);  // ldm==1
    short_walsh_wak_dit_4(f+4);  // ldm==2
    short_walsh_wak_dit_8(f+8);  // ldm==3
    short_walsh_wak_dit_16(f+16);  // ldm==4
    for (ulong ldm=5; ldm<ldn; ++ldm)  walsh_wak_loc_dit2(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

The routine is given in [FXT: walsh/walshwakloc2.h]. A decimation in frequency (DIF) version is obtained by reversion of the steps:

```
template <typename Type>
void walsh_wak_loc_dif2(Type *f, ulong ldn)
{
    if ( ldn<=13 )  // parameter: (2**13)*sizeof(Type) <= L1-cache
    {
        walsh_wak_dif4(f,ldn);
        return;
    }

    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)
        {
            Type u = f[t1];
            Type v = f[t2];
            f[t1] = u + v;
            f[t2] = u - v;
        }
    }

    // Recursion:
    short_walsh_wak_dif_2(f+2);  // ldm==1
    short_walsh_wak_dif_4(f+4);  // ldm==2
    short_walsh_wak_dif_8(f+8);  // ldm==3
    short_walsh_wak_dif_16(f+16);  // ldm==4
    for (ulong ldm=5; ldm<ldn; ++ldm)  walsh_wak_loc_dif2(f+(1UL<<ldm), ldm);
}
```

The double loop in the algorithm is a reversed Haar transform, see chapter 23 on page 465. The double loop in the DIF algorithm is a transposed reversed Haar transform. The (generated) short-length transforms are given in the files [FXT: walsh/shortwalshwakdif.h] and [FXT: walsh/shortwalshwakdit.h]. For example, the length-8, decimation in frequency routine is

```
template <typename Type>
inline void
short_walsh_wak_dif_8(Type *f)
{
 Type t0, t1, t2, t3, t4, t5, t6, t7;
 t0 = f[0]; t1 = f[1]; t2 = f[2]; t3 = f[3];
 t4 = f[4]; t5 = f[5]; t6 = f[6]; t7 = f[7];
 sumdiff( t0, t4 ); sumdiff( t1, t5 ); sumdiff( t2, t6 ); sumdiff( t3, t7 );
 sumdiff( t0, t2 ); sumdiff( t1, t3 ); sumdiff( t4, t6 ); sumdiff( t5, t7 );
 sumdiff( t0, t1 ); sumdiff( t2, t3 ); sumdiff( t4, t5 ); sumdiff( t6, t7 );
 f[0] = t0; f[1] = t1; f[2] = t2; f[3] = t3;
 f[4] = t4; f[5] = t5; f[6] = t6; f[7] = t7;
}
```

The used strategy leads to a very favorable memory access pattern that results in excellent performance for large transforms. Figure 22.6-A shows a comparison between the localized transforms and the matrix

```
    14 == ldn;   MemSize ==   128 kB  ==  16384 doubles;   rep == 2180
                walsh_wak_matrix(f,ldn);    dt=   0.672327    MB/s= 5674   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=   0.555851    MB/s= 6863   rel=   0.826756
              walsh_wak_loc_dit2(f,ldn);    dt=   0.498558    MB/s= 7652   rel=   0.741541 *
              walsh_wak_loc_dif2(f,ldn);    dt=   0.533746    MB/s= 7148   rel=   0.793878

    16 == ldn;   MemSize ==   512 kB  ==  65536 doubles;   rep == 477
                walsh_wak_matrix(f,ldn);    dt=   0.919579    MB/s= 4150   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=   0.692488    MB/s= 5511   rel=   0.753049
              walsh_wak_loc_dit2(f,ldn);    dt=   0.653256    MB/s= 5842   rel=   0.710386 *
              walsh_wak_loc_dif2(f,ldn);    dt=   0.670104    MB/s= 5695   rel=   0.728707

    18 == ldn;   MemSize ==     2 MB  ==  256 K doubles;   rep == 106
                walsh_wak_matrix(f,ldn);    dt=     2.2111    MB/s= 1726   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.36827    MB/s= 2789   rel=   0.618819
              walsh_wak_loc_dit2(f,ldn);    dt=   0.938006    MB/s= 4068   rel=   0.424225
              walsh_wak_loc_dif2(f,ldn);    dt=   0.927804    MB/s= 4113   rel=   0.419611 *

    20 == ldn;   MemSize ==     8 MB  ==  1024 K doubles;   rep == 24
                walsh_wak_matrix(f,ldn);    dt=    2.31178    MB/s= 1661   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.42614    MB/s= 2693   rel=   0.616901
              walsh_wak_loc_dit2(f,ldn);    dt=    1.11847    MB/s= 3433   rel=   0.483811
              walsh_wak_loc_dif2(f,ldn);    dt=    1.11142    MB/s= 3455   rel=   0.480765 *

    22 == ldn;   MemSize ==    32 MB  ==  4096 K doubles;   rep == 5
                walsh_wak_matrix(f,ldn);    dt=    2.00573    MB/s= 1755   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.23695    MB/s= 2846   rel=   0.616707
              walsh_wak_loc_dit2(f,ldn);    dt=    1.16461    MB/s= 3022   rel=   0.580644
              walsh_wak_loc_dif2(f,ldn);    dt=    1.16164    MB/s= 3030   rel=   0.579162 *

    24 == ldn;   MemSize ==   128 MB  ==  16384 K doubles;   rep == 1
                walsh_wak_matrix(f,ldn);    dt=    2.16536    MB/s= 1419   rel=          1
           walsh_wak_matrix_1(f,ldn,+1);    dt=    1.28455    MB/s= 2392   rel=   0.593226
              walsh_wak_loc_dit2(f,ldn);    dt=    1.10769    MB/s= 2773   rel=   0.511552
              walsh_wak_loc_dif2(f,ldn);    dt=    1.10601    MB/s= 2778   rel=   0.510775 *
```

Figure 22.6-A: Speed comparison between localized and matrix algorithms for the Walsh transform.

algorithm. Small sizes are omitted because the localized algorithm has the very same speed as the radix-4 algorithm it falls back to. The localized algorithms are the clear winners, even against the matrix algorithm with only one transposition. For very large transforms the decimation in frequency version is very slightly faster. This is due to the fact that it starts with smaller chunks of data so more of the data is in cache when the larger sub-arrays are accessed.

The localized algorithm can easily be implemented for transforms where a radix-2 step is known. Section 24.9 on page 497 gives the fast Hartley transform variant of the localized algorithm.

One can develop similar routines with higher radix. However, a radix-4 version was found to be slower than the given routines. A certain speedup can be achieved by unrolling and prefetching. We use the C-type `double` whose size is 8 bytes. Substitute the double loop in the DIF version (that is, the Haar transform) by

```
// machine specific prefetch instruction:
#define PREF(p,o)  asm volatile ("prefetchw " #o "(%0) " : : "r" (p) )
    ulong ldm;
    for (ldm=ldn; ldm>=6; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        PREF(f, 0);  PREF(f+mh, 0);
        PREF(f, 64);  PREF(f+mh, 64);
        PREF(f, 128);  PREF(f+mh, 128);
        PREF(f, 192);  PREF(f+mh, 192);

        for (ulong t1=0, t2=mh;  t1<mh;  t1+=8, t2+=8)
        {
            double *p1 = f + t1,  *p2 = f + t2;
            PREF(p1, 256);  PREF(p2, 256);

            double u0 = f[t1+0], v0 = f[t2+0];
            double u1 = f[t1+1], v1 = f[t2+1];
            double u2 = f[t1+2], v2 = f[t2+2];
            double u3 = f[t1+3], v3 = f[t2+3];
```

```
            sumdiff(u0, v0); f[t1+0] = u0; f[t2+0] = v0;
            sumdiff(u1, v1); f[t1+1] = u1; f[t2+1] = v1;
            sumdiff(u2, v2); f[t1+2] = u2; f[t2+2] = v2;
            sumdiff(u3, v3); f[t1+3] = u3; f[t2+3] = v3;

            double u4 = f[t1+4], v4 = f[t2+4];
            double u5 = f[t1+5], v5 = f[t2+5];
            double u6 = f[t1+6], v6 = f[t2+6];
            double u7 = f[t1+7], v7 = f[t2+7];
            sumdiff(u4, v4); f[t1+4] = u4; f[t2+4] = v4;
            sumdiff(u5, v5); f[t1+5] = u5; f[t2+5] = v5;
            sumdiff(u6, v6); f[t1+6] = u6; f[t2+6] = v6;
            sumdiff(u7, v7); f[t1+7] = u7; f[t2+7] = v7;
        }
    }

    for (  ; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
```

The following list gives the speed ratio between the optimized and the unoptimized DIF routine:

```
14 == ldn;   MemSize ==   128 kB;     ratio = 1.24252
16 == ldn;   MemSize ==   512 kB;     ratio = 1.43568
18 == ldn;   MemSize ==     2 MB;     ratio = 1.23875
20 == ldn;   MemSize ==     8 MB;     ratio = 1.21012
22 == ldn;   MemSize ==    32 MB;     ratio = 1.19939
24 == ldn;   MemSize ==   128 MB;     ratio = 1.18245
```

For sizes that are out of (level-2) cache most of the speedup is due to the memory prefetch.

Iterative versions of the algorithms

DIF start	DIF length	DIT start	DIT length
.....	1.....	...1.	....1.
....1.	....1.	...11.	....1.
...1..	...1..	...1..	...1..
...11.	....1.	..1.1.	....1.
..1...	..1...	..111.	....1.
..1.1.	....1.	..11..	...1..
..11..	...1..	..1...	..1...
..111.	....1.	.1..1.	....1.
.1....	.1....	.1.11.	....1.
.1..1.	....1.	.1.1..	...1..
.1.1..	...1..	.11.1.	....1.
.1.11.	....1.	.1111.	....1.
.11...	..1...	.111..	...1..
.11.1.	....1.	.11...	..1...
.111..	...1..	.1....	.1....
.1111.	....1.	......	1.....

Figure 22.6-B: Binary values of the start index and length of the Haar transforms in the iterative version of the localized DIF (left) and DIT (right) transform. Dots are used for zeros.

In the DIF algorithm the Haar transforms are executed at positions $f+2$, $f+4$, $f+6$, ... and the length of the transform at position $f+s$ is determined by the lowest set bit in s. Additionally, a full-length Haar transform has to be done at the beginning. As C++ code:

```
template <typename Type>
inline void haar_dif2(Type *f, ulong n)
{
    for (ulong m=n; m>=2; m>>=1)
    {
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}

template <typename Type>
void loc_dif2(Type *f, ulong n)
```

```
{
    haar_dif2(f, n);
    for (ulong z=2; z<n; z+=2)  haar_dif2(f+z,  (z&-z));
}
```

Note that the routines now take the length of the transform as second argument, not its base-2 logarithm.

With the DIT algorithm matters are slightly more complicated. A pattern can be observed by printing the binary expansions of the starting position and length of the transforms shown in figure 22.6-B (created with [FXT: fft/locrec-demo.cc]). The lengths are again determined by the lowest bit of the start position. And we have also seen the pattern in the left column: the reversed binary words in reversed (subset-) lexicographic order, see figure 1.27-A on page 65. The implementation is quite concise:

```
template <typename Type>
inline void haar_dit2(Type *f, ulong n)
{
    for (ulong m=1; m<=n; m<<=1)
    {
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
template <typename Type>
void loc_dit2(Type f, ulong n)
{
    for (ulong z=2, u=1; z<n; z+=2)
    {
        ulong s = u<<1;
        haar_dit2(f+s,  (s&-s));
        u = prev_lexrev(u);
    }
    haar_dit2(f, n);
}
```

The routines are slightly slower than the recursive version because they do not fall back to the full Walsh transforms when the transform size is small.

The DIT scheme is a somewhat surprising application of the seemingly esoteric routine [FXT: `prev_lexrev()` in bits/bitlex.h]. Plus we have found a recursive algorithm for the generation of the binary words in (subset-) lexicographic order [FXT: bits/bitlex-rec-demo.cc]:

```
void bitlex_b(ulong f, ulong n)
{
    for (ulong m=1; m<n; m<<=1)  bitlex_b(f+m, m);
    print_bin("  ", f, ldn);
}
```

22.7 Dyadic (XOR) convolution

Dyadic convolution has XOR where the usual one has plus

The *dyadic convolution* of the sequences a and b is the sequence h defined by

$$h_\tau := \sum_{i \oplus j = \tau} a_i \, b_j \tag{22.7-1a}$$

$$= \sum_{i} a_i \, b_{i \oplus \tau} \tag{22.7-1b}$$

where the symbol '$\oplus$' stands for bit-wise XOR operator. All three sequence must be of the same length that is a power of 2. The dyadic convolution could rightfully be called *XOR-convolution*.

The semi-symbolic scheme of the convolution is shown in figure 22.7-A. The table is equivalent to the one (for cyclic convolution) given in figure 21.1-A on page 410. The dyadic convolution can be used for the multiplication of hypercomplex numbers as shown in section 37.16 on page 787.

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

 4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
 5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
 6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
 7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   8  9 10 11  12 13 14 15   0  1  2  3   4  5  6  7
 9:   9  8 11 10  13 12 15 14   1  0  3  2   5  4  7  6
10:  10 11  8  9  14 15 12 13   2  3  0  1   6  7  4  5
11:  11 10  9  8  15 14 13 12   3  2  1  0   7  6  5  4

12:  12 13 14 15   8  9 10 11   4  5  6  7   0  1  2  3
13:  13 12 15 14   9  8 11 10   5  4  7  6   1  0  3  2
14:  14 15 12 13  10 11  8  9   6  7  4  5   2  3  0  1
15:  15 14 13 12  11 10  9  8   7  6  5  4   3  2  1  0
```

Figure 22.7-A: Semi-symbolic scheme for the dyadic convolution of two length-16 sequences.

A fast algorithm for the computation of the dyadic convolution uses the Walsh transform [FXT: `dyadic_convolution()` in walsh/dyadiccnvl.h]:

```
template <typename Type>
void dyadic_convolution(Type * restrict f, Type * restrict g, ulong ldn)
// Dyadic convolution (XOR-convolution):  h[] of f[] and g[]:
//   h[k] = sum( i XOR j == k,  f[i]*g[k] )
// Result is written to g[].
// ldn := base-2 logarithm of the array length
{
    walsh_wak(f, ldn);
    walsh_wak(g, ldn);
    const ulong n = (1UL<<ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    walsh_wak(g, ldn);
}
```

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  0  3  2   5  4  7  6   9  8 11 10  13 12 15 14
 2:   2  3  0  1   6  7  4  5  10 11  8  9  14 15 12 13
 3:   3  2  1  0   7  6  5  4  11 10  9  8  15 14 13 12

 4:   4  5  6  7   0  1  2  3  12 13 14 15   8  9 10 11
 5:   5  4  7  6   1  0  3  2  13 12 15 14   9  8 11 10
 6:   6  7  4  5   2  3  0  1  14 15 12 13  10 11  8  9
 7:   7  6  5  4   3  2  1  0  15 14 13 12  11 10  9  8

 8:   8  9 10 11  12 13 14 15   0- 1- 2- 3-  4- 5- 6- 7-
 9:   9  8 11 10  13 12 15 14   1- 0- 3- 2-  5- 4- 7- 6-
10:  10 11  8  9  14 15 12 13   2- 3- 0- 1-  6- 7- 4- 5-
11:  11 10  9  8  15 14 13 12   3- 2- 1- 0-  7- 6- 5- 4-

12:  12 13 14 15   8  9 10 11   4- 5- 6- 7-  0- 1- 2- 3-
13:  13 12 15 14   9  8 11 10   5- 4- 7- 6-  1- 0- 3- 2-
14:  14 15 12 13  10 11  8  9   6- 7- 4- 5-  2- 3- 0- 1-
15:  15 14 13 12  11 10  9  8   7- 6- 5- 4-  3- 2- 1- 0-
```

Figure 22.7-B: Semi-symbolic scheme for the dyadic equivalent of the negacyclic convolution. Negative contributions to a bucket have a minus appended.

An scheme similar to that of the negacyclic convolution is shown in figure 22.7-B. It can be obtained via

```
    walsh_wal_dif2_core(f, ldn);
    walsh_wal_dif2_core(g, ldn);
    ulong n = (1UL<<ldn);
    for (ulong i=0,j=n-1; i<j; --j,++i)  fht_mul(f[i], f[j], g[i], g[j], 0.5);
    walsh_wal_dit2_core(g, ldn);
```

where `fht_mul()` is the operation used for the convolution with fast Hartley transforms [FXT: convolu-

tion/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi,  h1m = xj;
    Type s1 = h1p + h1m,  d1 = h1p - h1m;
    Type h2p = yi,  h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

22.8 The Walsh transform: Walsh-Paley basis

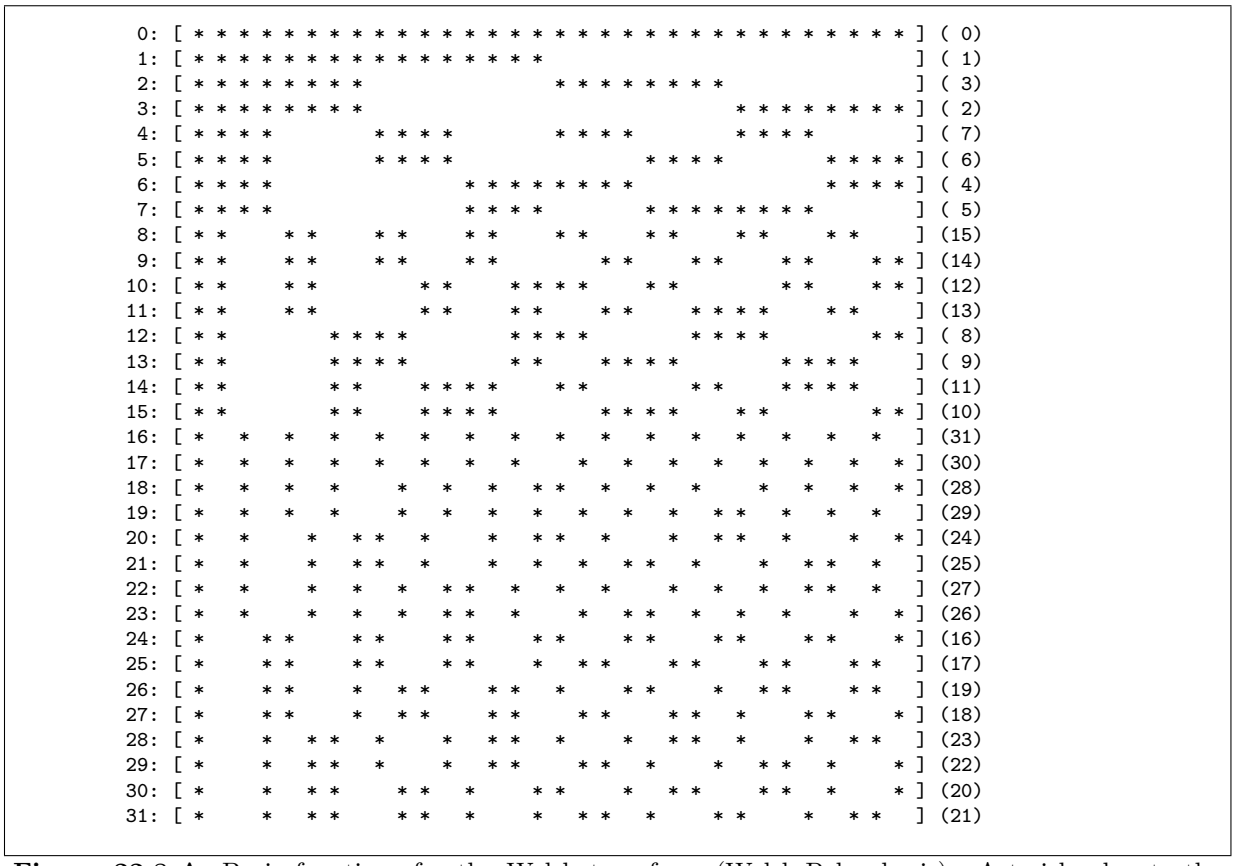


Figure 22.8-A: Basis functions for the Walsh transform (Walsh-Paley basis). Asterisks denote the value $+1$, blank entries denote -1.

A Walsh transform with a different (ordering of the) basis can be obtained by [FXT: walsh/walshpal.h]:

```
template <typename Type>
void walsh_pal(Type *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    walsh_wak(f, ldn);
// =^=
//    walsh_wak(f, ldn);
//    revbin_permute(f, n);
}
```

The basis functions are shown in figure 22.8-A. Actually one can also apply the revbin permutation before the transform. That is,

$$W_p \;=\; W_k\,R \;=\; R\,W_k \tag{22.8-1}$$

One has for W_p

$$W_p \;=\; G\,W_p\,G \;=\; G^{-1}\,W_p\,G^{-1} \tag{22.8-2}$$
$$\;=\; Z\,W_p\,Z \;=\; Z^{-1}\,W_p\,Z^{-1} \tag{22.8-3}$$

where Z denotes the zip permutation (see section 2.5 on page 93) and G denotes the Gray permutation (see section 2.8 on page 97).

A function that computes the k-th base function of the transform is [FXT: `walsh_pal_basefunc()` in walsh/walshbasefunc.h]:

```
template <typename Type>
void walsh_pal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

22.9 Sequency ordered Walsh transforms

The term corresponding to the frequency of the Fourier basis functions is the so-called *sequency* of the Walsh functions, the number of the changes of sign of the individual functions. Note that the sequency of a signal with frequency f usually is $2\,f$.

If the basis functions shall be ordered by their sequency one can use

```
    const ulong n = (1UL<<ldn);
    walsh_wak(f, ldn);
    revbin_permute(f, n);
    inverse_gray_permute(f, n);
```

That is

$$W_w \;=\; G^{-1}\,R\,W_k \;=\; W_k\,R\,G \tag{22.9-1}$$

A function that computes the k-th base function of the transform is [FXT: `walsh_wal_basefunc()` in walsh/walshbasefunc.h]:

```
template <typename Type>
void walsh_wal_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n)+1);
    k = gray_code(k);
//    // =^=
//    k = revbin(k, ld(n));
//    k = rev_gray_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

A version of the transform that avoids the Gray permutation is based on [FXT: walsh/walshwal.h]

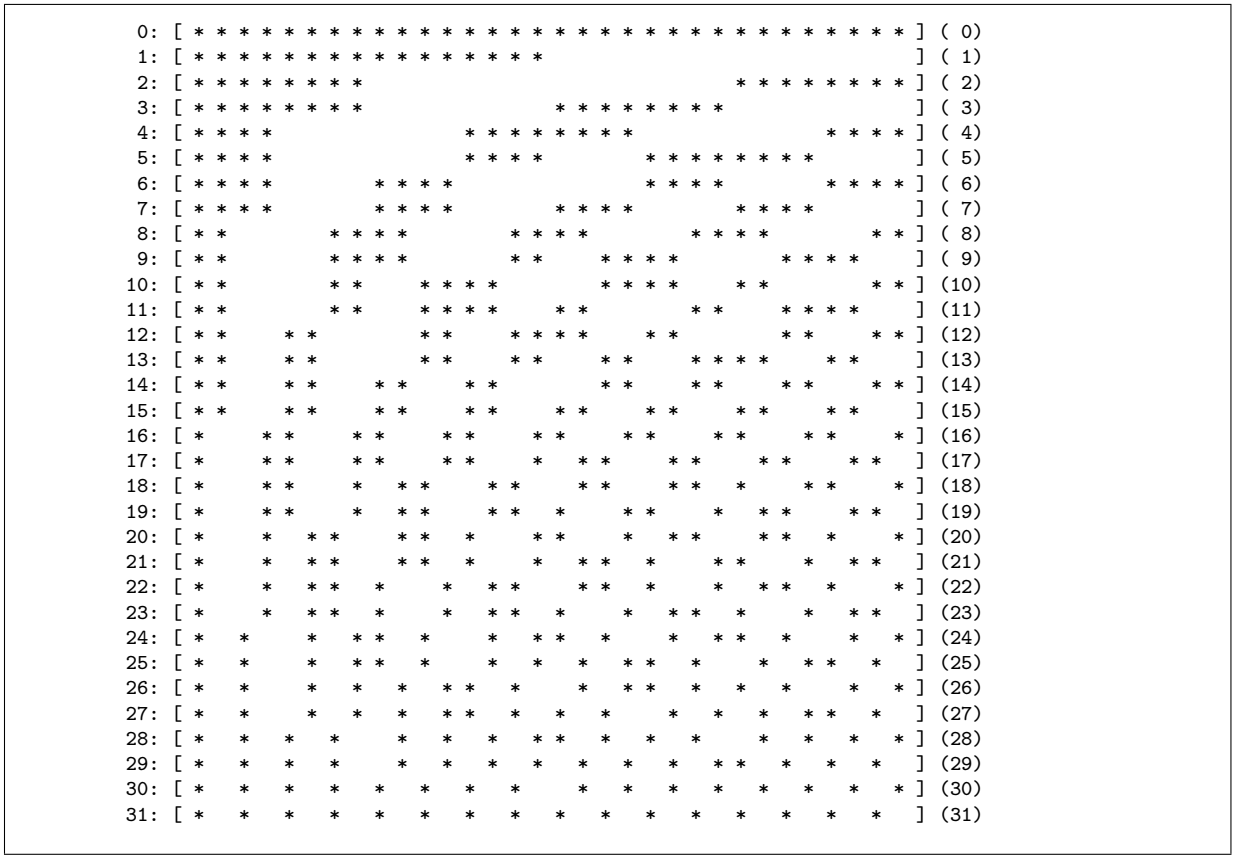


Figure 22.9-A: Basis functions for the sequency-ordered Walsh transform (Walsh-Kacmarz basis). Asterisks denote the value +1, blank entries denote −1.

```
template <typename Type>
void walsh_wal_dif2_core(Type *f, ulong ldn)
// Core routine for sequency ordered Walsh transform.
// Radix-2 decimation in frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=2; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong j;
            for (j=0; j<m4; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }

            for (  ; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = v - u;  // reversed
```

```
            }
        }
    }

    if ( ldn )
    {
        // ulong ldm=1;
        const ulong m = 2; //(1UL<<ldm);
        const ulong mh = 1; //(m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong j = 0;
//            for (ulong j=0; j<mh; ++j)
            {
                ulong t1 = r+j;
                ulong t2 = t1+mh;
                double u = f[t1];
                double v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The transform still needs the revbin permutation:

```
template <typename Type>
inline void walsh_wal(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dif2_core(f, ldn);
//  =^=
//    walsh_wal_dit2_core(f, ldn);
//    revbin_permute(f, (1UL<<ldn));
}
```

A decimation in time (DIT) version of the core-routine is also given in [FXT: walsh/walshwal.h]. The procedure `gray_permute()` is given in section 2.8 on page 97.

The sequence of statements

```
walsh_gray(f, ldn);  grs_negate(f, n);  revbin_permute(f, n);
```

is equivalent to `walsh_wal(f, ldn)` and might be faster for large arrays. We have

$$W_w \;=\; R\,Q\,W_g \;=\; W_g^{-1}\,R\,Q \tag{22.9-2}$$

22.9.1 Even/odd ordering of sequencies

An alternative ordering of the base functions (first even sequencies ascending then odd sequencies descending [FXT: `walsh_wal_rev()` in walsh/walshwalrev.h]) can be obtained by either of (with `n=1UL<<ldn`)

```
    revbin_permute(f, n);
    gray_permute(f, n);
    walsh_wak(f,ldn);
```

```
    walsh_wak(f,ldn);
    inverse_gray_permute(f, n);
    revbin_permute(f, n);
```

```
    zip_rev(f, n);
    walsh_wal(f, ldn);
```

```
    walsh_wal(f, ldn);
    unzip_rev(f, n);
```

```
    walsh_wak(f, ldn);
    inverse_gray_permute(f, n);
    revbin_permute(f, n);
```

```
    revbin_permute(f, n);
    walsh_gray(f, ldn);
    grs_negate(f, n);
```

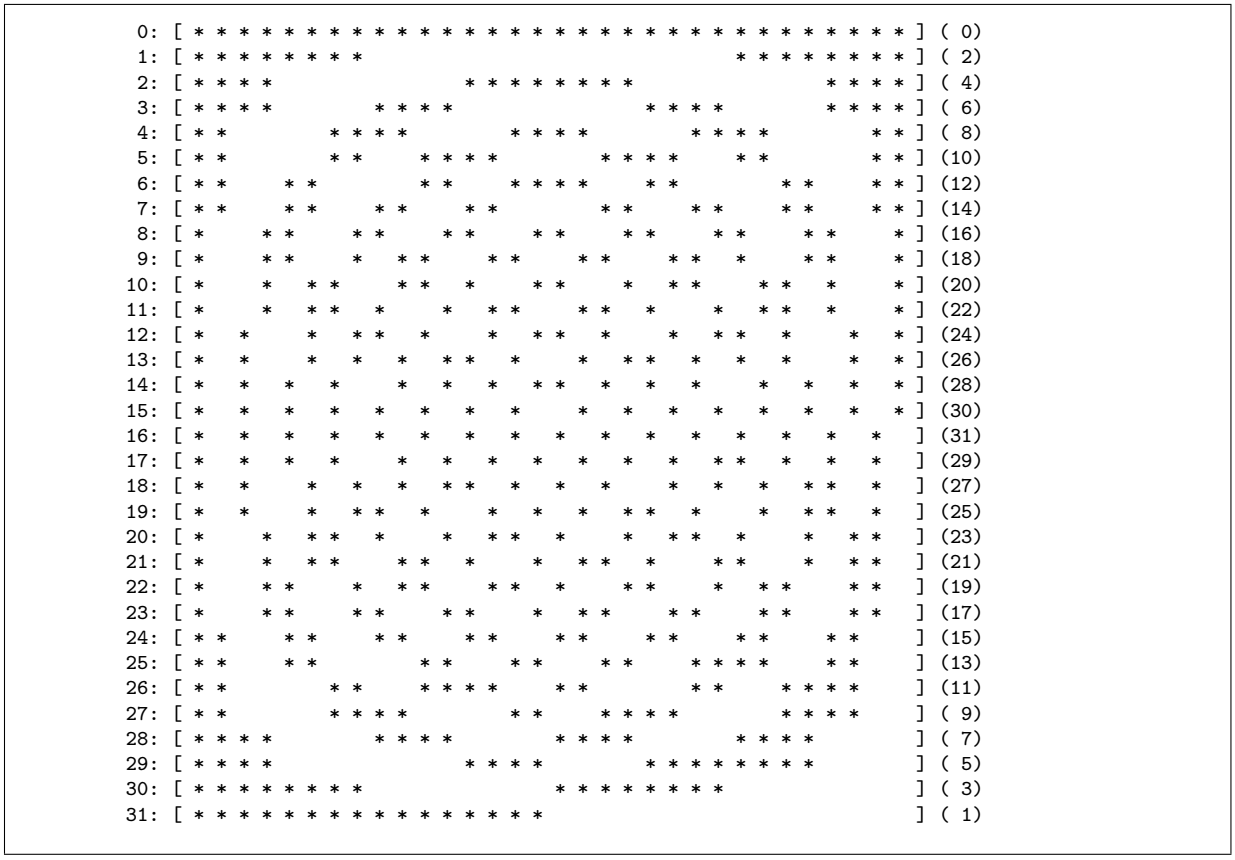


Figure 22.9-B: Basis functions for the reversed sequency ordered Walsh transform. Asterisks denote the value $+1$, blank entries denote -1.

That is,

$$\overline{W}_w = W_k\, G\, R = R\, G^{-1}\, W_k \tag{22.9-3}$$
$$= W_w\, \overline{Z} = \overline{Z}^{-1}\, W_w \tag{22.9-4}$$
$$= Q\, W_g\, R \tag{22.9-5}$$

However, an implementation that is more efficient uses the core-routines that have the Gray permutation 'absorbed' [FXT: `walsh_wal_rev()` in walsh/walshwalrev.h]:

```
template <typename Type>
inline void walsh_wal_rev(Type *f, ulong ldn)
{
    revbin_permute(f, (1UL<<ldn));
    walsh_wal_dit2_core(f, ldn);
    // =^=
//    walsh_wal_dif2_core(f, ldn);
//    revbin_permute(f, n);
}
```

This implementation uses the fact that

$$\overline{W}_w = R\, W_w\, R \tag{22.9-6}$$

Sometimes one can do still better, using the following sequence of statements that compute the same transform:

```
revbin_permute(f, n);  walsh_gray(f, ldn);  grs_negate(f, n);
```

Similar relations as for the transform with Walsh-Paley basis (22.8-2 and 22.8-3 on page 448) hold for W_w:

$$W_w = G\,W_w\,G = G^{-1}\,W_w\,G^{-1} \tag{22.9-7}$$

$$= \overline{Z}\,W_w\,\overline{Z} = \overline{Z}^{-1}\,W_w\,\overline{Z}^{-1} \tag{22.9-8}$$

The k-th base function of the transform can be computed as [FXT: `walsh_wal_rev_basefunc()` in walsh/walshbasefunc.h]

```
template <typename Type>
void walsh_wal_rev_basefunc(Type *f, ulong n, ulong k)
{
    k = revbin(k, ld(n));
    k = gray_code(k);
    // =^=
//    k = rev_gray_code(k);
//    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

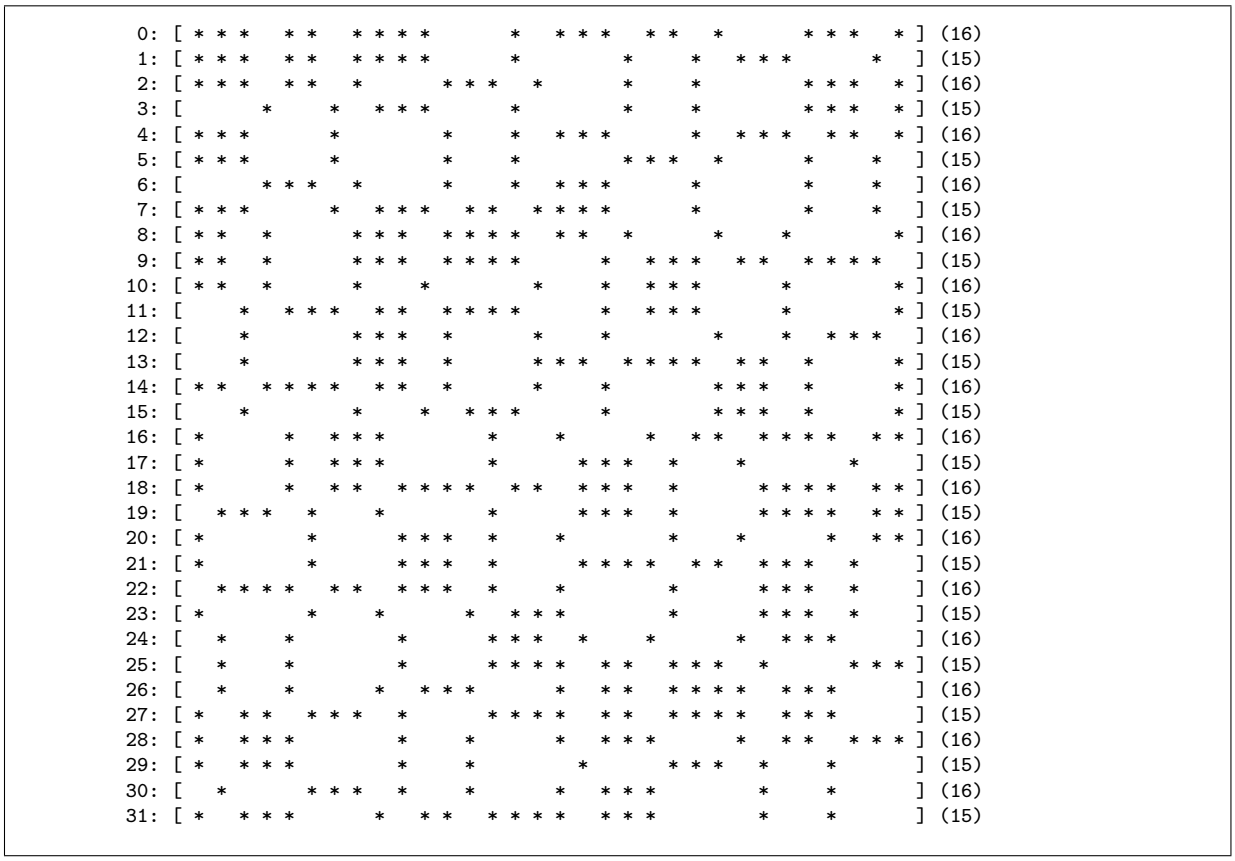


Figure 22.9-C: Basis functions for a self-inverse Walsh transform that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

22.9.2 Transforms with sequencies $n/2$ or $n/2-1$

The next variant of the Walsh transform has the interesting feature that the basis functions for a length-n transform have only sequencies $n/2$ and $n/2-1$ at the even and odd indices, respectively. The

transform is self-inverse (the basis is shown in figure 22.9-C) and can be obtained via [FXT: `walsh_q1()` in walsh/walshq.h]

```
template <typename Type>
void walsh_q1(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    grs_negate(f, n);
    walsh_gray(f, ldn);
    revbin_permute(f, n);
}
```

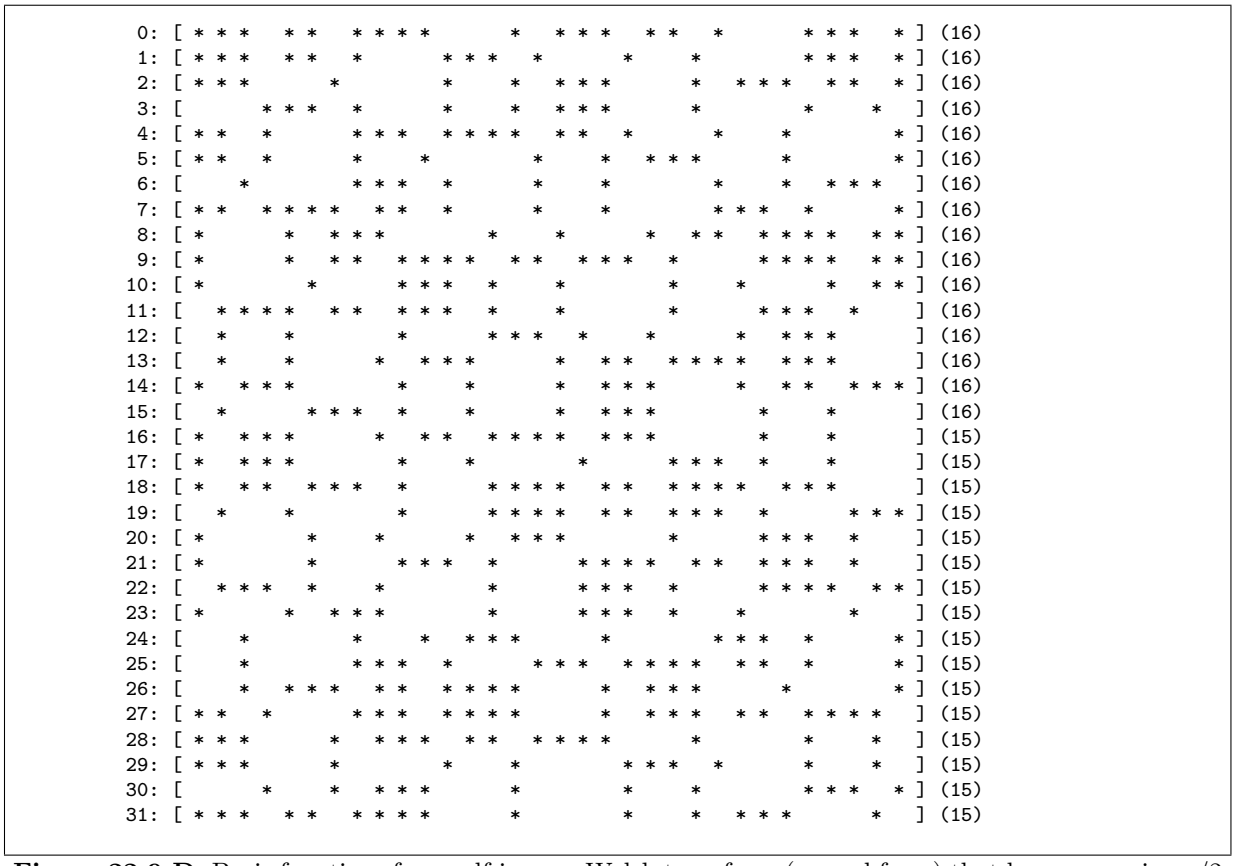


Figure 22.9-D: Basis functions for a self-inverse Walsh transform (second form) that has sequencies $n/2$ and $n/2-1$ only. Asterisks denote the value $+1$, blank entries denote -1.

A different transform with sequency $n/2$ for the first half of the basis, sequency $n/2-1$ for the second half ([FXT: `walsh_q2()` in walsh/walshq.h], basis shown in figure 22.9-D) is computed by

```
template <typename Type>
void walsh_q2(Type *f, ulong ldn)
{
    ulong n = 1UL << ldn;
    revbin_permute(f, n);
    grs_negate(f, n);
    walsh_gray(f, ldn);
// =^=
//    grs_negate(f, n);
//    revbin_permute(f, n);
//    walsh_gray(f, ldn);
}
```

One has:

$$W_{q2} \quad = \quad R\, W_{q1}\, R \tag{22.9-9}$$

The base functions of the transforms can be computed as [FXT: walsh/walshbasefunc.h]

```
template <typename Type>
void walsh_q1_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = gray_code(k);
    k = revbin(k, ld(n));
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

and

```
template <typename Type>
void walsh_q2_basefunc(Type *f, ulong n, ulong k)
{
    ulong qk = (grs_negative_q(k) ? 1 : 0);
    k = revbin(k, ld(n));
    k = gray_code(k);
    for (ulong i=0; i<n; ++i)
    {
        ulong x = i & k;
        x = parity(x);
        ulong qi = (grs_negative_q(i) ? 1 : 0);
        x ^= (qk ^ qi);
        f[i] = ( 0==x ? +1 : -1 );
    }
}
```

The function `grs_negative_q()` is described in section 1.15.5 on page 40.

22.10 Slant transform

The *slant transform* can be implemented using a Walsh Transform and just a little pre/post-processing [FXT: walsh/slant.cc]:

```
void slant(double *f, ulong ldn)
{
    walsh_wak(f, ldn);

    ulong n = 1UL<<ldn;
    for (ulong ldm=0; ldm<ldn-1; ++ldm)
    {
        ulong m = 1UL<<ldm;  // m = 1, 2, 4, 8, ..., n/4
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = a * f1 - b * f2;
            f[t2] = b * f1 + a * f2;
        }
    }
}
```

Apart from the Walsh transform only an amount of work linear with the array size has to be done: the inner loop accesses the elements in strides of 4, 8, 16, ..., 2^{n-1}.

The inverse transform is:

```
void inverse_slant(double *f, ulong ldn)
{
```

```
    ulong n = 1UL<<ldn;
    ulong ldm=ldn-2;
    do
    {
        ulong m = 1UL<<ldm;  // m = n/4, n/2, ..., 4, 2, 1
        double N = m*2, N2 = N*N;
        double a = sqrt(3.0*N2/(4.0*N2-1.0));
        double b = sqrt(1.0-a*a); // == sqrt((N2-1)/(4*N2-1));
        for (ulong j=m; j<n-1; j+=4*m)
        {
            ulong t1 = j;
            ulong t2 = j + m;
            double f1 = f[t1],  f2 = f[t2];
            f[t1] = b * f2 + a * f1;
            f[t2] = a * f2 - b * f1;
        }
    }
    while ( ldm-- );

    walsh_wak(f, ldn);
}
```

A sequency ordered version of the transform can be implemented as follows:

```
void slant_seq(double *f, ulong ldn)
{
    slant(f, ldn);

    ulong n = 1UL<<ldn;
    inverse_gray_permute(f, n);
    unzip_rev(f, n);
    revbin_permute(f, n);
}
```

This implementation can be optimized by combining the involved permutations, see [237].

The inverse is obtained by calling the inverse operations in reversed order:

```
void inverse_slant_seq(double *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    revbin_permute(f, n);
    zip_rev(f, n);
    gray_permute(f, n);

    inverse_slant(f, ldn);
}
```

22.11 Arithmetic transform

How to make a arithmetic transform out of a Walsh transform:
'*Forward: replace* `(u+v)` *by* `(u)`*, and* `(u-v)` *by* `(v-u)`*.*
Backward: replace `(u+v)` *by* `(u)`*, and* `(u-v)` *by* `(u+v)`*.* '

On to the code [FXT: walsh/arithtransform.h]:

```
template <typename Type>
void arith_transform_plus(Type *f, ulong ldn)
// Arithmetic Transform (positive sign).
// Radix-2 decimation In Frequency (DIF) algorithm.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
```

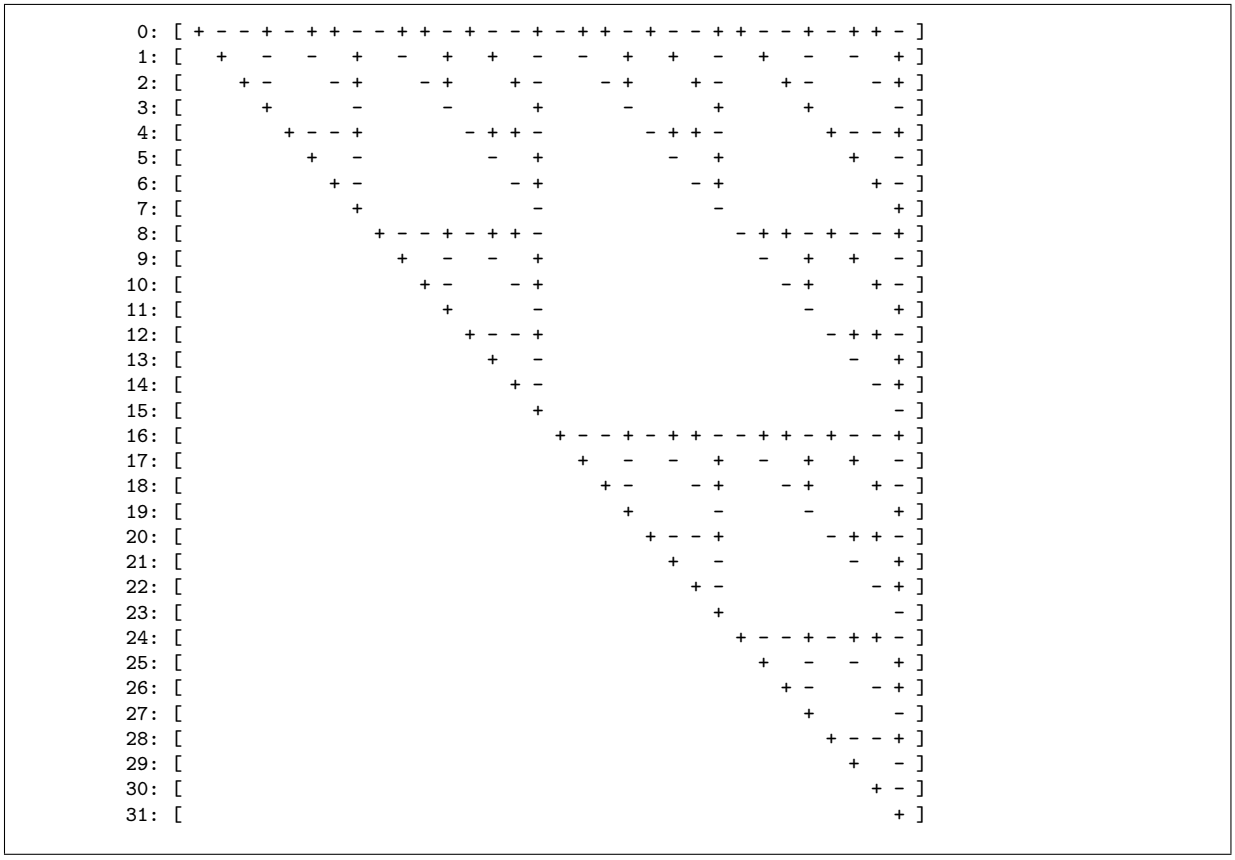


Figure 22.11-A: Basis functions for the Arithmetic transform (A^-, the one with the minus sign). The values are ± 1, blank entries denote 0.

```
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u + v;
            }
        }
    }
}
```

and

```
template <typename Type>
void arith_transform_minus(Type *f, ulong ldn)
// Arithmetic Transform (negative sign).
// Radix-2 decimation In Frequency (DIF) algorithm.
// Inverse of arith_transform_plus().
{
  [--snip--]
                f[t1] = u;
                f[t2] = v - u;
  [--snip--]
}
```

The length-2 transforms can be written as

$$A_2^+ \, v \;=\; \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \;=\; \begin{bmatrix} a \\ a+b \end{bmatrix} \tag{22.11-1}$$

$$A_2^- \, v \;=\; \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \;=\; \begin{bmatrix} a \\ b-a \end{bmatrix} \tag{22.11-2}$$

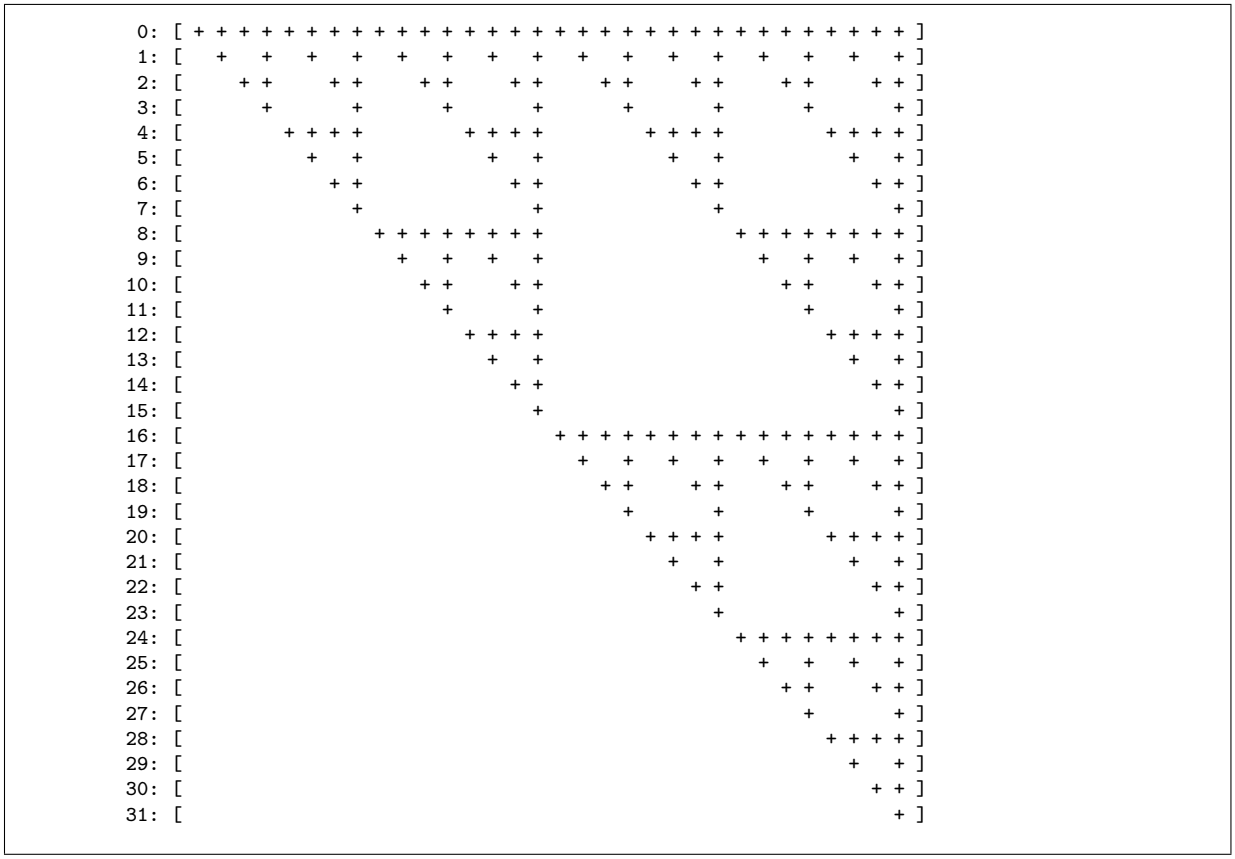


Figure 22.11-B: Basis functions for the inverse Arithmetic transform (A^+, the one without minus sign). The values are +1, blank entries denote 0.

That the transform with the minus is called the forward transform is tradition. Similar to the Fourier transform we avoid the forward- backward- naming scheme and put:

```
template <typename Type>
inline void arith_transform(Type *f, ulong ldn, int is)
{
    if ( is>0 )  arith_transform_plus(f, ldn);
    else         arith_transform_minus(f, ldn);
}
```

In Kronecker product notation (see section 22.3 on page 433) the arithmetic transform and its inverse can be written as

$$A_2^+ = \begin{bmatrix} +1 & 0 \\ +1 & +1 \end{bmatrix} \qquad A_n^+ = \bigotimes_{k=1}^{\log_2(n)} A_2^+ \tag{22.11-3a}$$

$$A_2^- = \begin{bmatrix} +1 & 0 \\ -1 & +1 \end{bmatrix} \qquad A_n^- = \bigotimes_{k=1}^{\log_2(n)} A_2^- \tag{22.11-3b}$$

The k-th element of the arithmetic A^+ transform is

$$A^+[a]_k = \sum_{i \subset k} a_i \tag{22.11-4a}$$

where $i \subset k$ means that the bits of i are a subset of the bits of k: $i \subset k \iff (i \wedge k) = i$. For the

transform A^- we have

$$A^-[a]_k \quad = \quad (-1)^{p(k)} \sum_{i \subset k} (-1)^{p(i)}\, a_i \tag{22.11-4b}$$

where $p(x)$ is the parity of x.

22.11.1 Transposed arithmetic transform

We define a transposed arithmetic transforms B^+ and B^- via

$$B_2^+ \quad = \quad \begin{bmatrix} +1 & +1 \\ 0 & +1 \end{bmatrix} \qquad B_n^+ = \bigotimes_{k=1}^{\log_2(n)} B_2^+ \tag{22.11-5a}$$

$$B_2^- \quad = \quad \begin{bmatrix} +1 & -1 \\ 0 & +1 \end{bmatrix} \qquad B_n^- = \bigotimes_{k=1}^{\log_2(n)} B_2^- \tag{22.11-5b}$$

Then the transforms are [FXT: walsh/arithtransform.h]

```
template <typename Type>
void transposed_arith_transform_plus(Type *f, ulong ldn)
{
  [--snip--]
                f[t1] = u + v;
                f[t2] = v;
  [--snip--]
}
template <typename Type>
void transposed_arith_transform_minus(Type *f, ulong ldn)
{
  [--snip--]
                f[t1] = u - v;
                f[t2] = v;
  [--snip--]
}
```

22.11.2 Relation to Walsh transform

To establish the relation to the Walsh transform recall that its decomposition as a Kronecker product is

$$W_2 \quad = \quad \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \qquad W_n = \bigotimes_{k=1}^{\log_2(n)} W_2 \tag{22.11-6}$$

Now as $(W_2\, A_2^+)\, A_2^- = W_2$ the expression in parenthesis is the matrix that transforms the 2-point arithmetic transform (with the negative sign) to the Walsh transform. Similarly, as $(\frac{1}{2}\, A_2^+\, W_2)\, W_2 = A_2^+$, the matrix leading to the conversion from Walsh to arithmetic transform can be determined. One finds:

$$\left(W_2\, A_2^+\right)\, A_2^- \quad = \quad W_2 = \quad \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix} A_2^- \tag{22.11-7a}$$

$$\left(W_2\, A_2^-\right)\, A_2^+ \quad = \quad W_2 = \quad \begin{bmatrix} 0 & +1 \\ +2 & -1 \end{bmatrix} A_2^+ \tag{22.11-7b}$$

$$\left(\frac{1}{2}\, A_2^-\, W_2\right)\, W_2 \quad = \quad A_2^- = \frac{1}{2} \begin{bmatrix} +1 & +1 \\ 0 & -2 \end{bmatrix} W_2 \tag{22.11-7c}$$

$$\left(\frac{1}{2}\, A_2^+\, W_2\right)\, W_2 \quad = \quad A_2^+ = \frac{1}{2} \begin{bmatrix} +1 & +1 \\ +2 & 0 \end{bmatrix} W_2 \tag{22.11-7d}$$

The Kronecker product of the given matrices gives the converting transform. For example, using relation 22.11-7a, define

$$T_n \;:=\; \bigotimes_{k=1}^{\log_2(n)} \begin{bmatrix} +2 & +1 \\ 0 & -1 \end{bmatrix} \tag{22.11-8}$$

Then T_n converts a arithmetic (minus sign) transform to a Walsh transform: $W_n = T_n\, A_n^-$. For the relations between the arithmetic transform, the Reed-Muller transform and the Walsh transform, see [226].

22.12 Reed-Muller transform

How to make a Reed-Muller transform out of the arithmetic transform:
'*Replace* + *and* - *by XOR, done.*'

The *Reed-Muller transform* is can be obtained from the arithmetic transform by working modulo two. The transform is self-inverse, its basis functions are identical to those of the arithmetic transform A^+, shown in figure 22.11-B on page 457. The implementation is almost identical to [FXT: `walsh_wak_dif2()` in walsh/walshwak2.h]. The only changes are

```
Walsh: f[t1] = u + v;   =-->  Reed-Muller: f[t1] = u;
Walsh: f[t2] = u - v;   =-->  Reed-Muller: f[t2] = u ^ v;
```

There we go [FXT: walsh/reedmuller.h]:

```
template <typename Type>
void word_reed_muller_dif2(Type *f, ulong ldn)
// Reed-Muller Transform.
// Radix-2 decimation in frequency (DIF) algorithm.
// Self-inverse.
// Type must have the XOR operator.
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong r=0; r<n; r+=m)
        {
            ulong t1 = r;
            ulong t2 = r+mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u;
                f[t2] = u ^ v;
            }
        }
    }
}
```

The decimation in time algorithm can be obtained from [FXT: `walsh_wak_dit2()` in walsh/walshwak2.h] by the very same changes. As given, the transforms work word-wise, if the bit-wise transform is wanted use

```
template <typename Type>
inline void bit_reed_muller(Type *f, ulong ldn)
{
    word_reed_muller_dif2(f, ldn);
    ulong n = 1UL << ldn;
    for (ulong k=0; k<n; ++k)  f[k] = yellow_code(f[k]);
}
```

The `yellow_code()` (see section 1.18 on page 45) can also be applied before the main loop. In fact, the yellow code *is* the Reed-Muller transform on a binary word.

The other 'color-transforms' of section 1.18 lead to variants of the Reed-Muller transform, the blue code gives another self-inverse transform, the red code and the green code give transforms R and E so that

$$R\,R\,R = \mathrm{id} \qquad R^{-1} = R\,R = E \tag{22.12-1}$$
$$E\,E\,E = \mathrm{id} \qquad E^{-1} = E\,E = R \tag{22.12-2}$$
$$R\,E = E\,R = \mathrm{id} \tag{22.12-3}$$

In fact, all relations given in the referenced section hold.

As can be seen from the 'atomic' matrices (relations 1.18-12c ... 1.18-12f on page 50) the four transforms corresponding to the 'color codes' are obtained by

```
Walsh:   f[t1] = u + v;    f[t2] = u - v;
   B:    f[t1] = u ^ v;    f[t2] = v;      (transposed Reed-Muller transform)
   Y:    f[t1] = u;        f[t2] = u ^ v;  (Reed-Muller transform)
   R:    f[t1] = v;        f[t2] = u ^ v;
   E:    f[t1] = u ^ v;    f[t2] = u;
```

The basis functions of the transforms are shown in figure 22.12-A.

The *transposed Reed-Muller transform* can be obtained by setting

```
Walsh: f[t1] = u + v;   =-->  transposed Reed-Muller: f[t1] = u ^ v;
Walsh: f[t2] = u - v;   =-->  transposed Reed-Muller: f[t2] = v;
```

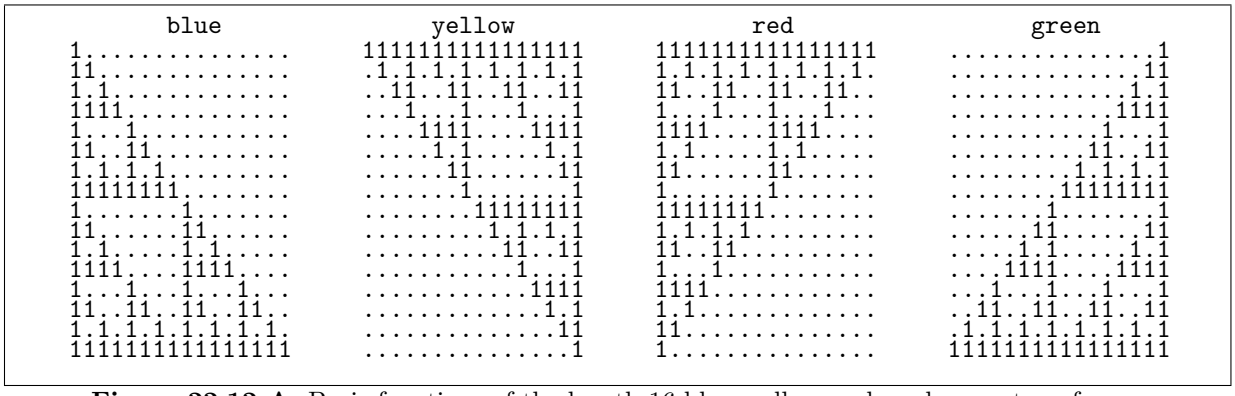
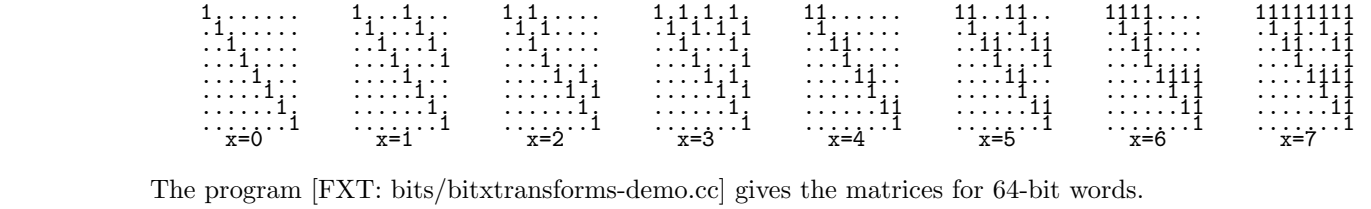


Figure 22.12-A: Basis functions of the length-16 blue, yellow, red, and green transforms.

The symbolic powering idea from section 1.18 on page 45 leads to transforms with bases (using eight element arrays and the yellow code):



The program [FXT: bits/bitxtransforms-demo.cc] gives the matrices for 64-bit words.

A function that computes the k-th base function of the transform is [FXT: walsh/reedmuller.h]:

```
template <typename Type>
inline void reed_muller_basefunc(Type *f, ulong n, ulong k)
{
    for (ulong i=0; i<n; ++i)
    {
        f[i] = ( (i & k)==k ? +1 : 0 );  // is k a bit-subset of i ?
    }
}
```

Functions that are the word-wise equivalents of the Gray code are given in [FXT: aux1/wordgray.h]:

```
template <typename Type>
void word_gray(Type *f, ulong n)
{
    for (ulong k=0;  k<n-1;  ++k)  f[k] ^= f[k+1];
}
```

and

```
void inverse_word_gray(Type *f, ulong n)
{
    ulong x = 0,  k = n;
    while ( k-- )  { x ^= f[k];  f[k] = x; }
}
```

As one might suspect, these are related to the Reed-Muller transform. Writing Y ('yellow') for the Reed-Muller transform, g for the word-wise Gray code and S_k for the cyclic shift by k words (word zero is moved to position k) one has

$$Y\,S_{+1}\,Y = g \tag{22.12-4}$$
$$Y\,S_{-1}\,Y = g^{-1} \tag{22.12-5}$$
$$Y\,S_{k}\,Y = g^{k} \tag{22.12-6}$$

These are exactly the relations 1.18-10a ... 1.18-10c on page 49 for the bit-wise transforms. For $k >= 0$ the operator S_k corresponds to the shift toward element zero (use [FXT: `rotate_sgn()` in perm/rotate.h]). The power of the word-wise Gray code is perfectly equivalent to the bit-wise version:

```
template <typename Type>
void word_gray_pow(Type *f, ulong n, ulong x)
{
    for (ulong s=1; s<n; s*=2)
    {
        if ( x & 1)
        {
            // word_gray ** s:
            for (ulong k=0, j=k+s;  j<n;  ++k,++j)  f[k] ^= f[j];
        }
        x >>= 1;
    }
}
```

Let e be the reversed Gray code operator, then we have for the transposed Reed-Muller transform B:

$$B\,S_{+1}\,B = e^{-1} \tag{22.12-7}$$
$$B\,S_{-1}\,B = e \tag{22.12-8}$$
$$B\,S_{k}\,B = e^{-k} \tag{22.12-9}$$

Further,

$$E\,S_{k}\,R = e^{k} \tag{22.12-10}$$
$$E\,e^{k}\,R = S_{k} \tag{22.12-11}$$

The transforms as Kronecker products (all operations are modulo two):

$$B_n = \bigotimes_{k=1}^{\log_2(n)} B_2 \quad \text{where} \quad B_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{22.12-12a}$$

$$Y_n = \bigotimes_{k=1}^{\log_2(n)} Y_2 \quad \text{where} \quad Y_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{22.12-12b}$$

$$R_n = \bigotimes_{k=1}^{\log_2(n)} R_2 \quad \text{where} \quad R_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{22.12-12c}$$

$$E_n = \bigotimes_{k=1}^{\log_2(n)} E_2 \quad \text{where} \quad E_2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \tag{22.12-12d}$$

22.13 The OR-convolution, and the AND-convolution

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 1:   1  1  3  3   5  5  7  7   9  9 11 11  13 13 15 15
 2:   2  3  2  3   6  7  6  7  10 11 10 11  14 15 14 15
 3:   3  3  3  3   7  7  7  7  11 11 11 11  15 15 15 15

 4:   4  5  6  7   4  5  6  7  12 13 14 15  12 13 14 15
 5:   5  5  7  7   5  5  7  7  13 13 15 15  13 13 15 15
 6:   6  7  6  7   6  7  6  7  14 15 14 15  14 15 14 15
 7:   7  7  7  7   7  7  7  7  15 15 15 15  15 15 15 15

 8:   8  9 10 11  12 13 14 15   8  9 10 11  12 13 14 15
 9:   9  9 11 11  13 13 15 15   9  9 11 11  13 13 15 15
10:  10 11 10 11  14 15 14 15  10 11 10 11  14 15 14 15
11:  11 11 11 11  15 15 15 15  11 11 11 11  15 15 15 15

12:  12 13 14 15  12 13 14 15  12 13 14 15  12 13 14 15
13:  13 13 15 15  13 13 15 15  13 13 15 15  13 13 15 15
14:  14 15 14 15  14 15 14 15  14 15 14 15  14 15 14 15
15:  15 15 15 15  15 15 15 15  15 15 15 15  15 15 15 15
```

Figure 22.13-A: Semi-symbolic scheme for the OR-convolution of two length-16 sequences.

```
 +--  0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
 |
 0:   0  0  0  0   0  0  0  0   0  0  0  0   0  0  0  0
 1:   0  1  0  1   0  1  0  1   0  1  0  1   0  1  0  1
 2:   0  0  2  2   0  0  2  2   0  0  2  2   0  0  2  2
 3:   0  1  2  3   0  1  2  3   0  1  2  3   0  1  2  3

 4:   0  0  0  0   4  4  4  4   0  0  0  0   4  4  4  4
 5:   0  1  0  1   4  5  4  5   0  1  0  1   4  5  4  5
 6:   0  0  2  2   4  4  6  6   0  0  2  2   4  4  6  6
 7:   0  1  2  3   4  5  6  7   0  1  2  3   4  5  6  7

 8:   0  0  0  0   0  0  0  0   8  8  8  8   8  8  8  8
 9:   0  1  0  1   0  1  0  1   8  9  8  9   8  9  8  9
10:   0  0  2  2   0  0  2  2   8  8 10 10   8  8 10 10
11:   0  1  2  3   0  1  2  3   8  9 10 11   8  9 10 11

12:   0  0  0  0   4  4  4  4   8  8  8  8  12 12 12 12
13:   0  1  0  1   4  5  4  5   8  9  8  9  12 13 12 13
14:   0  0  2  2   4  4  6  6   8  8 10 10  12 12 14 14
15:   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15
```

Figure 22.13-B: Semi-symbolic scheme for the AND-convolution of two length-16 sequences.

Let a and b be sequences of length a power of two. We define the *OR-convolution* h of a and b, as

$$h_\tau \quad = \quad \sum_{i \vee j = \tau} a_i \, b_j \tag{22.13-1}$$

where $\vee$ denotes bit-wise OR. The symbolic table for the OR-convolution is shown in figure 22.13-A (see figure 21.1-A on page 410 an explanation of the scheme). The OR-convolution can be computed as

$$h \quad = \quad A^- \left[\, A^+[a] \cdot A^+[b] \, \right] \tag{22.13-2}$$

where A^+ and A^- are the arithmetic transforms given in section 22.11 on page 455. An implementation is [FXT: walsh/or-convolution.h]:

```
template <typename Type>
inline void or_convolution(Type * restrict f, Type * restrict g, ulong ldn)
// Compute the OR-convolution h[] of f[] and g[]:
//  h[k] = sum(i | j == k,  f[i]*g[j])
// f[] and g[] must not overlap.
// Result written to g[].
{
    arith_transform_plus(f, ldn);
    arith_transform_plus(g, ldn);
```

```
    const ulong n = (1UL<<ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    arith_transform_minus(g, ldn);
}
```

We also have

$$h \;=\; Y^{-1}\,[\,Y[a]\cdot Y[b]\,] \;=\; Y\,[\,Y[a]\cdot Y[b]\,] \tag{22.13-3}$$

where Y is the Reed-Muller transform given in section 22.12 on page 459.

Define the *AND-convolution* h of two sequences a and b as

$$h_\tau \;=\; \sum_{i\wedge j=\tau} a_i\,b_j \tag{22.13-4}$$

where $\wedge$ denotes the bit-wise AND. The symbolic scheme is shown in figure 22.13-B. The AND-convolution can be computed as

$$h \;=\; B^-\,\big[\,B^+[a]\cdot B^+[b]\,\big] \tag{22.13-5}$$

where B^+ and B^- are the transposed arithmetic transforms. The implementation of the AND-convolution is [FXT: walsh/and-convolution.h]:

```
template <typename Type>
inline void and_convolution(Type * restrict f, Type * restrict g, ulong ldn)
// Compute the AND-convolution h[] of f[] and g[]:
//  h[k] = sum(i & j == k,  f[i]*g[j])
// f[] and g[] must not overlap.
// Result written to g[].
{
    transposed_arith_transform_plus(f, ldn);
    transposed_arith_transform_plus(g, ldn);
    const ulong n = (1UL<<ldn);
    for (ulong k=0; k<n; ++k)  g[k] *= f[k];
    transposed_arith_transform_minus(g, ldn);
}
```

The same is true modulo two:

$$h \;=\; B^{-1}\,[\,B[a]\cdot B[b]\,] \;=\; B\,[\,B[a]\cdot B[b]\,] \tag{22.13-6}$$

here B is the transposed Reed-Muller transform.

Chapter 23

The Haar transform

Haar transforms are invertible transforms that do not involve trigonometric factors. We present several variants of the transform whose computation involves just proportional n operations. Haar transforms can be used as building blocks of the Walsh transform. We give two non-standard splitting schemes for Haar transforms, based on the Fibonacci and Mersenne numbers.

23.1 The 'standard' Haar transform

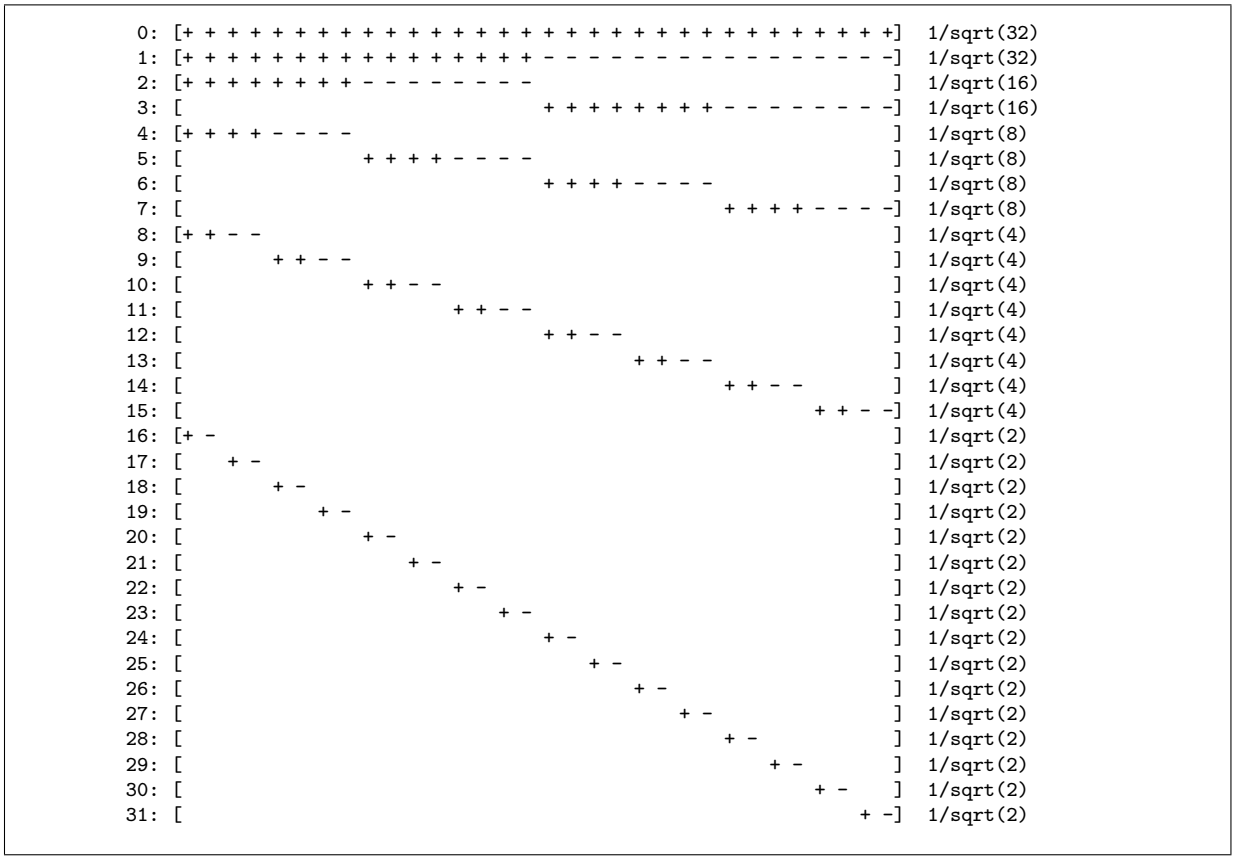


Figure 23.1-A: Basis functions for the Haar transform. Only the signs of the nonzero entries are shown. The absolute value of the nonzero entries in each row is given at the right. The norm of each row is one.

The *Haar transform* of a length-n sequence f consists of $\log_2(n)$ steps were the sums and differences of adjacent pairs of elements f_{2j}, f_{2j+1} are computed. The sums are then written to the lower half of the array f, the differences to the higher half. Ignoring the order (and normalization), each step corresponds to a matrix multiplication:

$$\begin{bmatrix} +1 & +1 & & & & & & & \\ +1 & -1 & & & & & & & \\ & & +1 & +1 & & & & & \\ & & +1 & -1 & & & & & \\ & & & & +1 & +1 & & & \\ & & & & +1 & -1 & & & \\ & & & & & & +1 & +1 & \\ & & & & & & +1 & -1 & \\ & & & & & & & & \ddots \end{bmatrix} \cdot \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ \vdots \end{bmatrix} \tag{23.1-1}$$

The step is applied to the full array, then to the lower half, the lower quarter, ..., the lower four elements, the lowest pair. (The array length n must be a power of two.) The computational cost of the transform is proportional $n + n/2 + n/4 + \ldots + 4 + 2$ which is $\sim O(n)$. The basis functions for the Haar transform have finite support, they are shown in figure 23.1-A.

The following implementation involves $2n$ multiplications $\sqrt{2}$ which make the transform orthogonal, corresponding to a scalar factor of $\sqrt{2}$ in the relation 23.1-1:

```
template <typename Type>
void haar(Type *f, ulong ldn)
{
    ulong n = (1UL<<ldn);
    const Type s2 = sqrt(0.5);   // normalization factor
    Type *g = new Type[n];       // scratch space
    for (ulong m=n; m>1; m>>=1)  // n, n/2, n/4, n/8, ..., 4, 2
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)  // sums and differences of adjacent pairs
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    = (x + y) * s2;   // sums to lower half
            g[mh+k] = (x - y) * s2;   // differences to higher half
        }
        copy(g, f, m);
    }
    delete [] g;
}
```

We can reduce the number of multiplications to n by delaying the multiplies with the sums [FXT: `haar()` in haar/haar.h]:

```
template <typename Type>
void haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        v *= s2;
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] = (x - y) * v;
        }
```

```
        copy(g, f, m);
    }
    f[0] *= v; // v == 1.0/sqrt(n);
    if ( !ws )  delete [] g;
}
```

The temporary workspace can be supplied by the caller.

The inverse Haar transform is obtained by applying the inverse steps in reversed order [FXT: `inverse_haar()` in haar/haar.h]:

```
template <typename Type>
void inverse_haar(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}
```

A generalization of the steps used in the Haar transform leads to the wavelet transforms treated in chapter 26 on page 515.

23.2 In-place Haar transform

The 'standard' Haar transform routines are not in-place, they use a temporary storage. A rather simple reordering of the basis functions, however, allows for to an in-place algorithm [FXT: `haar_inplace()` in haar/haar.h]:

```
template <typename Type>
void haar_inplace(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(0.5);
    Type v = 1.0;
    for (ulong js=2; js<=n; js<<=1)
    {
        v *= s2;
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  = (x - y) * v;
        }
    }
    f[0] *= v; // v==1.0/sqrt(n);
}
```

The inverse is [FXT: `inverse_haar_inplace()` in haar/haar.h]:

```
template <typename Type>
void inverse_haar_inplace(Type *f, ulong ldn)
```

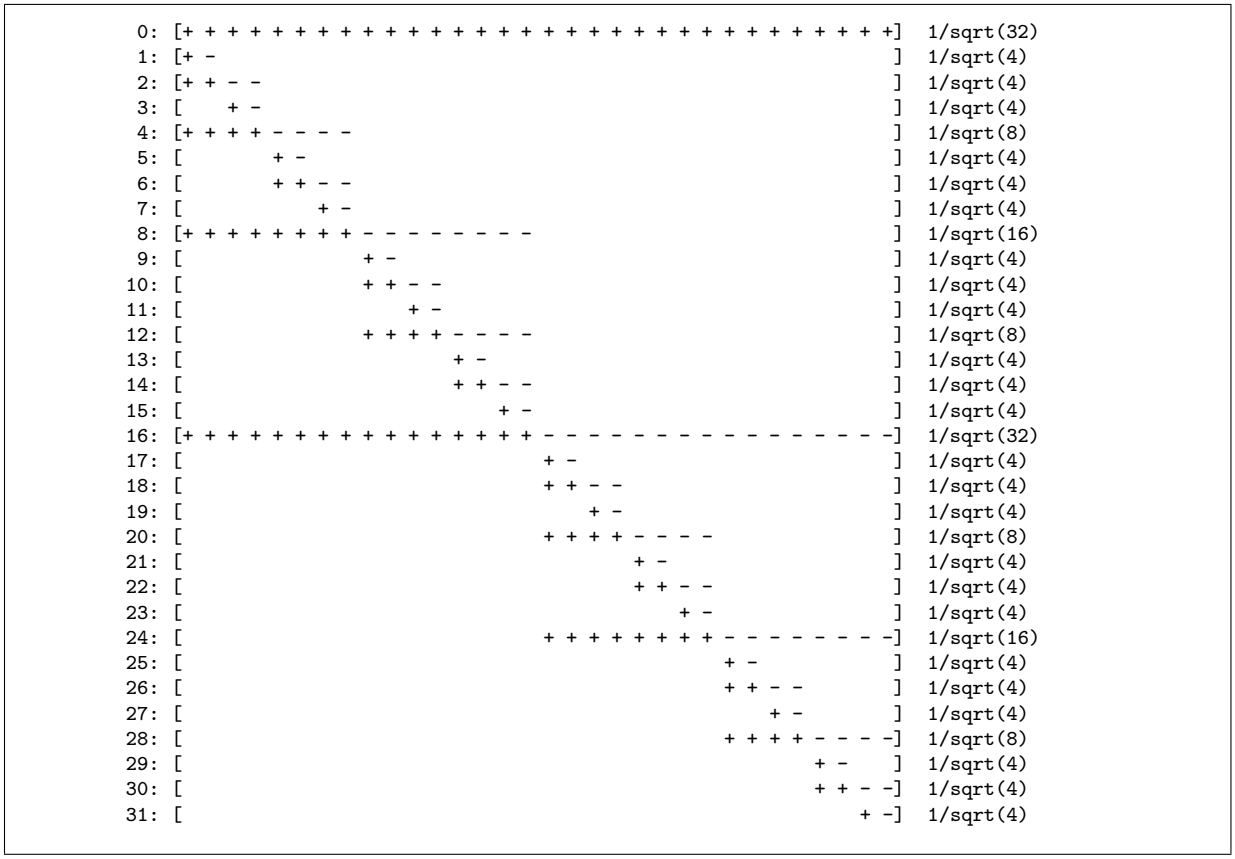


Figure 23.2-A: Haar basis functions, in-place order. Only the signs of the nonzero entries are shown. The absolute value of the nonzero entries in each row is given at the right. The norm of each row is one.

```
{
    ulong n = 1UL<<ldn;
    Type s2 = sqrt(2.0);
    Type v = 1.0/sqrt(n);
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
        v *= s2;
    }
}
```

The in-place Haar transform H_i is related to the 'usual' Haar transform H by a permutation P_H via the relations

$$H = P_H \cdot H_i \tag{23.2-1}$$

$$H^{-1} = H_i^{-1} \cdot P_H^{-1} \tag{23.2-2}$$

The permutation P_H can be programmed as

```
template <typename Type>
void haar_permute(Type *f, ulong n)
{
    revbin_permute(f, n);
    for (ulong m=4; m<=n/2; m*=2)  revbin_permute(f+m, m);
```

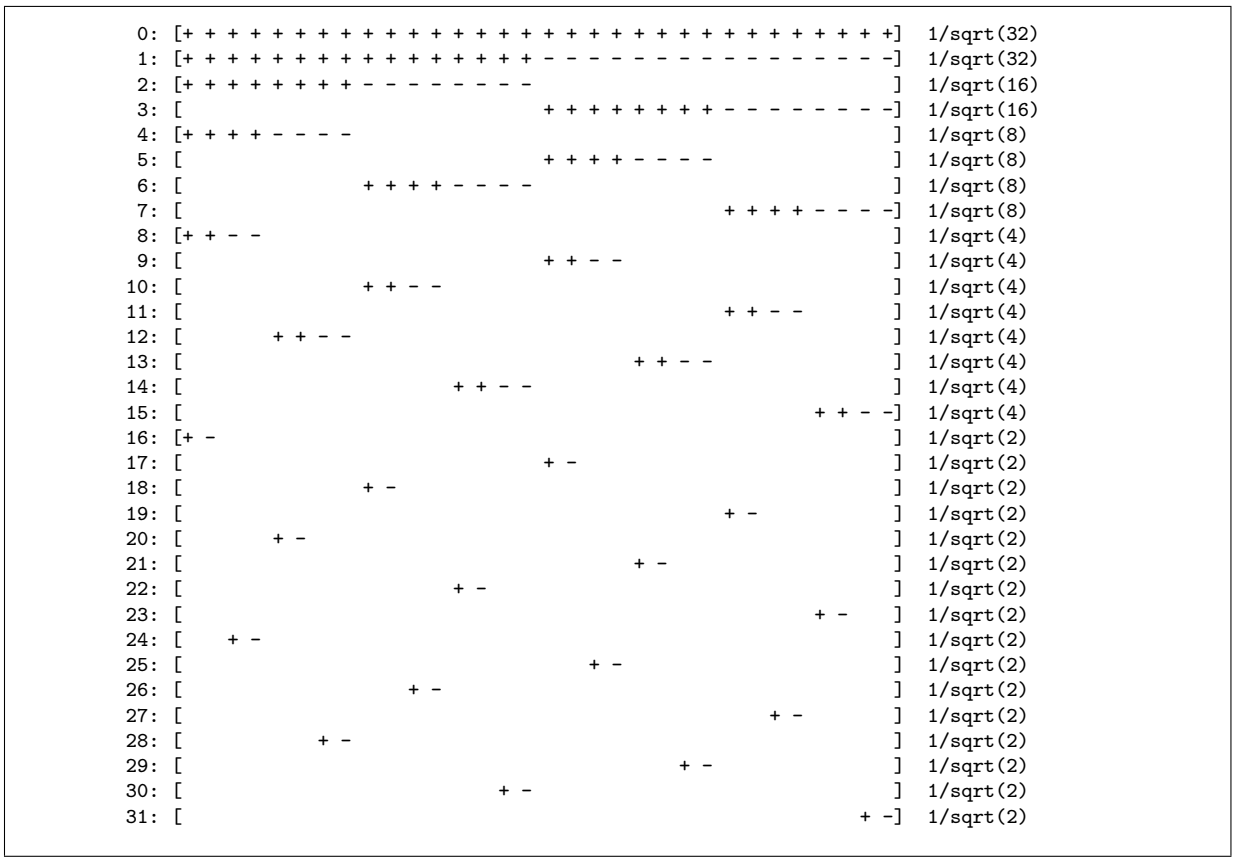


Figure 23.2-B: Basis functions of the in-place order Haar transform followed by a revbin permutation. The ordering is such that basis functions that are identical up to a shift appear consecutively.

```
}
```

The revbin permutations in the loop do not overlap, so the inverse Haar permutation is obtained by simply swapping the loop with the full-length revbin permutation [FXT: perm/haarpermute.h]:

```
template <typename Type>
void inverse_haar_permute(Type *f, ulong n)
{
    for (ulong m=4; m<=n/2; m*=2)  revbin_permute(f+m, m);
    revbin_permute(f, n);
}
```

Then, as given above, `haar` is equivalent to

```
    haar_inplace();  haar_permute();
```

and `inverse_haar` is equivalent to

```
    inverse_haar_permute();  inverse_haar_inplace();
```

23.3 Non-normalized Haar transforms

Versions of the Haar transform without normalization are given in [FXT: haar/haarnn.h]. The basis functions are the same as for the normalized versions, only the absolute value of the nonzero entries are different.

```
template <typename Type>
void haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
```

```
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j];
            Type y = f[j+1];
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The inverse is

```
template <typename Type>
void inverse_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type s2 = 2.0;
    Type v = 1.0/n;
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    f[0] *= v;
    for (ulong m=2; m<=n; m<<=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k] * v;
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
        v *= s2;
    }
    if ( !ws )  delete [] g;
}
```

An unnormalized transform that works in-place is

```
template <typename Type>
void haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

The inverse routine is

```
template <typename Type>
void inverse_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    Type s2 = 2.0;
    Type v = 1.0/n;
    f[0] *= v;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
```

```
            Type x = f[j];
            Type y = f[t] * v;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
        v *= s2;
    }
}
```

The sequence of statements `{ haar_inplace_nn(); haar_permute(); }` is equivalent to `{ haar_nn(); }`. The sequence `{ inverse_haar_permute(); inverse_haar_inplace(); }` is equivalent to `{ inverse_haar(); }`.

23.4 Transposed Haar transforms

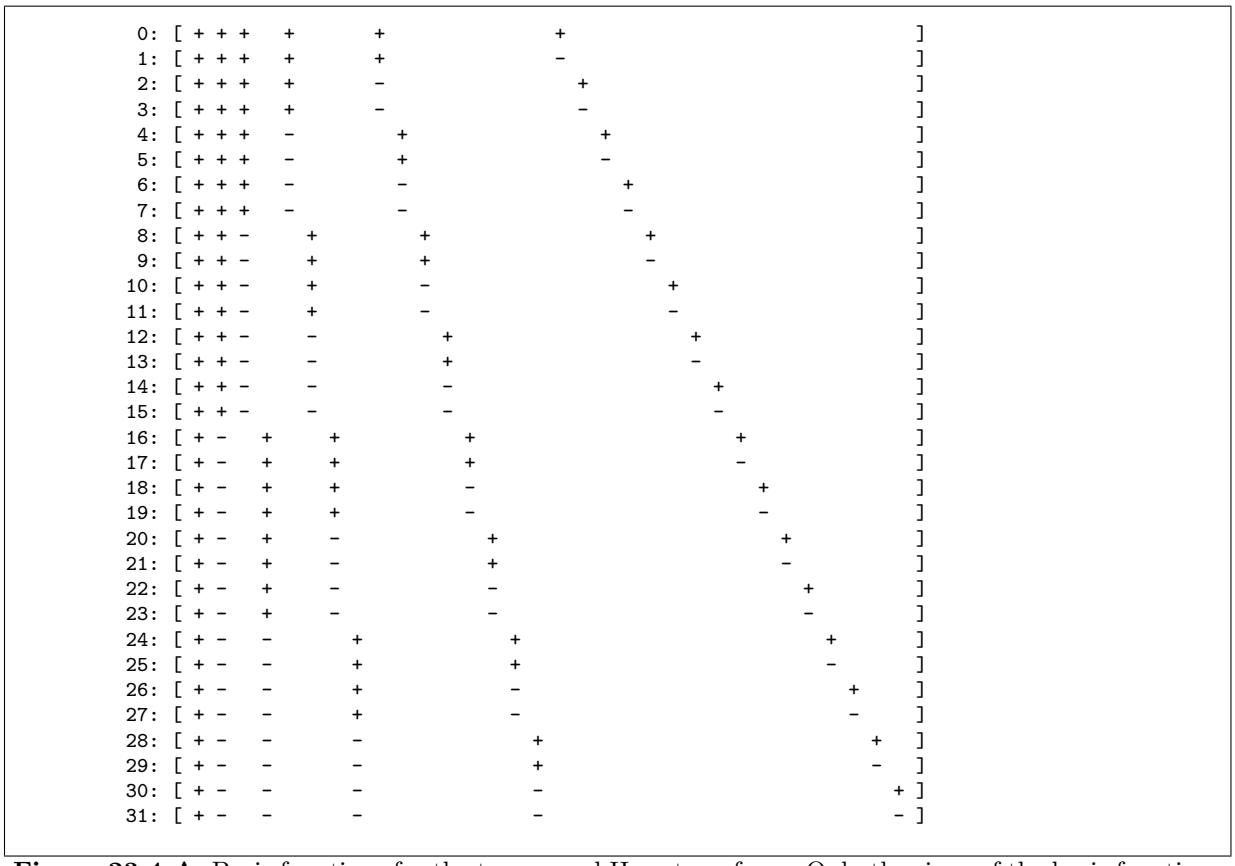


Figure 23.4-A: Basis functions for the transposed Haar transform. Only the signs of the basis functions are shown. At the blank entries the functions are zero.

Figure 23.4-A shows the bases functions of the transposed Haar transform. The shown routine are given in [FXT: haar/transposedhaarnn.h]. The following routine does an unnormalized Haar transform. The result is, up to normalization, the same as with `inverse_haar()`. The implementation uses a scratch array:

```
template <typename Type>
void transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=2; m<=n; m<<=1)
    {
```

```
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[k];
            Type y = f[mh+k];
            g[j]    =  x + y;
            g[j+1]  =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_nn(Type *f, ulong ldn, Type *ws=0)
{
    ulong n = (1UL<<ldn);
    Type *g = ws;
    if ( !ws )  g = new Type[n];
    for (ulong m=n; m>1; m>>=1)
    {
        ulong mh = (m>>1);
        for (ulong j=0, k=0;  j<m;  j+=2, k++)
        {
            Type x = f[j]   * 0.5;
            Type y = f[j+1] * 0.5;
            g[k]    =  x + y;
            g[mh+k] =  x - y;
        }
        copy(g, f, m);
    }
    if ( !ws )  delete [] g;
}
```

The next routine is equivalent to the sequence of statements `{ inverse_haar_permute(); transposed_haar_inplace_nn(); }`. Its advantage is that no scratch array is needed:

```
template <typename Type>
void transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=n; js>=2; js>>=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j];
            Type y = f[t];
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_inplace_nn(Type *f, ulong ldn)
{
    ulong n = 1UL<<ldn;
    for (ulong js=2; js<=n; js<<=1)
    {
        for (ulong j=0, t=js>>1;  j<n;  j+=js, t+=js)
        {
            Type x = f[j] * 0.5;
            Type y = f[t] * 0.5;
            f[j]  =  x + y;
            f[t]  =  x - y;
        }
    }
}
```

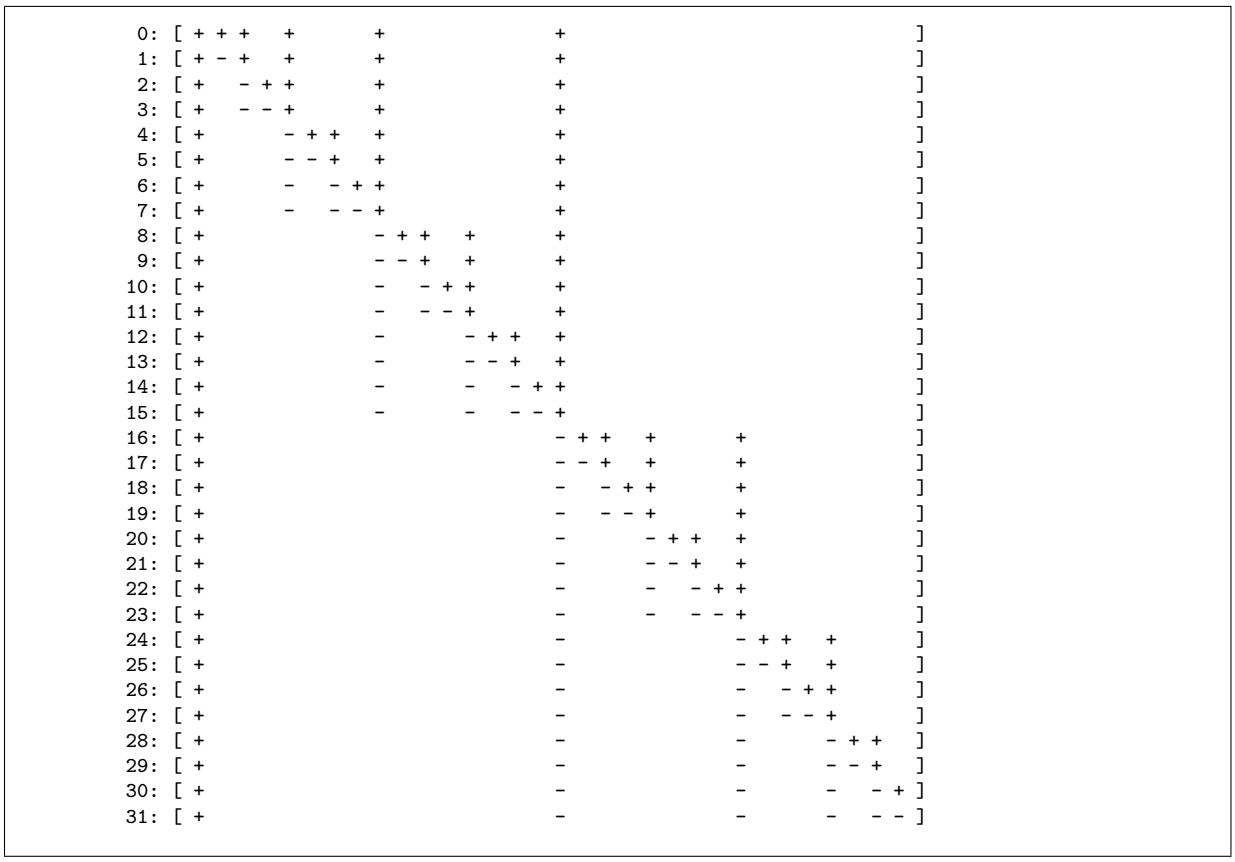


Figure 23.4-B: Basis functions for the transposed in-place Haar transform. Only the signs of the basis functions are shown. At the blank entries the functions are zero.

23.5 The reversed Haar transform

Let H_{ni} denote the non-normalized in-place Haar transform (`haar_inplace_nn`), Let H_{tni} denote the transposed non-normalized in-place Haar transform (`transposed_haar_inplace_nn`), R the revbin permutation, $\overline{H}$ the reversed Haar transform and $\overline{H}_t$ the transposed reversed Haar transform. Then

$$\overline{H} = R\,H_{ni}\,R \tag{23.5-1a}$$
$$\overline{H}_t = R\,H_{tni}\,R \tag{23.5-1b}$$
$$\overline{H}^{-1} = R\,H_{ni}^{-1}\,R \tag{23.5-1c}$$
$$\overline{H}_t^{-1} = R\,H_{tni}^{-1}\,R \tag{23.5-1d}$$

Code for the reversed Haar transform [FXT: `haar_rev_nn()` in haar/haarrevnn.h]:

```
template <typename Type>
void haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
```

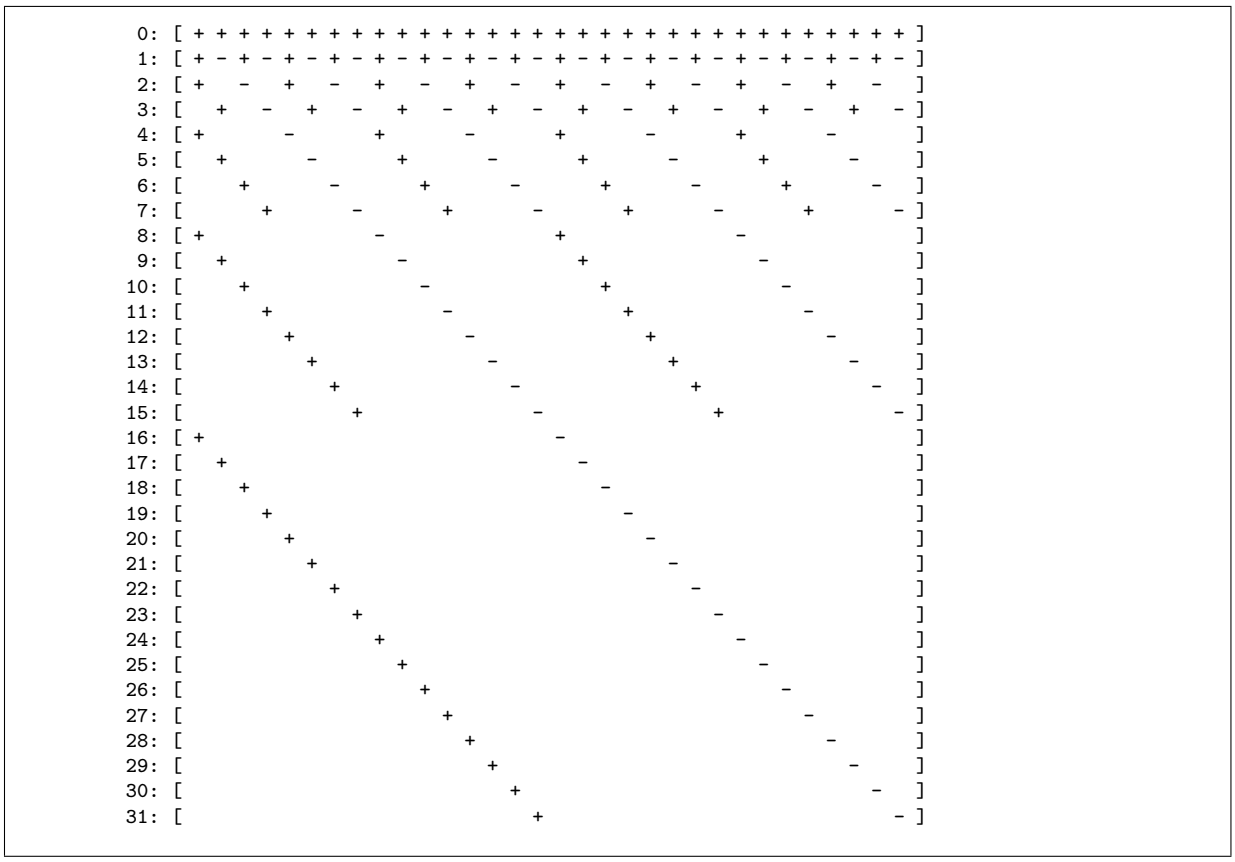


Figure 23.5-A: Basis functions for the reversed Haar transform. Only the signs of the nonzero entries are shown.

```
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

Note that this is almost the radix-2 DIF implementation for the Walsh transform The only change is that the line `for (ulong r=0; r<n; r+=m)` was replaced by `ulong r = 0`. The transform can also be computed via the following sequence of statements: `{ revbin_permute(); haar_inplace_nn(); revbin_permute(); }`.

The inverse transform is obtained by the equivalent modification with the DIT implementation for the Walsh transform and normalization:

```
template <typename Type>
void inverse_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
```

```
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The reversed transposed Haar transform is, up to normalization, the inverse of `haar_rev_nn()`. It is given in [FXT: haar/transposedhaarrevnn.h]:

```
template <typename Type>
void transposed_haar_rev_nn(Type *f, ulong ldn)
{
    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dit2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1];
                Type v = f[t2];
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

The same result would be obtained by the following sequence of statements: `{ revbin_permute(); transposed_haar_inplace_nn(); revbin_permute(); }`. The inverse transform is

```
template <typename Type>
void inverse_transposed_haar_rev_nn(Type *f, ulong ldn)
{
//    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        ulong r = 0;
//        for (ulong r=0; r<n; r+=m)  // almost walsh_wak_dif2()
        {
            ulong t1 = r;
            ulong t2 = r + mh;
            for (ulong j=0;  j<mh;  ++j, ++t1, ++t2)
            {
                Type u = f[t1] * 0.5;
                Type v = f[t2] * 0.5;
                f[t1] = u + v;
                f[t2] = u - v;
            }
        }
    }
}
```

23.6 Relations between Walsh and Haar transforms

23.6.1 Walsh transforms from Haar transforms

A length-n Walsh transform can be obtained from one length-n Haar transform, one transform of length-$\frac{n}{2}$, two transforms of length-$\frac{n}{4}$, four transforms of length-$\frac{n}{8}$, ... and $\frac{n}{4}$ transforms of length-2. Using the reversed Haar transform the implementation is most straightforward: A Walsh transform (W_k, the one with the Walsh Kronecker base) can be implemented as

```
Haar transforms:
    H(16)                  H(8)          H(4)      H(2)
  AAAAAAAAaaaaaaaa        BBBBbbbb       CCcc      Dd
  AAAAaaaa                BBbb           Cc
  AAaa                    Bb
  Aa

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AAAAAAAAaaaaaaaa
  AAAAaaaaBBBBbbbb
  AAaaCCccBBbbCCcc
  AaDdCcDdBbDdCcDd
```

Figure 23.6-A: Symbolic description of how to build a Walsh transform from Haar transforms.

```
Transposed Haar transforms:
    H(16)                  H(8)          H(4)      H(2)
  Aa
  AAaa                    Bb
  AAAAaaaa                BBbb           Cc
  AAAAAAAAaaaaaaaa        BBBBbbbb       CCcc      Dd

Walsh(16) =^= 1*H(16) + 1*H(8) + 2*H(4) + 4*H(2)
  AaDdCcDdBbDdCcDd
  AAaaCCccBBbbCCcc
  AAAAaaaaBBBBbbbb
  AAAAAAAAaaaaaaaa
```

Figure 23.6-B: Symbolic description of how to build a Walsh transform from Haar transforms, transposed version.

```
// algorithm WH1:
ulong n = 1UL<<ldn;
haar_rev_nn(f, ldn);
for (ulong ldk=ldn-1; ldk>0; --ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  haar_rev_nn(f+j, ldk);
}
```

The idea, as a symbolic scheme, is shown in figure 23.6-A. The scheme obtained by reversing the order of the lines is shown in figure 23.6-B. It corresponds to the computation of W_k using the transposed version of the Haar transform:

```
// algorithm WH1T:
ulong n = 1UL<<ldn;
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    for (ulong j=k; j<n; j+=2*k)  transposed_haar_rev_nn(f+j, ldk);
}
transposed_haar_rev_nn(f, ldn);
```

Two more methods are obtained by reversing the individual lines of the schemes seen so far, see figure 23.6-C. These correspond to the computation of the inverse Walsh transform ($W_k^{-1} = \frac{1}{n} W_k$) either as

```
// algorithm WH2T:
ulong n = 1UL<<ldn;
inverse_transposed_haar_rev_nn(f, ldn);
for (ulong ldk=ldn-1; ldk>0; --ldk)
```

Figure 23.6-C: Symbolic scheme of the four versions of the computation of the Walsh transform via Haar transforms.

```
    {
        ulong k = 1UL << ldk;
        for (ulong j=k; j<n; j+=2*k)  inverse_transposed_haar_rev_nn(f+j, ldk);
    }
```

or as

```
    // algorithm WH2:
    ulong n = 1UL<<ldn;
    for (ulong ldk=1; ldk<ldn; ++ldk)
    {
        ulong k = 1UL << ldk;
        for (ulong j=k; j<n; j+=2*k)  inverse_haar_rev_nn(f+j, ldk);
    }
    inverse_haar_rev_nn(f, ldn);
```

23.6.2 Haar transforms from Walsh transforms

The ($\sim n \log(n)$) schemes given here are not a efficient method to compute the Haar transform (which is $\sim n$). Instead, they can be used to identify the type of Haar transform that is the building block of a given Walsh transform.

The non-normalized transposed reversed Haar transform can (up to normalization) be obtained via

```
    // algorithm HW1:  transposed_haar_rev_nn(f, ldn); =^=
    for (ulong ldk=1; ldk<ldn; ++ldk)
    {
        ulong k = 1UL << ldk;
        walsh_wak(f+k, ldk);
    }
    walsh_wak(f, ldn);
```

and its inverse as

```
    // algorithm HW1I:  inverse_transposed_haar_rev_nn(f, ldn); =^=
    walsh_wak(f, ldn);
    for (ulong ldk=1; ldk<ldn; ++ldk)
    {
        ulong k = 1UL << ldk;
        walsh_wak(f+k, ldk);
    }
```

The non-normalized transposed Haar transform can (again, up to normalization) be obtained via

```
    // algorithm HW2:  transposed_haar_nn(f, ldn); =^=
    for (ulong ldk=1; ldk<ldn; ++ldk)
    {
```

```
Walsh transform:
        W(16)
      AaDdCcDdBbDdCcDd
      AAaaCCccBBbbCCcc
      AAAAaaaaBBBBbbbb
      AAAAAAAAaaaaaaaa

Inverse (or transposed) Walsh transforms:
       W(8):           W(4):      W(2):
      BBBBbbbb           CCcc        Dd
      BBbbCCcc           CcDd
      BbDdCcDd


                                      BBBBbbbb
                                    CCccBBbbCCcc
                                  DdCcDdBbDdCcDd
      Aa                         AaDdCcDdBbDdCcDd
      AAaa                       AAaaCCccBBbbCCcc
      AAAAaaaa                   AAAAaaaaBBBBbbbb
      AAAAAAAAaaaaaaaa           AAAAAAAAaaaaaaaa
        Haar(16)      =^=  W(16) + W(8) + W(4) + W(2)
```

Figure 23.6-D: Symbolic description of how to build a Haar transform from Walsh transforms.

```
    ulong k = 1UL << ldk;
    walsh_pal(f+k, ldk);
}
walsh_pal(f, ldn);
```

and its inverse as

```
// algorithm HW2I:  inverse_transposed_haar_nn(f, ldn); =^=
walsh_pal(f, ldn);  // =^= revbin_permute(f, n); walsh_wak(f, ldn);
for (ulong ldk=1; ldk<ldn; ++ldk)
{
    ulong k = 1UL << ldk;
    walsh_pal(f+k, ldk);
}
```

The symbolic scheme is given in figure 23.6-D.

23.7 Nonstandard splitting schemes *

All radix-2 transforms recursively split the length of the array into halves. The size of the transforms is limited to powers of two. In a recursive implementation we use the fact that $2^k = 2^{k-1} + 2^{k-1}$. With $N_k := 2^k$ we have $N_0 = 1$, and $N_k = N_{k-1} + N_{k-1}$. We use different recursive schemes to derive nonstandard variants of the Haar and Walsh transforms.

23.7.1 Fibonacci-Haar and Fibonacci-Walsh transform

One can use the Fibonacci numbers $F_n = F_{n-1} + F_{n-1}$ (where $F_0 = 0$ and $F_1 = 1$) to construct a *Fibonacci-Haar transform* as follows [FXT: haar/fib-haar.h]:

```
inline void fibonacci_haar(double *a, ulong f0, ulong f1)
// In-place Fibonacci-Haar transform of a[0,...,f0-1].
```

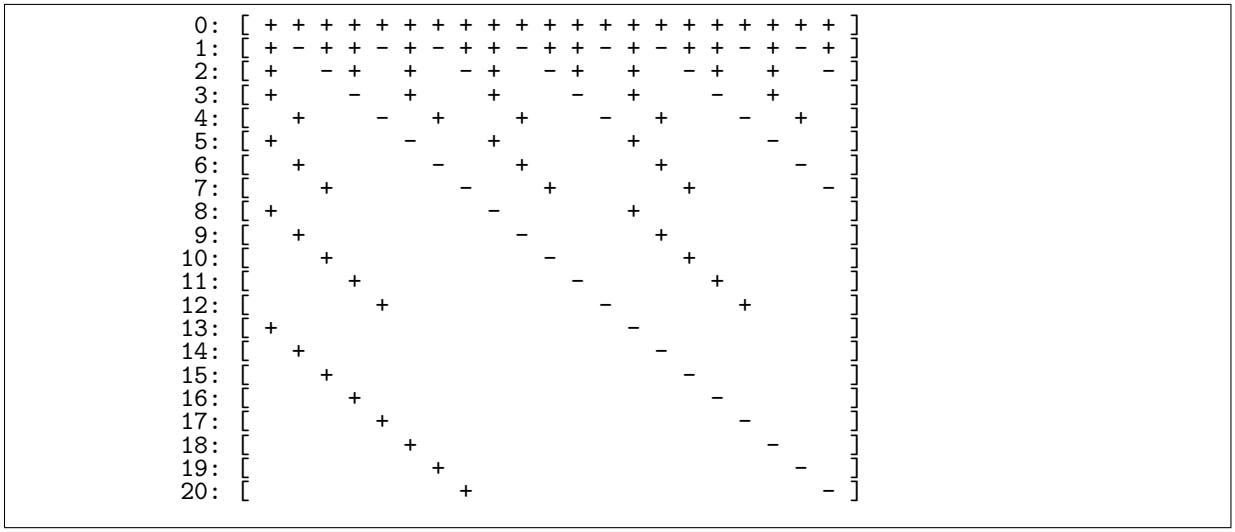


Figure 23.7-A: Basis functions for the non-normalized Fibonacci-Haar transform. Only the signs of the nonzero entries are shown. At the blank entries the functions are zero.

```
// f0 must be a Fibonacci number, f1 the next smaller Fibonacci number.
{
    if ( f0 < 2 )  return;
    ulong f2 = f0 - f1;
    for (ulong j=0,k=f1;  j<f2;  ++j,++k)
    {
        double u = a[j],  v = a[k];
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    }
    fibonacci_haar(a, f1, f2);
}
```

A non-normalized version is obtained by omitting the multiplications with $1/\sqrt{2}$ (=`SQRT1_2`). The basis functions for the non-normalized transform with length-21 (= F_8) are shown in figure 23.7-A (compare to figure 23.5-A on page 474). The second row corresponds to the *rabbit sequence* described in section 36.11 on page 718. Figure 23.7-A was created with the program [FXT: fft/fib-haar-demo.cc].

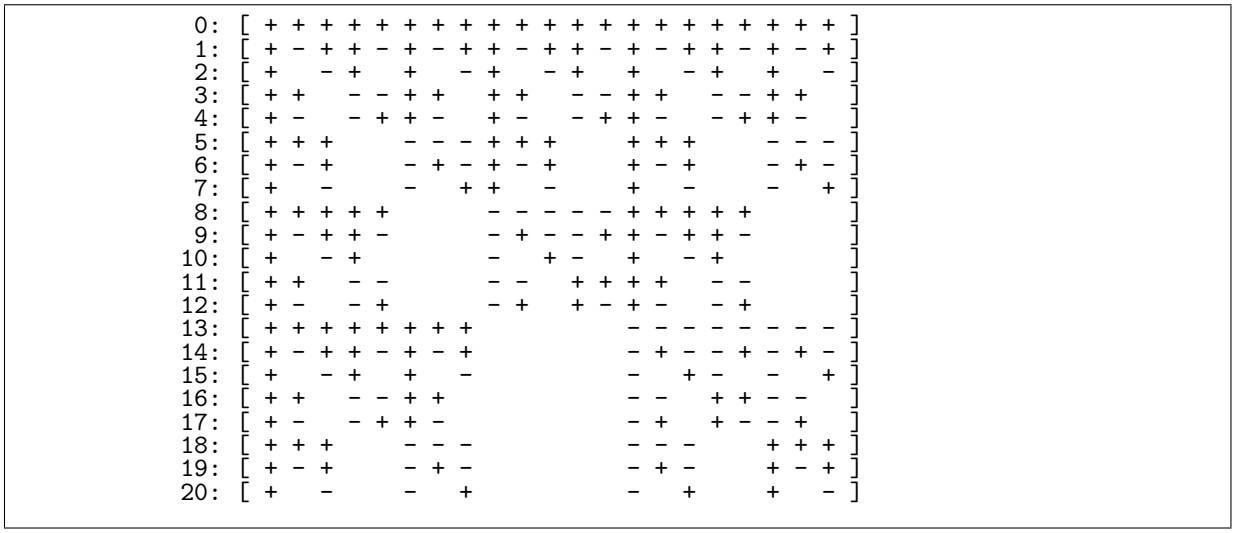


Figure 23.7-B: Basis functions for the non-normalized Fibonacci-Walsh transform.

A *Fibonacci-Walsh transform* can be obtained by adding one line in the recursive implementation of the

Fibonacci-Haar transform [FXT: walsh/fib-walsh.h]:

```
inline void fibonacci_walsh(double *a, ulong f0, ulong f1)
// In-place Fibonacci-Walsh transform of a[0,...,f0-1].
// f0 must be a Fibonacci number, f1 the next smaller Fibonacci number.
{
    if ( f0 < 2 )  return;
    ulong f2 = f0 - f1;
    for (ulong j=0,k=f1;  j<f2;  ++j,++k)
    {
        double u = a[j],  v = a[k];
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    }
    fibonacci_walsh(a, f1, f2);
    fibonacci_walsh(a+f1, f2, f1-f2);  // <--= omit line to obtain Haar transform
}
```

The basis functions for the length 21 transform are shown in figure 23.7-B which was created with the program [FXT: fft/fib-walsh-demo.cc].

The given routines can be optimized by inserting short-length transforms as recursion end.

One can obtain Haar-like and Walsh-like transforms for any linear recursive sequence that is increasing. A construction for recurrences $N_k = N_{k-1} + N_{k-1-p}$ is considered in [106].

23.7.2 Mersenne-Haar and Mersenne-Walsh transform

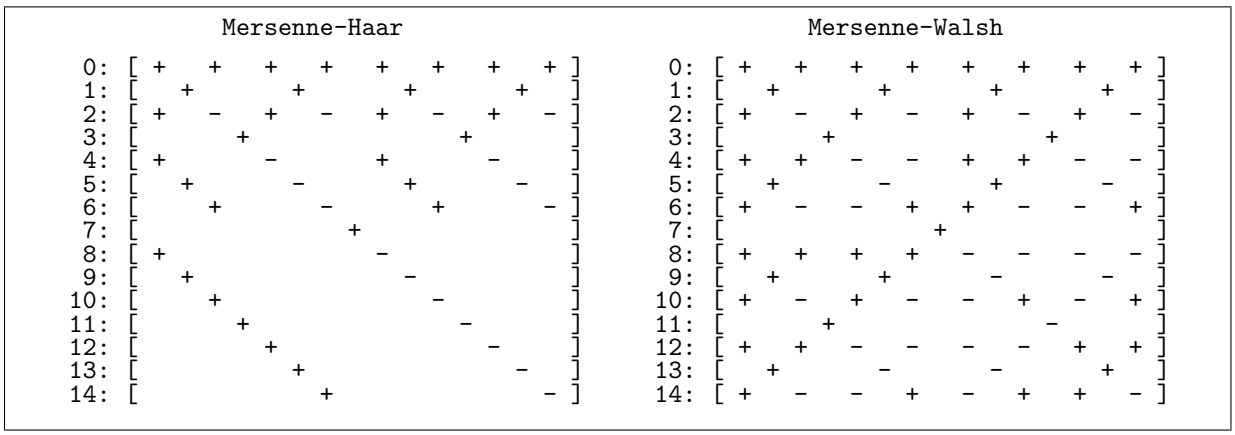


Figure 23.7-C: Basis functions for the non-normalized Mersenne-Haar transform (left), and Mersenne-Walsh transform (right). Only the signs of the nonzero entries are shown. At the blank entries the functions are zero.

For the Mersenne numbers $M_k = 2^k - 1$ we have the recursion $M_k = 2 \cdot M_{k-1} + 1$. This can be used to obtain a *Mersenne-Walsh transform* [FXT: walsh/mers-walsh.h]:

```
inline void mersenne_walsh(double *a, ulong f0)
// In-place Mersenne-Walsh transform of a[0,...,f0-1].
// f0 must be a Mersenne number.
// Self-inverse.
{
    if ( f0 < 2 )  return;
    ulong f1 = f0 >> 1;  // next smaller Mersenne number
    for (ulong j=0,k=f1+1;  j<f1;  ++j,++k)
    {
        double u = a[j],  v = a[k];
        a[j] = (u+v) * SQRT1_2;
        a[k] = (u-v) * SQRT1_2;
    }
    mersenne_walsh(a, f1);
```

```
    mersenne_walsh(a+f1+1, f1);  // <--= omit line to obtain Mersenne-Haar transform
}
```

Figure 23.7-C (right) gives the basis functions for the non-normalized Mersenne-Walsh transform. The Mersenne-Haar transform is obtained by deleting one line as indicated. The implementation is given in [FXT: haar/mers-haar.h], the basis functions of the non-normalized version are shown at the left of figure 23.7-C. The figure was created with the programs [FXT: fft/mers-walsh-demo.cc] and [FXT: fft/mers-haar-demo.cc]. Note that both transforms leave the central element unchanged.

Chapter 24

The Hartley transform

The Hartley transform is a trigonometric transform whose practical importance comes from the fact that it maps real data to real data. While the fast algorithms for radix-2 can be found without great difficulty the higher radix algorithms are not obvious. Therefore it is appropriate to describe the Hartley transform in terms of the Fourier transform. A method for the conversion of FFT algorithms to fast Hartley transform (FHT) algorithms is given.

Routines for the conversion of Hartley transforms to and from Fourier transforms are described. Convolution routines based on the FHT are given for complex and real valued data. An efficient procedure for the computation of the negacyclic convolution is described.

24.1 Definition and symmetries

The *discrete Hartley transform* (HT) of a length-n sequence a is defined as

$$c = \mathcal{H}[a] \tag{24.1-1a}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x \left(\cos \frac{2\,\pi\,k\,x}{n} + \sin \frac{2\,\pi\,k\,x}{n} \right) \tag{24.1-1b}$$

That is, almost like the Fourier transform but with '$\cos + \sin$' instead of '$\cos + i \cdot \sin$'. The (continuous version of the) Hartley transform is treated in [130].

The Hartley transform of a purely real sequence is purely real:

$$\mathcal{H}[a] \in \mathbb{R} \qquad \text{for} \quad a \in \mathbb{R} \tag{24.1-2}$$

It also is its own inverse:

$$\mathcal{H}[\mathcal{H}[a]] = a \tag{24.1-3}$$

Symmetry is conserved, like for the Fourier transform: the Hartley transform of a symmetric, antisymmetric sequence is symmetric, antisymmetric, respectively. Using the notation from section 20.7 on page 395 one has

$$\mathcal{H}[a_S] = +\overline{\mathcal{H}[a_S]} = +\mathcal{H}[\overline{a_S}] \tag{24.1-4a}$$

$$\mathcal{H}[a_A] = -\overline{\mathcal{H}[a_A]} = -\mathcal{H}[\overline{a_A}] \tag{24.1-4b}$$

An algorithm for the fast ($n \log(n)$-) computation of the Hartley transform is called a *fast Hartley transform* (FHT).

24.2 Radix-2 FHT algorithms

24.2.1 Decimation in time (DIT) FHT

Notation: For a length-n sequence a of let $\mathcal{X}^{1/2}a$ denote the sequence with elements $a_x \cos \pi\, x/n + \overline{a}_x \sin \pi\, x/n$. The operator $\mathcal{X}^{1/2}$ is the equivalent to the operator $\mathcal{S}^{1/2}$ of the Fourier transform algorithms. We use the notation $(even)$ and (odd) as introduced on page 378. The radix-2 decimation in time (DIT) step for the FHT:

$$\mathcal{H}\,[a]^{(left)} \quad \stackrel{n/2}{=} \quad \mathcal{H}\left[a^{(even)}\right] + \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{24.2-1a}$$

$$\mathcal{H}\,[a]^{(right)} \quad \stackrel{n/2}{=} \quad \mathcal{H}\left[a^{(even)}\right] - \mathcal{X}^{1/2}\mathcal{H}\left[a^{(odd)}\right] \tag{24.2-1b}$$

This is the equivalent to relations 20.3-3a and 20.3-3b on page 379.

Pseudo code for a recursive radix-2 DIT FHT (C++ version in [FXT: fht/recfht2.cc]):

```
procedure rec_fht_dit2(a[], n, x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2;

    for k:=0 to nh-1
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    rec_fht_dit2(s[], nh, b[])
    rec_fht_dit2(t[], nh, c[])

    hartley_shift(c[], nh, 1/2)

    for k:=0 to nh-1
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The result is returned in the array in `x[]`. The procedure `hartley_shift()` implements the operator $\mathcal{X}^{1/2}$, it replaces element c_k of the input sequence c by $c_k\, cos(\pi\, k/n) + c_{n-k}\, sin(\pi\, k/n)$. As pseudo code:

```
procedure hartley_shift_05(c[], n)
// real c[0..n-1] input, result
{
    nh := n/2
    j := n-1
    for k:=1 to nh-1
    {
        c := cos( PI*k/n )
        s := sin( PI*k/n )

        {c[k], c[j]} := {c[k]*c+c[j]*s, c[k]*s-c[j]*c}

        j := j-1
    }
}
```

C++ implementations are given in [FXT: fht/hartleyshift.h]. A version that exploits the symmetry of the trigonometric factors is

```
#define Tdouble long double
#define Sin     sinl
```

```
template <typename Type>
inline void hartley_shift_05_v2rec(Type *f, ulong n)
{
    const ulong nh = n/2;
    if ( n>=4 )
    {
        ulong i0=nh/2, j0=3*i0;
        Type fi = f[i0],  fj = f[j0];
        double cs = SQRT1_2;
        f[i0] = (fi + fj) * cs;
        f[j0] = (fi - fj) * cs;

        if ( n>=8 )
        {
            const Tdouble phi0 = PI/n;
            Tdouble be = Sin(phi0),  al = Sin(0.5*phi0);  al *= (2.0*al);
            Tdouble s = 0.0,  c = 1.0;
            for (ulong i=1, j=n-1, k=nh-1, l=nh+1;  i<k;  ++i, --j, --k, ++l)
            {
                { Tdouble tt = c;  c -= (al*tt+be*s);  s -= (al*s-be*tt); }

                fi = f[i];
                fj = f[j];
                f[i] = fi * (double)c + fj * (double)s;
                f[j] = fi * (double)s - fj * (double)c;

                fi = f[k];
                fj = f[l];
                f[k] = fi * (double)s + fj * (double)c;
                f[l] = fi * (double)c - fj * (double)s;
            }
        }
    }
}

#undef Tdouble
#undef Sin
```

Pseudo code for a non-recursive radix-2 DIT FHT:

```
procedure fht_depth_first_dit2(a[], ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2

    revbin_permute(a[], n)

    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4

        for r:=0 to n-m step m
        {
            for j:=1 to m4-1  // hartley_shift(a+r+mh,mh,1/2)
            {
                k := mh - j

                u := a[r+mh+j]
                v := a[r+mh+k]

                c := cos(j*PI/mh)
                s := sin(j*PI/mh)

                {u, v} := {u*c+v*s, u*s-v*c}

                a[r+mh+j] := u
                a[r+mh+k] := v
            }

            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

The derivation of the 'usual' DIT2 FHT algorithm starts by combining the Hartley-shift with the sum/diff-operations [FXT: fht/fhtdit2.cc]:

```
void fht_depth_first_dit2(double *f, ulong ldn)
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }

            for (ulong j=1, k=mh-1; j<k; ++j,--k)
            {
                double s, c;
                SinCos(phi0*j, &s, &c);

                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

Finally, as with the FFT equivalent (see page 380), the number of trigonometric computations can be reduced by swapping the innermost loops [FXT: fht/fhtdit2.cc]:

```
void fht_dit2(double *f, ulong ldn)
// Radix-2 decimation in time (DIT) FHT.
{
    const ulong n = 1UL<<ldn;

    revbin_permute(f, n);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;

        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
```

```
            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }

        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;
                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
        }
    }
}
```

24.2.2 Decimation in frequency (DIF) FHT

The radix-2 decimation in frequency step for the FHT is (compare to relations 20.3-6a and 20.3-6b on page 382):

$$\mathcal{H}\,[a]^{(even)} \quad \overset{n/2}{=} \quad \mathcal{H}\left[a^{(left)} + a^{(right)}\right] \tag{24.2-2a}$$

$$\mathcal{H}\,[a]^{(odd)} \quad \overset{n/2}{=} \quad \mathcal{H}\left[\mathcal{X}^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{24.2-2b}$$

Pseudo code for a recursive radix-2 DIF FHT (the C++ equivalent is given in [FXT: fht/recfht2.cc]):

```
procedure rec_fht_dif2(a[], n, x[])
// real a[0..n-1] input
// real x[0..n-1] result
{
    real b[0..n/2-1], c[0..n/2-1]   // workspace
    real s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then
    {
        x[0] := a[0]
        return
    }

    nh := n/2;

    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left'  elements
        t[k] := a[k+nh]  // 'right' elements
    }

    for k:=0 to nh-1
    {
        {s[k], t[k]} := {s[k]+t[k], s[k]-t[k]}
    }

    hartley_shift(t[], nh, 1/2)

    rec_fht_dif2(s[], nh, b[])
    rec_fht_dif2(t[], nh, c[])
```

```
    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

Pseudo code for a non-recursive radix-2 DIF FHT (C++ version in [FXT: fht/fhtdif2.cc]):

```
procedure fht_depth_first_dif2(a[], ldn)
// real a[0..n-1] input,result
{
    n := 2**ldn  // length of a[] is a power of 2
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        m4 := m/4
        for r:=0 to n-m step m
        {
            for j:=0 to mh-1
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
            for j:=1 to m4-1
            {
                k := mh - j
                u := a[r+mh+j]
                v := a[r+mh+k]
                c := cos(j*PI/mh)
                s := sin(j*PI/mh)
                {u, v} := {u*c+v*s, u*s-v*c}
                a[r+mh+j] := u
                a[r+mh+k] := v
            }
        }
    }
    revbin_permute(a[], n)
}
```

The 'usual' DIF2 FHT algorithm then is again obtained by swapping the inner loops, a C++ implementation is [FXT: `fht_dif2()` in fht/fhtdif2.cc]:

```
void fht_dif2(double *f, ulong ldn)
// Radix-2 decimation in frequency (DIF) FHT
{
    const ulong n = (1UL<<ldn);
    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        const ulong m4 = (mh>>1);
        const double phi0 = M_PI/mh;
        for (ulong r=0; r<n; r+=m)
        {
            {  // j == 0:
                ulong t1 = r;
                ulong t2 = t1 + mh;
                sumdiff(f[t1], f[t2]);
            }
            if ( m4 )
            {
                ulong t1 = r + m4;
                ulong t2 = t1 + mh;
```

```
                sumdiff(f[t1], f[t2]);
            }
        }

        for (ulong j=1, k=mh-1; j<k; ++j,--k)
        {
            double s, c;
            SinCos(phi0*j, &s, &c);

            for (ulong r=0; r<n; r+=m)
            {
                ulong tj = r + mh + j;
                ulong tk = r + mh + k;

                ulong t1 = r + j;
                ulong t2 = tj; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                t1 = r + k;
                t2 = tk; // == t1 + mh;
                sumdiff(f[t1], f[t2]);

                double fj = f[tj];
                double fk = f[tk];
                f[tj] = fj * c + fk * s;
                f[tk] = fj * s - fk * c;
            }
        }
    }

    revbin_permute(f, n);
}
```

24.3 Complex FT by HT

The relations between the HT and the FT can be read off directly from their definitions and their symmetry relations. Let σ be the sign of the exponent in the FT, then the HT of a complex sequence $d \in \mathbb{C}$ is

$$\mathcal{F}\big[\,d\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[d] + \overline{\mathcal{H}\,[d]} + \sigma\, i\,\left(\mathcal{H}\,[d] - \overline{\mathcal{H}\,[d]}\right)\right) \tag{24.3-1}$$

Written out for the real and imaginary part of $d = a + i\,b$ $(a, b \in \mathbb{R})$:

$$\mathfrak{Re}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[a] + \overline{\mathcal{H}\,[a]} - \sigma\,\left(\mathcal{H}\,[b] - \overline{\mathcal{H}\,[b]}\right)\right) \tag{24.3-2a}$$

$$\mathfrak{Im}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\left(\mathcal{H}\,[b] + \overline{\mathcal{H}\,[b]} + \sigma\,\left(\mathcal{H}\,[a] - \overline{\mathcal{H}\,[a]}\right)\right) \tag{24.3-2b}$$

Using the symmetry relations 24.1-4a and 24.1-4b on page 483 one can recast the relations as

$$\mathfrak{Re}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\,\mathcal{H}\,[a_S - \sigma\,b_A] \tag{24.3-3a}$$

$$\mathfrak{Im}\,\mathcal{F}\big[\,a + i\,b\,\big] \;=\; \frac{1}{2}\,\mathcal{H}\,[b_S + \sigma\,a_A] \tag{24.3-3b}$$

Both formulations lead to the very same conversion procedure. The following pseudo code is for a complex FT by HT conversion:

```
fht_fft_conversion(a[], b[], n, is)
// preprocessing to use two length-n FHTs
// to compute a length-n complex FFT
// or
// postprocessing to use two length-n FHTs
// to compute a length-n complex FFT
//
// Self-inverse.
{
    for k:=1 to n/2-1
```

```
    {
        t := n-k

        as := a[k] + a[t]
        aa := a[k] - a[t]

        bs := b[k] + b[t]
        ba := b[k] - b[t]

        aa := is * aa
        ba := is * ba

        a[k] := 1/2 * (as - ba)
        a[t] := 1/2 * (as + ba)

        b[k] := 1/2 * (bs + aa)
        b[t] := 1/2 * (bs - aa)
    }
}
```

The C++ implementations are given in [FXT: fft/fhtfft.cc] for type `double` and [FXT: fft/fhtcfft.cc] for type `complex`. Now we have two options to compute a complex FT by two HTs. Version 1 does the FHTs first:

```
fft_by_fht1(a[], b[], n, is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    fht(a[], n)
    fht(b[], n)
    fht_fft_conversion(a[], b[], n, is)
}
```

Version 2 does the FHTs at the end of the routine:

```
fft_by_fht2(a[], b[], n, is)
// real a[0..n-1] input,result (real part)
// real b[0..n-1] input,result (imaginary part)
{
    fht_fft_conversion(a[], b[], n, is)
    fht(a[], n)
    fht(b[], n)
}
```

Note that the real and imaginary parts of the FT are computed independently by this procedure. This can be very advantageous when the real and imaginary part of complex data lies in separate arrays. The C++ version is given in [FXT: fft/fhtfft.cc].

24.4 Complex FT by complex HT and vice versa

A complex valued HT is simply two HTs (one of the real, one of the imaginary part). So we can use either version from section 24.3 and there is nothing new. Really? If one has a type `complex` version of both the conversion and the FHT routine then the complex FFT can be computed as either

```
fft_by_fht1(c[], n, is)
// complex c[0..n-1] input,result
{
    fht(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

or the same with swapped statements.

This may not make you scream but here is the message: it makes sense to do so. One saves half of the trigonometric computations and book keeping. It is pretty easy to derive a complex FHT from the real version and with a well optimized FHT you get an even better optimized FFT. C++ implementations of complex FHTs are given in [FXT: fht/cfhtdif.cc] (DIF algorithm), [FXT: fht/cfhtdit.cc] (DIT algorithm), and, for zero padded data, [FXT: fht/cfht0.cc].

The other way round: computation of a complex FHT using FFTs. Let T be the operator corresponding

to the `fht_fft_conversion`. The operator is its own inverse: $T = T^{-1}$. We have seen that

$$\mathcal{F} = \mathcal{H} \cdot T \qquad \text{and} \qquad \mathcal{F} = T \cdot \mathcal{H} \tag{24.4-1}$$

Thereby (multiply the relations with T and use $T \cdot T = 1$):

$$\mathcal{H} = T \cdot \mathcal{F} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F} \cdot T \tag{24.4-2}$$

Hence we have either

```
fht_by_fft(c[], n, is)
// complex c[0..n-1] input,result
{
    fft(c[], n)
    fht_fft_conversion(c[], n, is)
}
```

or the same thing with swapped lines [FXT: fft/fhtcfft.cc]. The same ideas also work for separate real and imaginary parts but in that case one should rather use separate FHTs for the two arrays.

24.5 Real FT by HT and vice versa

To express the real and imaginary part of a Fourier transform of a purely real sequence $a \in \mathbb{R}$ by its Hartley transform use relations 24.3-2a and 24.3-2b on page 489 and set $b = 0$:

$$\mathfrak{Re}\, \mathcal{F}\big[\, a \,\big] = \frac{1}{2}\left(\mathcal{H}\,[a] + \overline{\mathcal{H}\,[a]}\right) \tag{24.5-1a}$$

$$\mathfrak{Im}\, \mathcal{F}\big[\, a \,\big] = \sigma\,\frac{1}{2}\left(\mathcal{H}\,[a] - \overline{\mathcal{H}\,[a]}\right) \tag{24.5-1b}$$

A C++ implementation is [FXT: `fht_real_complex_fft()` in realfft/realfftbyfht.cc]:

```
template <typename Type>
static inline void sumdiff05(Type &a, Type &b)
// {a, b}  <--| {0.5*(a+b),  0.5*(a-b)}
{ Type t=(a-b)*0.5; a+=b; a*=0.5; b=t; }

template <typename Type>
static inline void sumdiff05_r(Type &a, Type &b)
// {a, b}  <--| {0.5*(a+b), 0.5*(b-a)}
{ Type t=(b-a)*0.5; a+=b; a*=0.5; b=t; }

void
fht_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
{
    fht(f, ldn);

    const ulong n = (1UL<<ldn);

    if ( is>0 )  for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff05(f[i], f[j]);
    else         for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff05_r(f[i], f[j]);
}
```

At the end of the procedure the ordering of the output data $c = \mathcal{F}\big[\, a \,\big] \in \mathbb{C}$ is

$$\begin{aligned}
\texttt{a[0]} &= \mathfrak{Re}\, c_0 \\
\texttt{a[1]} &= \mathfrak{Re}\, c_1 \\
\texttt{a[2]} &= \mathfrak{Re}\, c_2 \\
&\dots \\
\texttt{a[n/2]} &= \mathfrak{Re}\, c_{n/2} \\
\texttt{a[n/2+1]} &= \mathfrak{Im}\, c_{n/2-1} \\
\texttt{a[n/2+2]} &= \mathfrak{Im}\, c_{n/2-2} \\
\texttt{a[n/2+3]} &= \mathfrak{Im}\, c_{n/2-3} \\
&\dots \\
\texttt{a[n-1]} &= \mathfrak{Im}\, c_1
\end{aligned} \tag{24.5-2}$$

The inverse procedure is given in [FXT: realfft/realfftbyfht.cc]:

```
void
fht_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
{
    const ulong n = (1UL<<ldn);
    if ( is>0 )  for (ulong i=1,j=n-1; i<j; i++,j--)  sumdiff(f[i], f[j]);
    else         for (ulong i=1,j=n-1; i<j; i++,j--)  diffsum(f[i], f[j]);
    fht(f,ldn);
}
```

The function `sumdiff()` is defined in [FXT: aux0/sumdiff.h]:

```
template <typename Type>
static inline void sumdiff(Type &a, Type &b)
// {a, b}  <--| {a+b, a-b}
{ Type t=a-b; a+=b; b=t; }
template <typename Type>
static inline void diffsum(Type &a, Type &b)
// {a, b}  <--| {a-b, a+b}
{ Type t=a-b; b+=a; a=t; }
```

The input has to be ordered as given above (relations 24.5-2). The sign of the transform (`is`) has to be the same as with the forward version.

Computation of a (real-valued) FHT using a real-valued FFT proceeds similar as for complex versions. Let T_{r2c} be the operator corresponding to the post-processing in `real_complex_fft_by_fht()`, and T_{c2r} correspond to the preprocessing in `complex_real_fft_by_fht()`. That is

$$\mathcal{F}_{c2r} = \mathcal{H} \cdot T_{c2r} \qquad \text{and} \qquad \mathcal{F}_{r2c} = T_{r2c} \cdot \mathcal{H} \tag{24.5-3}$$

The operators are mutually inverse: $T_{r2c} = T_{c2r}^{-1}$ and $T_{c2r} = T_{r2c}^{-1}$. Multiplying the relations and using $T_{r2c} \cdot T_{c2r} = T_{c2r} \cdot T_{r2c} = 1$ gives

$$\mathcal{H} = T_{c2r} \cdot \mathcal{F}_{r2c} \qquad \text{and} \qquad \mathcal{H} = \mathcal{F}_{c2r} \cdot T_{r2c} \tag{24.5-4}$$

The corresponding code should be obvious. Watch out for real-to-complex FFTs that use a different ordering of the output than given in relation 24.5-2.

24.6 Higher radix FHT algorithms

Higher radix FHT algorithms seem to get complicated due to the structure of the Hartley shift operator. In fact there is a straightforward way to turn any FFT decomposition into an FHT algorithm.

For the moment assume that we want to compute a complex HT, further assume we want to use a radix-r algorithm. At each step we have r short HTs and want to combine them to a longer HT but we do not know how this might be done. In section 24.3 on page 489 we learned how to turn a HT into an FT using the T-operator. And we have seen radix-r algorithms for the FFT. The crucial idea is to use the conversion operator T as a wrapper around the FFT-step that combines several short FTs into a longer one. Here is how to turn a radix-r FFT-step into an FHT-step, simply do the following:

1. first convert the r short HTs into FTs (use T on the subsequences)
2. then perform the radix-r the FFT step
3. finally convert the FT into a HT (use T on the sequence)

For efficient implementations one obviously wants to combine the computations.

To obtain real-valued FHTs note that the real and imaginary parts do not 'mix': one can use the identical algorithm with real input (and the corresponding data types). With a radix-r step the scheme always accesses $2r$ elements simultaneously. The symmetry of the trigonometric factors is thereby automatically exploited. Splitting steps for the radix-4 FHT and the split-radix FHT are given in [218].

24.7 Convolution via FHT

The convolution property of the Hartley transform can be stated as

$$\mathcal{H}[a \circledast b] = \frac{1}{2}\Big(\mathcal{H}[a]\,\mathcal{H}[b] - \overline{\mathcal{H}[a]}\,\overline{\mathcal{H}[b]} + \mathcal{H}[a]\,\overline{\mathcal{H}[b]} + \overline{\mathcal{H}[a]}\,\mathcal{H}[b]\Big) \tag{24.7-1}$$

or, with $c := \mathcal{H}[a]$ and $d := \mathcal{H}[b]$, written element-wise:

$$\mathcal{H}[a \circledast b]_k = \frac{1}{2}\Big(c_k\,d_k - \overline{c_k}\,\overline{d_k} + c_k\,\overline{d_k} + \overline{c_k}\,d_k\Big) \tag{24.7-2a}$$

$$= \frac{1}{2}\Big(c_k\,(d_k + \overline{d_k}) + \overline{c_k}\,(d_k - \overline{d_k})\Big) \tag{24.7-2b}$$

The latter form reduces the number of multiplications. When turning the relation into an algorithm one has to keep in mind that both elements $y_k = \mathcal{H}[a \circledast b]$ and y_{-k} must be computed simultaneously.

24.7.1 Implementation as pseudo code

Pseudo code for the cyclic convolution of two real valued sequences `x[]` and `y[]` via the FHT. `n` must be even, the result is returned in `y[]`:

```
procedure fht_cyclic_convolution(x[], y[], n)
// real x[0..n-1] input, modified
// real y[0..n-1] result
{
    // transform data:
    fht(x[], n)
    fht(y[], n)

    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        xi := x[i]
        xj := x[j]

        yp := y[i] + y[j]   // ==   y[j] + y[i]
        ym := y[i] - y[j]   // == -(y[j] - y[i])

        y[i] := (xi*yp + xj*ym)/2
        y[j] := (xj*yp - xi*ym)/2

        j := j-1
```

```
    }
    y[0] := x[0]*y[0]
    if n>1 then  y[n/2] := x[n/2]*y[n/2]

    // transform back:
    fht(y[], n)

    // normalize:
    for i:=0 to n-1
    {
        y[i] := y[i] / n
    }
}
```

It is assumed that the procedure `fht()` does no normalization. The C++ equivalent is given in [FXT: convolution/fhtcnvl.cc].

Equation 24.7-2a on the previous page (slightly optimized) for the auto convolution is

$$\mathcal{H}\left[a \circledast a\right]_k = \frac{1}{2}\Big(c_k\left(c_k + \overline{c_k}\right) + \overline{c_k}\left(c_k - \overline{c_k}\right)\Big) \tag{24.7-3a}$$

$$= c_k\,\overline{c_k} + \frac{1}{2}\left(c_k^2 - \overline{c_k}^2\right) \tag{24.7-3b}$$

where $c = \mathcal{H}\left[a\right]$.

We give pseudo code for the cyclic auto convolution that uses a fast Hartley transform, `n` must be even:

```
procedure cyclic_self_convolution(x[], n)
// real x[0..n-1] input, result
{
    // transform data:
    fht(x[], n)

    // convolution in transformed domain:
    j := n-1
    for i:=1 to n/2-1
    {
        ci := x[i]
        cj := x[j]

        t1 := ci*cj               // == cj*ci
        t2 := 1/2*(ci*ci-cj*cj)   // == -1/2*(cj*cj-ci*ci)

        x[i] := t1 + t2
        x[j] := t1 - t2

        j := j-1
    }
    x[0]   := x[0]*x[0]

    if n>1 then  x[n/2] := x[n/2]*x[n/2]

    // transform back:
    fht(x[], n)

    // normalize:
    for i:=0 to n-1
    {
        x[i] := x[i] / n
    }
}
```

For odd `n` replace the line

```
    for i:=1 to n/2-1
```

by

```
    for i:=1 to (n-1)/2
```

and omit the line

```
    if n>1 then  x[n/2] := x[n/2]*x[n/2]
```

in both procedures above.

24.7.2 C++ implementations

The FHT based routine for the cyclic convolution of two real sequences is:

```
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht(f, ldn);
    fht(g, ldn);
    fht_convolution_core(f, g, ldn);
    fht(g, ldn);
}
```

The equivalent to the element-wise multiplication is given in [FXT: convolution/fhtcnvlcore.cc]:

```
void
fht_convolution_core(const double * restrict f, double * restrict g, ulong ldn,
                     double v/*=0.0*/)
// Auxiliary routine for the computation of convolutions
//   via Fast Hartley Transforms.
// ldn := base-2 logarithm of the array length.
// v!=0.0 chooses alternative normalization.
{
    const ulong n = (1UL<<ldn);

    if ( v==0.0 )  v = 1.0/n;

    g[0]  *=  (v * f[0]);
    const ulong  nh = n/2;

    if ( nh>0 )
    {
        g[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<j; i++,j--)  fht_mul(f[i], f[j], g[i], g[j], v);
    }
}
```

where [FXT: convolution/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_mul(Type xi, Type xj, Type &yi, Type &yj, double v)
// yi <-- v*( 2*xi*xj + xi*xi - xj*xj )
// yj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type h1p = xi,  h1m = xj;
    Type s1 = h1p + h1m,  d1 = h1p - h1m;
    Type h2p = yi,  h2m = yj;
    yi = (h2p * s1 + h2m * d1) * v;
    yj = (h2m * s1 - h2p * d1) * v;
}
```

A C++ implementation of the FHT based self-convolution is given in [FXT: convolution/fhtcnvla.cc]. It uses the routine

```
void
fht_auto_convolution_core(double *f, ulong ldn,
                          double v/*=0.0*/)
// v!=0.0 chooses alternative normalization
{
    const ulong  n  = (1UL<<ldn);
    if ( v==0.0 )  v = 1.0/n;
    f[0] *= (v * f[0]);
    if ( n>=2 )
    {
        const ulong  nh = n/2;
        f[nh] *= (v * f[nh]);
        v *= 0.5;
        for (ulong i=1,j=n-1; i<nh; i++,j--)  fht_sqr(f[i], f[j], v);
    }
}
```

where [FXT: convolution/fhtmulsqr.h]:

```
template <typename Type>
static inline  void
fht_sqr(Type &xi, Type &xj, double v)
// xi <-- v*( 2*xi*xj + xi*xi - xj*xj )
```

```
// xj <-- v*( 2*xi*xj - xi*xi + xj*xj )
{
    Type a = xi,  b = xj;
    Type s1 = (a + b) * (a - b);
    a *= b;
    a += a;
    xi = (a+s1) * v;
    xj = (a-s1) * v;
}
```

24.7.3 Avoiding the revbin permutations

The observation that the revbin permutations can be omitted with FFT based convolutions (see section 21.1.3 on page 411) applies again [FXT: convolution/fhtcnvlcore.cc]:

```
void
fht_convolution_revbin_permuted_core(const double * restrict f,
                                     double * restrict g,
                                     ulong ldn,
                                     double v/*=0.0*/)
// Same as fht_convolution_core() but with data access in revbin order.
{
    const ulong n = (1UL<<ldn);

    if ( v==0.0 )  v = 1.0/n;

    g[0] *= (v * f[0]);  // 0 == revbin(0)
    if ( n>=2 )  g[1] *= (v * f[1]); // 1 == revbin(nh)

    if ( n<4 )  return;

    v *= 0.5;
    const ulong nh = (n>>1);

    ulong r=nh, rm=n-1; // nh == revbin(1),  n1-1 == revbin(n-1)
    fht_mul(f[r], f[rm], g[r], g[rm], v);

    ulong k=2, km=n-2;
    while ( k<nh  )
    {
        // k even:
        rm -= nh;
        ulong tr = r;
        r^=nh;  for (ulong m=(nh>>1); !((r^=m)&m); m>>=1)  {;}
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;

        // k odd:
        rm += (tr-r);
        r += nh;
        fht_mul(f[r], f[rm], g[r], g[rm], v);
        --km;
        ++k;
    }
}
```

The optimized version saving three revbin permutations is [FXT: convolution/fhtcnvl.cc]:

```
void fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht_dif_core(f, ldn);
    fht_dif_core(g, ldn);
    fht_convolution_revbin_permuted_core(f, g, ldn);
    fht_dit_core(g, ldn);
}
```

24.8 Negacyclic convolution via FHT

Pseudo code for the computation of the negacyclic (auto-) convolution via FHT:

```
procedure negacyclic_self_convolution(x[], n)
// real x[0..n-1]  input, result
```

```
{
    hartley_shift_05(x, n)    // preproces
    fht(x, n)    // transform data

    // convolution in transformed domain:
    j := n-1
    for i:=0 to n/2-1  // here i starts from zero
    {
        a := x[i]
        b := x[j]

        x[i] := a*b+(a*a-b*b)/2
        x[j] := a*b-(a*a-b*b)/2
        j := j-1
    }

    fht(x, n)    // transform back
    hartley_shift_05(x, n)    // postprocess
}
```

C++ implementations for the negacyclic convolution and self convolution are given in [FXT: convolution/fhtnegacnvl.cc]. The FHT-based negacyclic convolution is turns out to be extremely useful for the computation of weighted transforms, for example in the MFA-based convolution for real input, see section 21.4.1 on page 421.

24.9 Localized FHT algorithms

Localized routines for the FHT can be obtained by slight modifications of the corresponding algorithms for the Walsh transform described in section 22.6 on page 440. The decimation in time (DIT) version is [FXT: fht/fhtloc2.h]:

```
template <typename Type>
void fht_loc_dit2_core(Type *f, ulong ldn)
{
    if ( ldn<=13 )  // sizeof(Type)*(2**thres) <= L1_CACHE_BYTES
    {
        fht_dit_core(f, ldn);
        return;
    }

    // Recursion:
    fht_dit_core_2(f+2);  // ldm==1
    fht_dit_core_4(f+4);  // ldm==2
    fht_dit_core_8(f+8);  // ldm==3
    for (ulong ldm=4; ldm<ldn; ++ldm)  fht_loc_dit2_core(f+(1UL<<ldm), ldm);

    for (ulong ldm=1; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        hartley_shift_05(f+mh, mh);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
    }
}
```

The routine `hartley_shift_05()` is described in 24.2.1 on page 484. One should choose a implementation that uses trigonometric recursion as this improves performance considerably.

The decimation in frequency (DIF) version is:

```
template <typename Type>
void fht_loc_dif2_core(Type *f, ulong ldn)
{
    if ( ldn<=13 )  // sizeof(Type)*(2**thres) <= L1_CACHE_BYTES
    {
        fht_dif_core(f, ldn);
        return;
    }

    for (ulong ldm=ldn; ldm>=1; --ldm)
    {
```

```
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);
        for (ulong t1=0, t2=mh;  t1<mh;  ++t1, ++t2)  sumdiff(f[t1], f[t2]);
        hartley_shift_05(f+mh, mh);
    }
    // Recursion:
    fht_dif_core_2(f+2);  // ldm==1
    fht_dif_core_4(f+4);  // ldm==2
    fht_dif_core_8(f+8);  // ldm==3
    for (ulong ldm=4; ldm<ldn; ++ldm)  fht_loc_dif2_core(f+(1UL<<ldm), ldm);
}
```

The (generated) short-length transforms are given in the files [FXT: fht/shortfhtdifcore.h] and [FXT: fht/shortfhtditcore.h]. For example, the length-8 decimation in frequency routine is

```
template <typename Type>
inline void
fht_dif_core_8(Type *f)
{
   Type g0, f0, f1, g1;
   sumdiff(f[0], f[4], f0, g0);
   sumdiff(f[2], f[6], f1, g1);
   sumdiff(f0, f1);
   sumdiff(g0, g1);
   Type s1, c1, s2, c2;
   sumdiff(f[1], f[5], s1, c1);
   sumdiff(f[3], f[7], s2, c2);
   sumdiff(s1, s2);
   sumdiff(f0, s1, f[0], f[1]);
   sumdiff(f1, s2, f[2], f[3]);
   c1 *= SQRT2;
   c2 *= SQRT2;
   sumdiff(g0, c1, f[4], f[5]);
   sumdiff(g1, c2, f[6], f[7]);
}
```

An additional revbin permutation is needed if the data is required in order. The FHT can be computed by either

```
    fht_loc_dif2_core(f, ldn);
    revbin_permute(f, 1UL<<ldn);
```

or

```
    revbin_permute(f, 1UL<<ldn);
    fht_loc_dit2_core(f, ldn);
```

Performance for large arrays is excellent: the convolutions based on the transforms [FXT: convolution/fhtloccnvl.cc]

```
void
loc_fht_convolution(double * restrict f, double * restrict g, ulong ldn)
{
    fht_loc_dif2_core(f, ldn);
    fht_loc_dif2_core(g, ldn);
    fht_convolution_revbin_permuted_core(f, g, ldn);
    fht_loc_dit2_core(g, ldn);
}
```

and [FXT: convolution/fhtloccnvla.cc]

```
void
loc_fht_auto_convolution(double *f, ulong ldn)
{
    fht_loc_dif2_core(f, ldn);
    fht_auto_convolution_revbin_permuted_core(f, ldn);
    fht_loc_dit2_core(f, ldn);
}
```

gave a significant (more than 50 percent) speedup for the high precision multiplication routines (see section 27.3 on page 532) used in the hfloat library [20].

24.10 Two-dimensional FHTs

A two-dimensional FHT can be computed almost as easy as a two-dimensional FFT, only a trivial additional step is needed. Start with the row-column algorithm described in section 20.10.2 on page 405 [FXT: fht/twodimfht.cc]:

```
void
row_column_fht(double *f, ulong nr, ulong nc)
// FHT over rows and columns.
// nr := number of rows
// nc := number of columns
{
    ulong n = nr * nc;

    // fht over rows:
    ulong ldc = ld(nc);
    for (ulong k=0; k<n; k+=nc)  FHT(f+k, ldc);

    // fht over columns:
    double *w = new double[nr];
    for (ulong k=0; k<nc; k++)  skip_fht(f+k, nr, nc, w);
    delete [] w;
}
```

Note that no attempt has been made to make the routine cache friendly: the routine `skip_fht()` [FXT: fht/skipfht.cc] simply copies a column into the scratch array, does the FHT and copies the data back. This is not yet a two-dimensional FHT, the following post-processing must be made:

```
void
y_transform(double *f, ulong nr, ulong nc)
// Transforms row-column-FHT to 2-dimensional FHT.
// Self-inverse.
// nr := number of rows
// nc := number of columns
{
    ulong rh = nr/2;
    if ( nr&1 )  rh++;

    ulong ch = nc/2;
    if ( nc&1 )  ch++;

    ulong n = nr*nc;
    for (ulong tr=1, ctr=nc; tr<rh; tr++,ctr+=nc) // ctr=nc*tr
    {
        double *pa = f + ctr;
        double *pb = pa + nc;
        double *pc = f + n - ctr;
        double *pd = pc + nc;
        for (ulong tc=1; tc<ch; tc++)
        {
            pa++;
            pb--;
            pc++;
            pd--;
            double e = (*pa + *pd - *pb - *pc) * 0.5;
            *pa -= e;
            *pb += e;
            *pc += e;
            *pd -= e;
        }
    }
}
```

The canned routine is therefore

```
void
twodim_fht(double *f, ulong nr, ulong nc)
// Two dimensional fast Hartley transform (FHT)
// nr := number of rows
// nc := number of columns
{
    row_column_fht(f, nr, nc);
    y_transform(f, nr, nc);
}
```

24.11 Discrete cosine transform (DCT) by HT

The discrete cosine transform (DCT) with respect to the basis

$$u(k) = \nu(k) \cdot \cos\left(\frac{\pi\, k\, (i+1/2)}{n}\right) \quad \text{where} \tag{24.11-1}$$

$$\nu(k) = \begin{cases} 1 & \text{if} \quad k = 0 \\ \sqrt{2} & \text{else} \end{cases}$$

can be computed from the FHT using an auxiliary routine which is its own inverse. As pseudo code:

```
procedure cos_rot(x[], y[], n)
// Real x[0..n-1] input
// Real y[0..n-1] result
{
    nh := n/2
    y[0]  := x[0]
    y[nh] := x[nh]
    phi := PI/2/n
    for k:=1 to nh-1
    {
        c := cos(phi*k)
        s := sin(phi*k)

        cps := (c+s)*sqrt(1/2)
        cms := (c-s)*sqrt(1/2)

        y[k]   := cms*x[k] + cps*x[n-k]
        y[n-k] := cps*x[k] - cms*x[n-k]
    }
}
```

The C++ equivalent is [FXT: `cos_rot()` in dctdst/cosrot.cc].

Pseudo code for the computation of the DCT via FHT:

```
procedure dcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    real y[0..n-1]  // workspace
    unzip_rev(x, y, n)
    fht(y[],ldn)
    cos_rot(y[], x[], n)
}
```

where `unzip_rev()` is the reversed unzip permutation (see section 2.6 on page 95):

```
procedure unzip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

Pseudo code for the computation of the inverse discrete cosine transform via FHT:

```
procedure idcth(x[], ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    real y[0..n-1]  // workspace
    cos_rot(x[], y[], n);
    fht(y[],ldn)
    zip_rev(y[], x[], n)
}
```

where the routine `zip_rev()` is the reversed zip permutation:

```
procedure zip_rev(a[], b[], n)
// real a[0..n-1] input
// real b[0..n-1] result
{
    nh := n/2
    for k:=0 to nh-1
    {
        k2 := 2*k
        b[k]    := a[k2]
        b[nh+k] := a[n-1-k2]
    }
}
```

The C++ implementations of both the forward and the backward transform [FXT: dctdst/dcth.cc] avoid the temporary array if no scratch space is supplied. The algorithms are given in [177] and [178]. An alternative variant for the computation of the DCT that also uses the FHT is given in [FXT: dctdst/dctzapata.cc], the algorithm is described in [16].

24.12 Discrete sine transform (DST) by DCT

The basis of the *discrete sine transform* (DST) is

$$u(k) \quad = \quad \sin\left(\frac{\pi\,(k+1)\,(i+1/2)}{n}\right) \tag{24.12-1}$$

Pseudo code for the computation of the DST via the discrete cosine transform (DCT):

```
procedure dst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    nh := n/2
    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }
    dct(x, ldn)
    for k:=0 to nh-1
    {
        swap(x[k], x[n-1-k])
    }
}
```

The corresponding C++ implementation is [FXT: `dsth()` in dctdst/dsth.cc]. Pseudo code for the computation of the inverse sine transform using the inverse cosine transform:

```
procedure idst(x[],ldn)
// real x[0..n-1] input,result
{
    n := 2**ldn
    nh := n/2
    for k:=0 to nh-1
    {
        swap(x[k], x[n-1-k])
    }
    idct(x, ldn)
    for k:=1 to n-1 step 2
    {
        x[k] := -x[k]
    }
}
```

The C++ version is [FXT: `idsth()` in dctdst/dsth.cc].

24.13 Automatic generation of transform code

FFT *generators* are programs that output FFT routines, usually for fixed (short) lengths. In fact the thoughts here are not at all restricted to FFT codes. However, fast transforms and routines that can be unrolled like those for matrix multiplication or convolution are prime candidates for automated generation.

One can write code generators that have a built-in algorithmic knowledge. We restrict our attention to a simpler method known as *partial evaluation.* Writing such a program is easy: take an existing FFT and change all computations into print statements that emit the necessary code. The process, however, is less than delightful and error-prone.

It would be much better to have another program that takes the existing FFT code as input and emit the code for the generator. Let us call this a *meta-generator*. Implementing such a meta-generator of course is highly nontrivial. It actually is equivalent to writing an interpreter for the language used plus the necessary data flow analysis. A practical compromise is to write a program that, while theoretically not even close to a meta-generator, creates output that, after a little hand editing, is a usable generator code.

One may further want to print the current values of the loop variables of the original code as comments at the beginning of a block. Thereby it is possible to locate the corresponding part (with respect to both file and temporal location) of a piece of generated code in the original file. In addition one may keep the comments of the original code.

With FFTs it is necessary to identify ('reverse engineer') the trigonometric values that occur in the process in terms of the corresponding argument (rational multiples of π). The actual values should be inlined to some greater precision than actually needed, thereby one avoids the generation of multiple copies of the (logically) same value with differences only due to numeric inaccuracies. Printing the arguments, both as they appear and in lowest terms, inside comments helps to understand (or further optimize) the generated code:

```
double c1=.98078528040323044912618223613 4; // == cos(Pi*1/16) == cos(Pi*1/16)
double s1=.19509032201612826784828486847 6; // == sin(Pi*1/16) == sin(Pi*1/16)
double c2=.92387953251128675612818318939 7; // == cos(Pi*2/16) == cos(Pi*1/8)
double s2=.38268343236508977172845998402 9; // == sin(Pi*2/16) == sin(Pi*1/8)
```

Automatic verification of the generated codes against the original is a mandatory part of the process.

A level of abstraction for the array indices is of great use: when the print statements in the generator emit some function of the index instead of its plain value it is easy to generate modified versions of the code for permuted input. That is, instead of

```
cout << "sumdiff(f0, f2, g[" << k0 << "], g[" << k2 << "]);"  << endl;
cout << "sumdiff(f1, f3, g[" << k1 << "], g[" << k3 << "]);"  << endl;
```

use

```
cout << "sumdiff(f0, f2, " << idxf(g,k0) << ", " << idxf(g,k2) << ");"  << endl;
cout << "sumdiff(f1, f3, " << idxf(g,k1) << ", " << idxf(g,k3) << ");"  << endl;
```

where `idxf(g, k)` can be defined to print a modified (for example, revbin-permuted) index `k`.

A generated length-8 DIT FHT core (from [FXT: fht/shortfhtditcore.h]) shall serve as an example:

```
template <typename Type>
inline void fht_dit_core_8(Type *f)
// unrolled version for length 8
{
 { // start initial loop
  { // fi = 0  gi = 1
   Type g0, f0, f1, g1;
   sumdiff(f[0], f[1], f0, g0);
   sumdiff(f[2], f[3], f1, g1);
   sumdiff(f0, f1);
   sumdiff(g0, g1);
   Type s1, c1, s2, c2;
   sumdiff(f[4], f[5], s1, c1);
   sumdiff(f[6], f[7], s2, c2);
```

```
   sumdiff(s1, s2);
   sumdiff(f0, s1, f[0], f[4]);
   sumdiff(f1, s2, f[2], f[6]);
   c1 *= SQRT2;
   c2 *= SQRT2;
   sumdiff(g0, c1, f[1], f[5]);
   sumdiff(g1, c2, f[3], f[7]);
  }
 } // end initial loop
}
// opcount by generator:  #mult=2=0.25/pt   #add=22=2.75/pt
```

Generated DIF FHT codes for lengths up to 64 are given in [FXT: fht/shortfhtdifcore.h].

The generated codes can be useful when one wants to spot parts of the original code that need further optimization. Especially repeated trigonometric values and unused symmetries tend to be apparent in the unrolled code.

It is a good idea to let the generator count the number of operations (multiplications, additions, loads and stores) of the code it emits. It is even better if those numbers are compared to the corresponding values found in the compiled assembler code.

Checking the generated machine code

It is possible to have GCC produce the assembler code with the original source interlaced. This is a great tool for code optimization. The necessary commands are (include- and warning flags omitted)

```
# create assembler code:
c++ -S -fverbose-asm -g -O2  test.cc -o test.s
# create asm interlaced with source lines:
as -alhnd test.s  > test.lst
```

For example, the generated length-4 DIT FHT core from [FXT: fht/shortfhtditcore.h] is

```
template <typename Type>
inline void fht_dit_core_4(Type *f)
// unrolled version for length 4
{
   Type f0, f1, f2, f3;
   sumdiff(f[0], f[1], f0, f1);
   sumdiff(f[2], f[3], f2, f3);
   sumdiff(f0, f2, f[0], f[2]);
   sumdiff(f1, f3, f[1], f[3]);
}
```

With `Type` set to `double` the generated assembler is (some editing for readability)

```
  16:test.cc **** void fht_dit_core_4(double *f)
  17:test.cc **** {
  18:test.cc ****     double f0, f1, f2, f3;
  19:test.cc ****     sumdiff(f[0], f[1], f0, f1);
  49 0000 660F120F            movlpd  (%rdi), %xmm1   #* f, tmp63
  50 0004 660F1247            movlpd  8(%rdi), %xmm0  #, tmp64
  50      08
  20:test.cc ****     sumdiff(f[2], f[3], f2, f3);
  52 0009 660F1257            movlpd  16(%rdi), %xmm2 #, tmp67
  52      10
  54 000e F20F10D9            movsd   %xmm1, %xmm3    # tmp63, f0
  56 0012 F20F5CC8            subsd   %xmm0, %xmm1    # tmp64, f1
  59 0016 F20F10E2            movsd   %xmm2, %xmm4    # tmp67, f2
  62 001a F20F58D8            addsd   %xmm0, %xmm3    # tmp64, f0
  64 001e 660F1247            movlpd  24(%rdi), %xmm0 #, tmp68
  64      18
  65 0023 F20F58E0            addsd   %xmm0, %xmm4    # tmp68, f2
  66 0027 F20F5CD0            subsd   %xmm0, %xmm2    # tmp68, f3
  21:test.cc ****     sumdiff(f0, f2, f[0], f[2]);
  69 002b F20F10C3            movsd   %xmm3, %xmm0    # f0, tmp71
  70 002f F20F58C4            addsd   %xmm4, %xmm0    # f2, tmp71
  71 0033 F20F5CDC            subsd   %xmm4, %xmm3    # f2, f0
  72 0037 F20F1107            movsd   %xmm0, (%rdi)   # tmp71,* f
  22:test.cc ****     sumdiff(f1, f3, f[1], f[3]);
  74 003b F20F10C1            movsd   %xmm1, %xmm0    # f1, tmp73
```

```
 75 003f F20F5CCA              subsd   %xmm2, %xmm1    # f3, f1
 77 0043 F20F115F              movsd   %xmm3, 16(%rdi) # f0,
 77      10
 79 0048 F20F58C2              addsd   %xmm2, %xmm0    # f3, tmp73
 80 004c F20F114F              movsd   %xmm1, 24(%rdi) # f1,
 80      18
 81 0051 F20F1147              movsd   %xmm0, 8(%rdi)  # tmp73,
 81      08
 23:test.cc **** }
```

Note that the assembler code is not always in sync with the corresponding source lines, especially with higher levels of optimization.

24.14 Eigenvectors of the Fourier and Hartley transform *

Let $a_S := a + \overline{a}$ be the symmetric part of a sequence a, then

$$\mathcal{F}\big[\,\mathcal{F}\big[\,a_S\,\big]\,\big] \;=\; a_S \tag{24.14-1}$$

Now let $u_+ := a_S + \mathcal{F}\big[\,a_S\,\big]$ and $u_- := a_S - \mathcal{F}\big[\,a_S\,\big]$, then

$$\mathcal{F}\big[\,u_+\,\big] \;=\; \mathcal{F}\big[\,a_S\,\big] + a_S \;=\; a_S + \mathcal{F}\big[\,a_S\,\big] \;=\; +1 \cdot u_+ \tag{24.14-2a}$$
$$\mathcal{F}\big[\,u_-\,\big] \;=\; \mathcal{F}\big[\,a_S\,\big] - a_S \;=\; -(a_S - \mathcal{F}\big[\,a_S\,\big]) \;=\; -1 \cdot u_- \tag{24.14-2b}$$

Both u_+ and u_- are symmetric. For $a_A := a - \overline{a}$, the antisymmetric part of a, we have

$$\mathcal{F}\big[\,\mathcal{F}\big[\,a_A\,\big]\,\big] \;=\; -a_A \tag{24.14-3}$$

Therefore with $v_+ := a_A + i\,\mathcal{F}\big[\,a_A\,\big]$ and $v_- := a_A - i\,\mathcal{F}\big[\,a_A\,\big]$:

$$\mathcal{F}\big[\,v_+\,\big] \;=\; \mathcal{F}\big[\,a_A\,\big] - i\,a_A \;=\; -i\,(a_A + i\,\mathcal{F}\big[\,a_A\,\big]) \;=\; -i \cdot v_+ \tag{24.14-4a}$$
$$\mathcal{F}\big[\,v_-\,\big] \;=\; \mathcal{F}\big[\,a_A\,\big] + i\,a_A \;=\; +i\,(a_A - i\,\mathcal{F}\big[\,a_A\,\big]) \;=\; +i \cdot v_- \tag{24.14-4b}$$

Both v_+ and v_- are antisymmetric. The sequences u_+, u_-, v_+, and v_- are *eigenvectors* of the FT, with *eigenvalues* $+1$, -1, $-i$ and $+i$ respectively. The eigenvectors are pairwise perpendicular. Using the relation

$$a \;=\; \frac{1}{2}\,(u_+ + u_- + v_+ + v_-) \tag{24.14-5}$$

we can, for a given sequence, find a transform that is a 'square root' of the FT: compute u_+, u_-, v_+, and v_-, and a transform $\mathcal{F}^\lambda\,[a]$ for $\lambda \in \mathbb{R}$ as

$$\mathcal{F}^\lambda\,[a] \;=\; \frac{1}{2}\Big((+1)^\lambda\,u_+ + (-1)^\lambda\,u_- + (-i)^\lambda\,v_+ + (+i)^\lambda\,v_-\Big) \tag{24.14-6}$$

Then $\mathcal{F}^0\,[a]$ is the identity and $\mathcal{F}^1\,[a]$ is the usual FT. The transform $\mathcal{F}^{1/2}\,[a]$ is a transform so that $\mathcal{F}^{1/2}\,\big[\mathcal{F}^{1/2}\,[a]\big] = \mathcal{F}\big[\,a\,\big]$, that is, a 'square root' of the FT. The transform $\mathcal{F}^{1/2}\,[a]$ is not unique as the expressions $\pm 1^{1/2}$ and $\pm i^{1/2}$ are not.

The eigenvectors of the Hartley Transform are

$$u_+ \;:=\; a + \mathcal{H}\,[a] \tag{24.14-7a}$$
$$u_- \;:=\; a - \mathcal{H}\,[a] \tag{24.14-7b}$$

The eigenvalues are ± 1, one has $\mathcal{H}\,[u_+] = +1 \cdot u_+$ and $\mathcal{H}\,[u_-] = -1 \cdot u_-$.

Let M be the $n \times n$ matrix corresponding to the length-n Fourier transform with positive sign σ, that is $M_{r,c} = 1/\sqrt{n}\,\exp\,(2\,\pi\,i\,r\,c/n)$. Then its characteristic polynomial (see relation 40.5-2 on page 864) is

$$p(x) \;=\; (x-1)^{\lfloor (n+4)/4 \rfloor}\,(x+1)^{\lfloor (n+2)/4 \rfloor}\,(x-i)^{\lfloor (n+1)/4 \rfloor}\,(x+i)^{\lfloor (n-1)/4 \rfloor} \tag{24.14-8}$$

We write $p(x) = x^n + c_{n-1}\,x^{n-1} + \ldots + c_1\,x + c_0$. The trace of the matrix M is

$$\mathrm{Tr}(M) \quad = \quad \sqrt{n}\,\sum_{k=0}^{n-1} \exp\left(2\,i\,\pi\,k^2/n\right) \tag{24.14-9}$$

It equals ($-c_{n-1}$, the negative sum of all roots of $p(x)$, and)

$$1+i,\ +1,\ 0,\ +i \tag{24.14-10}$$

for $n \bmod 4 \equiv 0, 1, 2, 3$, respectively. A closed form is $(1 + i^{-n}) \,/\, (1 - i)$. The generating function for the sequence of values is $((1 + i) - x) \,/\, \left(1 + (-1 + i)\,x - i\,x^2\right)$.

The determinant of M equals ($(-1)^n\,c_0$, $(-1)^n$ times the product of all roots of $p(x)$, and)

$$+i,\ +1,\ -1,\ -i,\ -i,\ -1,\ +1,\ +i \tag{24.14-11}$$

for $n \bmod 8 \equiv 0, 1, 2, \ldots, 7$. A closed form is $(1 + i)\,/2\ (1 + (-i)^n)$. The generating function for the sequence is $\left(i + x - x^2 - i\,x^3\right) \,/\, \left(1 + x^4\right)$.

Chapter 25

Number theoretic transforms (NTTs)

We introduce the number theoretic transforms (NTTs). After understanding the necessary concepts from number theory (see also chapter 37) it turns out that the routines for the fast NTTs are rather straightforward translations of the FFT algorithms. We give radix-2 and radix-4 routines but there should be no difficulty to translate any given (complex valued) FFT algorithm into the equivalent NTT algorithm. For the translation of real valued FFT (or FHT) routines one needs to express sines and cosines in modular arithmetic, this is presented in sections 37.12.6 and 37.12.7.

As no rounding errors occur with the underlying modular arithmetic the main application of NTTs is the fast computation of exact convolutions.

25.1 Prime moduli for NTTs

How to make a number theoretic transform out of your FFT:
'*Replace* $exp(\pm 2\pi i/n)$ *by a primitive n-th root of unity, done.*'

We want to implement FFTs in $\mathbb{Z}/m\mathbb{Z}$ (the ring of integers modulo some integer m) instead of $\mathbb{C}$, the (field of the) complex numbers. These FFTs are called *number theoretic transforms* (NTTs), *mod m FFTs* or (if m is a prime) *prime modulus transforms*.

There is a restriction for the choice of m: for a length n NTT we need a primitive n-th root of unity. A number r is called an n-th root of unity if $r^n = 1$. It is called a *primitive n-th root* if $r^k \neq 1 \; \forall \, k < n$.

In $\mathbb{C}$ matters are simple: $e^{\pm 2\pi i/n}$ is a primitive n-th root of unity for arbitrary n. For example, $e^{2\pi i/21}$ is a primitive 21-th root of unity. Now $r = e^{2\pi i/3}$ is also 21-th root of unity but not a primitive root, because $r^3 = 1$. A primitive n-th root of 1 in $\mathbb{Z}/m\mathbb{Z}$ is also called an *element of order n*. The 'cyclic' property of the elements r of order n lies in the heart of all FFT algorithms: $r^{n+k} = r^k$.

In $\mathbb{Z}/m\mathbb{Z}$ things are not that simple: for a given modulus m primitive n-th roots of unity do not exist for arbitrary n. They exist for some maximal order R only. Roots of unity of an order different from R are available only for the divisors d_i of R: r^{R/d_i} is a d_i-th root of unity because $(r^{R/d_i})^{d_i} = r^R = 1$.

Therefore n, the length of the transform, must divide the maximal order R. This is the first condition for NTTs.

The operations needed in FFTs are addition, subtraction and multiplication. Division is not needed, except for division by n for the final normalization after transform and back-transform. Division by n is multiplication by the inverse of n, so n must be invertible in $\mathbb{Z}/m\mathbb{Z}$.

Therefore n, the length of the transform, must be coprime to the modulus m: $\gcd(n, m) = 1$. This is the second condition for NTTs.

We restrict our attention to prime moduli, though NTTs are also possible with composite moduli. If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = \mathrm{GF}(p)$: all elements except 0 have inverses and 'division is possible'. Thereby the second condition is trivially fulfilled for all NTT lengths $n < p$: a prime p is coprime to all integers $n < p$.

Roots of unity are available for the maximal order $R = p-1$ and its divisors: Therefore the first condition on n for a length-n mod p NTT being possible is that n divides $p - 1$. This restricts the choice for p to primes of the form $p = v\,n + 1$: for length-$n = 2^k$ NTTs one will use primes like $p = 3 \cdot 5 \cdot 2^{27} + 1$ (31 bits), $p = 13 \cdot 2^{28} + 1$ (32 bits), $p = 3 \cdot 29 \cdot 2^{56} + 1$ (63 bits) or $p = 27 \cdot 2^{59} + 1$ (64 bits).

```
arg 1: 62 == wb  [word bits, wb<=63]  default=62
arg 2: 0.01 == deltab  [results are in the range [wb-deltab, wb]]  default=0.01
 minb = 61.99 = wb-0.01
arg 3: 44 == minx  [log_2(min(fftlen))]  default=44
  ---- x = 44: -----
4580495072570638337 = 0x3f91300000000001 = 1 + 2^44 * 83 * 3137          (61.9902 bits)
4581058022524059649 = 0x3f93300000000001 = 1 + 2^44 * 3 * 11 * 13 * 607  (61.9904 bits)
4582113553686724609 = 0x3f96f00000000001 = 1 + 2^44 * 3 * 7 * 79 * 157   (61.9907 bits)
4585702359639785473 = 0x3fa3b00000000001 = 1 + 2^44 * 3^2 * 11 * 2633    (61.9918 bits)
4587039365779161089 = 0x3fa8700000000001 = 1 + 2^44 * 7 * 193^2          (61.9923 bits)
4587391209500049409 = 0x3fa9b00000000001 = 1 + 2^44 * 3 * 17 * 5113      (61.9924 bits)
4588130081313914881 = 0x3fac500000000001 = 1 + 2^44 * 3 * 5 * 17387      (61.9926 bits)
4589572640569556993 = 0x3fb1700000000001 = 1 + 2^44 * 11 * 37 * 641      (61.9931 bits)
[--snip--]
4610999923171655681 = 0x3ffd900000000001 = 1 + 2^44 * 5 * 19 * 31 * 89   (61.9998 bits)
4611105476287922177 = 0x3ffdf00000000001 = 1 + 2^44 * 262111             (61.9998 bits)
  ---- x = 45: -----
4580336742896238593 = 0x3f90a00000000001 = 1 + 2^45 * 29 * 67^2          (61.9902 bits)
4581533011547258881 = 0x3f94e00000000001 = 1 + 2^45 * 3 * 5 * 8681       (61.9905 bits)
4584347761314365441 = 0x3f9ee00000000001 = 1 + 2^45 * 5 * 11 * 23 * 103  (61.9914 bits)
4587655092290715649 = 0x3faaa00000000001 = 1 + 2^45 * 3 * 7^2 * 887      (61.9925 bits)
[--snip--]
  ---- x = 48: -----
4585508845593296897 = 0x3fa3000000000001 = 1 + 2^48 * 11 * 1481          (61.9918 bits)
  ---- x = 49: -----
4582975570802900993 = 0x3f9a000000000001 = 1 + 2^49 * 7 * 1163           (61.991 bits)
4595360469778169857 = 0x3fc6000000000001 = 1 + 2^49 * 3^2 * 907          (61.9949 bits)
  ---- x = 50: -----
4601552919265804289 = 0x3fdc000000000001 = 1 + 2^50 * 61 * 67            (61.9968 bits)
```

Figure 25.1-A: Primes suitable for NTTs of lengths dividing 2^{44}.

```
     modulus (hex)       ==    factorization + 1     log(m-1)/log(2)
    0x3f40f80000000001  == 2^43.3^2.5^2.7^2.47+1         61.9831
    0x3c0eb50000000001  == 2^40.3^3.5^2.7^3.17+1         61.9083
    0x3d673d0000000001  == 2^40.3^2.5^3.7^2.73+1         61.9402
    0x3fc22b0000000001  == 2^40.3^2.5^2.7^2.379+1        61.9945
    0x3bf6190000000001  == 2^40.3^2.5^3.7.499+1          61.906
    0x3d1d690000000001  == 2^40.3^2.5^2.7.2543+1         61.9335
    0x3d8c270000000001  == 2^40.3^2.5^2.7.13.197+1       61.9436
    0x3e8e8d0000000001  == 2^40.3^2.5^2.7.19.137+1       61.9671
    0x3ee4af0000000001  == 2^40.3^2.5^2.7.2617+1         61.9748
    0x3ed23a0000000001  == 2^41.3^2.5^2.7.1307+1         61.9732
    0x3fafb60000000001  == 2^41.3^2.5^4.7.53+1           61.9929
    0x3c46140000000001  == 2^42.3^3.5^2.7.11.19+1        61.9135
    0x3e32440000000001  == 2^42.3^2.5^2.7.647+1          61.9588
    0x3d23900000000001  == 2^44.3^3.5^2.7.53+1           61.934
```

Figure 25.1-B: Primes suitable for NTTs of lengths dividing $2^{40}\,3^2\,5^2\,7$.

Primes suitable with NTTs can be generated with the program [FXT: mod/fftprimes-demo.cc]. A shortened sample output is shown in figure 25.1-A A few moduli that allow for transforms of lengths dividing $2^{40} \cdot 3^2 \cdot 5^2 \cdot 7$ are shown in figure 25.1-B, the data is taken from [FXT: mod/moduli.txt].

25.2 Implementation of NTTs

To implement NTTs (modulo m, length n) one has to implement modular arithmetics and replace $e^{\pm 2\pi i/n}$ by an primitive n-th root r of unity in $\mathbb{Z}/m\mathbb{Z}$ in the code. A C++ class implementing modular arithmetics in FXT is [FXT: `class mod` in mod/mod.h].

For the inverse transform one uses the (mod m) inverse r^{-1} of r that was used for the forward transform. The element r^{-1} is also a primitive n-th root. Methods for the computation of the modular inverse are described in section 37.1.4 on page 734 (*gcd* algorithm) and on page 746 (powering algorithm: $r^{-1} = r^{R-1}$).

While the notion of the Fourier transform as a 'decomposition into frequencies' seems to be meaningless for NTTs the algorithms are denoted with 'decimation in time/frequency' in analogy to those in the complex domain.

The nice feature of NTTs is that there is no loss of precision in the transform as with the floating point FFTs. Using the trigonometric recursion in its most naive form is mandatory, as the computation of roots of unity is expensive.

25.2.1 Radix-2 DIT NTT

Pseudo code for the radix-2 decimation in time (DIT) NTT (to be called with `ldn=log2(n)`):

```
procedure mod_fft_dit2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn

    rn := element_of_order(n)  // (mod_type)

    if is<0 then  rn := rn**(-1)

    revbin_permute(f[], n)

    for ldm:=1 to ldn
    {
        m  := 2**ldm
        mh := m/2

        dw := rn**(2**(ldn-ldm))   // (mod_type)
        w  := 1                    // (mod_type)

        for j:=0 to mh-1
        {
            for r:=0 to n-m step m
            {
                t1 := r+j
                t2 := t1+mh

                v := f[t2]*w  // (mod_type)
                u := f[t1]    // (mod_type)

                f[t1] := u+v
                f[t2] := u-v
            }

            w := w*dw  // trig recursion
        }
    }
}
```

As shown in section 20.3.1 on page 378 it is a good idea to extract the `ldm==1` stage of the outermost loop: Replace

```
    for ldm:=1 to ldn
    {
```

by

```
    for r:=0 to n-1 step 2
    {
        {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
    }
```

```
    for ldm:=2 to ldn
    {
```

The C++ implementation is given in [FXT: ntt/nttdit2.cc]:

```
void
ntt_dit2_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dit2()
// Decimation in time (DIT) radix-2 FFT
// Input data must be in revbin_permuted order
// ldn := base-2 logarithm of the array length
// is := sign of the transform
{
    const ulong n = 1UL<<ldn;

    for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);

    for (ulong ldm=2; ldm<=ldn; ++ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);

        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;

                mod v = f[t2] * w;
                mod u = f[t1];

                f[t1] = u + v;
                f[t2] = u - v;
            }
            w *= dw;
        }
    }
}

void
ntt_dit2(mod *f, ulong ldn, int is)
// Radix-2 decimation in time (DIT) NTT
{
    revbin_permute(f, 1UL<<ldn);
    ntt_dit2_core(f, ldn, is);
}
```

The elements of order 2^k are precomputed upon initialization of the `mod` class. The call to `mod::root2pow()` is a simple table lookup.

25.2.2 Radix-2 DIF NTT

Pseudo code for the radix-2 decimation in frequency (DIF) NTT:

```
procedure mod_fft_dif2(f[], ldn, is)
// mod_type f[0..2**ldn-1]
{
    n := 2**ldn
    dw := element_of_order(n)  // (mod_type)

    if is<0 then  dw := rn**(-1)

    for  ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2

        w := 1  // (mod_type)

        for j:=0 to mh-1
        {
            for r:=0 to n-m step m
            {
                t1 := r+j
                t2 := t1+mh
```

```
                v := f[t2]  // (mod_type)
                u := f[t1]  // (mod_type)

                f[t1] :=  u+v
                f[t2] := (u-v)*w
            }

            w := w*dw  // trig recursion
        }

        dw := dw*dw
    }

    revbin_permute(f[], n)
}
```

As in section 20.3.2 on page 381 extract the `ldm==1` stage of the outermost loop: Replace the line

```
    for  ldm:=ldn to 1 step -1
```

by

```
    for  ldm:=ldn to 2 step -1
```

and insert

```
    for r:=0 to n-1 step 2
    {
        {f[r], f[r+1]} := {f[r]+f[r+1], f[r]-f[r+1]}
    }
```

before the call of `revbin_permute(f[],n)`.

The C++ implementation is given in [FXT: ntt/nttdif2.cc]:

```
void
ntt_dif2_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dif2().
// Decimation in frequency (DIF) radix-2 NTT.
// Output data is in revbin_permuted order.
// ldn := base-2 logarithm of the array length.
// is := sign of the transform
{
    const ulong n = (1UL<<ldn);
    mod dw = mod::root2pow( is>0 ? ldn : -ldn );

    for (ulong ldm=ldn; ldm>1; --ldm)
    {
        const ulong m = (1UL<<ldm);
        const ulong mh = (m>>1);

        mod w = mod::one;

        for (ulong j=0; j<mh; ++j)
        {
            for (ulong r=0; r<n; r+=m)
            {
                const ulong t1 = r + j;
                const ulong t2 = t1 + mh;

                mod v = f[t2];
                mod u = f[t1];

                f[t1] = (u + v);
                f[t2] = (u - v) * w;
            }
            w *= dw;
        }
        dw *= dw;
    }

    for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
}

void
ntt_dif2(mod *f, ulong ldn, int is)
// Radix-2 decimation in frequency (DIF) NTT
{
    ntt_dif2_core(f, ldn, is);
    revbin_permute(f, 1UL<<ldn);
```

```
}
```

25.2.3 Radix-4 NTTs

The radix-4 versions of the NTT are straightforward translations of the routines that use complex numbers. We simply give the C++ implementations

Decimation in time (DIT) algorithm

Code for a radix-4 decimation in time (DIT) NTT [FXT: ntt/nttdit4.cc]:

```
static const ulong LX = 2;

void
ntt_dit4_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dit4()
// Decimation in time (DIT) radix-4 NTT
// Input data must be in revbin_permuted order
// ldn := base-2 logarithm of the array length
// is := sign of the transform
{
    const ulong n = (1UL<<ldn);

    if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    ulong ldm = LX + (ldn&1);
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a2 = f[i1] * w2;
                mod a1 = f[i2] * w;
                mod a3 = f[i3] * w3;

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = t02 + t13;
                f[i2] = t02 - t13;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i1] = t02 + t13;
                f[i3] = t02 - t13;
            }

            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }
}

void
ntt_dit4(mod *f, ulong ldn, int is)
```

```
// Radix-4 decimation in time (DIT) NTT
{
    revbin_permute(f, 1UL<<ldn);
    ntt_dit4_core(f, ldn, is);
}
```

Decimation in frequency (DIF) algorithm

Code for a radix-4 decimation in frequency (DIT) NTT [FXT: ntt/nttdif4.cc]:

```
static const ulong LX = 2;

void
ntt_dif4_core(mod *f, ulong ldn, int is)
// Auxiliary routine for ntt_dif4().
// Decimation in frequency (DIF) radix-4 NTT.
// Output data is in revbin_permuted order.
// ldn := base-2 logarithm of the array length.
// is := sign of the transform
{
    const ulong n = (1UL<<ldn);

    const mod imag = mod::root2pow( is>0 ? 2 : -2 );

    for (ulong ldm=ldn; ldm>=LX; ldm-=LX)
    {
        const ulong m = (1UL<<ldm);
        const ulong m4 = (m>>LX);

        const mod dw = mod::root2pow( is>0 ? ldm : -ldm );
        mod w = (mod::one);
        mod w2 = w;
        mod w3 = w;

        for (ulong j=0; j<m4; j++)
        {
            for (ulong r=0, i0=j+r; r<n; r+=m, i0+=m)
            {
                const ulong i1 = i0 + m4;
                const ulong i2 = i1 + m4;
                const ulong i3 = i2 + m4;

                mod a0 = f[i0];
                mod a1 = f[i1];
                mod a2 = f[i2];
                mod a3 = f[i3];

                mod t02 = a0 + a2;
                mod t13 = a1 + a3;

                f[i0] = (t02 + t13);
                f[i1] = (t02 - t13) * w2;

                t02 = a0 - a2;
                t13 = a1 - a3;
                t13 *= imag;

                f[i2] = (t02 + t13) * w;
                f[i3] = (t02 - t13) * w3;
            }

            w *= dw;
            w2 = w * w;
            w3 = w * w2;
        }
    }

    if ( ldn & 1 )  // n is not a power of 4, need a radix-2 step
    {
        for (ulong i=0; i<n; i+=2)  sumdiff(f[i], f[i+1]);
    }
}

void
ntt_dif4(mod *f, ulong ldn, int is)
// Radix-4 decimation in frequency (DIF) NTT
{
    ntt_dif4_core(f, ldn, is);
    revbin_permute(f, 1UL<<ldn);
}
```

25.3 Convolution with NTTs

The NTTs are natural candidates for the computation of *exact* integer convolutions, as used in high precision multiplication algorithms. One must keep in mind that 'everything is mod m', the largest value that can be represented is $m-1$. One can simply choose a big enough m to get rid of this problem. If m does not fit into a single machine word this may slow down the computation unacceptably.

It is better to choose m as the product of several coprime moduli m_i that are all just below machine word size, compute the convolutions for each modulus m_i, and finally use the Chinese Remainder Theorem (see section 37.7 on page 747) to obtain the result modulo m.

If length-n FFTs are used for convolution there must be an inverse element for n. This imposes the condition $\gcd(n, m) = 1$, the modulus must be coprime to n. For length-2^k FFTs this simply means that m must be odd.

C++ code for the NTT based exact convolution can be found in [FXT: ntt/nttcnvl.cc]. The routines are virtually identical to their complex equivalents. For example, a routine for cyclic self-convolution can be given as

```
void
ntt_auto_convolution(mod *f, ulong ldn)
// Cyclic (self-)convolution
// Use zero padded data for linear convolution.
{
    assert_two_invertible();  // so we can normalize later

    const int is = +1;

    ntt_dif4_core(f, ldn, is);  // transform

    const ulong n = (1UL<<ldn);
    for (ulong i=0; i<n; ++i)  f[i] *= f[i];  // multiply element-wise

    ntt_dit4_core(f, ldn, -is);  // inverse transform

    multiply_val(f, n, (mod(n)).inv() );  // normalize
}
```

The revbin permutations are avoided as explained in section 21.1.3 on page 411.

For further applications of the NTT see the survey article [128] and the references given there.

Chapter 26

Fast wavelet transforms

The discrete wavelet transforms are a class of transforms that can be computed in linear time. We treat wavelet transforms whose basis functions have compact support. These can be derived as a generalization of the Haar transform.

26.1 Wavelet filters

We motivate the *wavelet transform* as a generalization of the 'standard' Haar transform given in section 23.1 on page 465. We reformulate the Haar transform as a sequence of filtering steps.

We consider only (moving average) filters F defined by n coefficients ('taps') f_0, f_1, ..., f_{n-1}. Let A be the length-N sequence a_0, a_1, ..., a_{N-1}. We define $F_k(A)$ as the weighted sum

$$F_k(A) \quad := \quad \sum_{j=0}^{n-1} f_j \, a_{k+j \bmod N} \tag{26.1-1}$$

That is, $F_k(A)$ is the result of applying the filter F to the n elements a_k, a_{k+1}, a_{k+2}, ... a_{k+n-1}, possibly wrapping around.

Now assume that N is a power of two. Let H be the low-pass filter defined by $h_0 = h_1 = +1/\sqrt{2}$, and G be the high-pass filter defined by $g_0 = +1/\sqrt{2}$, $g_1 = -1/\sqrt{2}$. A single filtering step of the Haar transform consists of

- Computing the sums $s_0 = H_0(A)$, $s_2 = H_2(A)$, $s_4 = H_4(4)$, ..., $s_{N-2} = H_{N-2}(A)$
- Computing the differences $d_0 = G_0(A)$, $d_2 = G_2(A)$, $d_4 = G_4(4)$, ..., $d_{N-2} = G_{N-2}(A)$
- Writing the sums to the left half of A, and the differences to the right half:
  $A = [s_0, s_2, s_4, s_6, \ldots, s_{N-2}, d_0, d_2, d_4, d_6, \ldots, d_{N-2}]$

The Haar transform is obtained by applying the step to the whole sequence, then to its left half, then to its left quarter, ..., the left four elements, the left two elements. With the Haar transform no wrap-around occurs.

A the analogous filtering step for the wavelet transform is obtained by defining two length-n filters H (low-pass) and G (high-pass) subject to certain conditions. Firstly, we consider only filters with an even number n of coefficients.

Secondly, we define coefficients of G to be the reversed sequence of the coefficients of H with alternating signs:
$g_0 = +h_{n-1}$, $g_1 = -h_{n-2}$, $g_2 = +h_{n-3}$, $g_4 = -h_{n-3}$, ..., $g_{n-3} = -h_2$, $g_{n-2} = +h_1$, $g_{n-1} = -h_0$.

Thirdly, we require that the resulting transform is orthogonal. Let S be the matrix corresponding to one filtering step, ignoring the order:

$$S\,A \;=\; [s_0, d_0, s_2, d_2, s_4, d_4, s_6, d_6, \ldots, s_{N-2}, d_{N-2}] \tag{26.1-2}$$

With length-6 filters and $N = 16$ the matrix S would be

$$S \;=\; \begin{bmatrix}
h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\
h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 \\
g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 \\
h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 \\
g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1
\end{bmatrix} \tag{26.1-3a}$$

$$= \begin{bmatrix}
+h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
+h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & +h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & +h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & +h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & \ldots & 0 \\
0 & 0 & 0 & 0 & +h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & \ldots & 0 \\
 & & & \ddots & & & & & & \ddots & & &
\end{bmatrix} \tag{26.1-3b}$$

The orthogonality requires that $S\,S^T = \mathrm{id}$, that is (setting $h_j = 0$ for $j < 0$ and $j \geq n$)

$$\sum_j h_j^2 \;=\; 1 \tag{26.1-4a}$$

$$\sum_j h_j\, h_{j+2} \;=\; 0 \tag{26.1-4b}$$

$$\sum_j h_j\, h_{j+4} \;=\; 0 \tag{26.1-4c}$$

In general, the following $n/2$ *wavelet conditions* are obtained:

$$\sum_j h_j^2 \;=\; 1 \tag{26.1-5a}$$

$$\sum_j h_j\, h_{j+2i} \;=\; 0 \quad \text{where} \quad i = 1, 2, 3, \ldots, n/2 - 1 \tag{26.1-5b}$$

We call a filter H satisfying these conditions a *wavelet filter*.

For the wavelet transform with $n = 2$ filter taps there is only condition, $h_0^2 + h_1^2 = 1$, leading to the parametric solution $h_0 = \sin(\phi)$, $h_1 = \cos(\phi)$. Setting $\phi = \pi/4$ one obtains $h_0 = h_1 = 1/\sqrt{2}$, corresponding to the Haar transform.

26.2 Implementation

A container class for wavelet filters is [FXT: `class wavelet_filter` in wavelet/waveletfilter.h]:

```
class wavelet_filter
{
public:
    double *h_;  // low-pass filter
    double *g_;  // high-pass filter
    ulong n_;    // number of taps

    void ctor_core()
    {
        h_ = new double[n_];
        g_ = new double[n_];
    }

    wavelet_filter(const double *w, ulong n=0)
    {
        if ( 0!=n )  n_ = n;
        else  // zero terminated array w[]
        {
            n_ = 0;
            while ( w[n_]!=0 )  ++n_;
        }

        ctor_core();

        for (ulong i=0, j=n_-1;  i<n_;  ++i, --j)
        {
            h_[i] = w[i];

            if ( !(i&1) )  g_[j] = -h_[i];  // even indices
            else           g_[j] = +h_[i];  // odd indices
        }
    }

  [--snip--]
```

The wavelet conditions can be checked via

```
    bool check(double eps=1e-6) const
    {
        if ( fabs(norm_sqr(0)-1.0) > eps )  return false;

        for (ulong i=1;  i<n_/2;  ++i)
            if ( fabs(norm_sqr(i)) > eps )  return false;

        return true;
    }
```

where `norm_sqr()` computes the sums in the relations 26.1-5a and 26.1-5b:

```
    static double norm_sqr(const double *h, ulong n, ulong s=0)
    {
        s *= 2;  // Note!
        if ( s>=n )  return 0.0;

        double v = 0;
        for (ulong k=0,j=s;  j<n;  ++k,++j)  v += (h[k]*h[j]);
        return  v;
    }

    double norm_sqr(ulong s=0)  const  { return norm_sqr(h_, n_, s); }
```

A wavelet step can be implemented as [FXT: wavelet/wavelet.cc]:

```
void
wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
    const ulong nh = (n>>1);
    const ulong m = n-1;  // mask to compute modulo n (n is a power of two)
    for (ulong i=0,j=0; i<n; i+=2,++j) // i \in [0,2,4,..,n-2];  j \in [0,1,2,..,n/2-1]
    {
        double s = 0.0,  d = 0.0;
        for (ulong k=0; k<wf.n_; ++k)
        {
            ulong w = (i+k) & m;
            s += (wf.h_[k] * f[w]);
            d += (wf.g_[k] * f[w]);
```

```
        }
        t[j] = s;
        t[nh+j] = d;
    }
    copy(t, f, n);  // f[] := t[]
}
```

The wavelet transform itself is

```
void
wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=n; m>=minm; m>>=1)  wavelet_step(f, m, wf, t);
}
```

The step for the inverse transform is [FXT: wavelet/invwavelet.cc]:

```
void
inverse_wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
    const ulong nh = (n>>1);
    const ulong m = n-1;  // mask to compute modulo n (n is a power of two)
    null(t, n);  // t[] := [0,0,...,0]
    for (ulong i=0, j=0;  i<n;  i+=2, ++j)
    {
        const double x = f[j],  y = f[nh+j];
        for (ulong k=0;  k<wf.n_;  ++k)
        {
            ulong w = (i+k) & m;
            t[w] += (wf.h_[k] * x);
            t[w] += (wf.g_[k] * y);
        }
    }
    copy(t, f, n);  // f[] := t[]
}
```

The inverse transform itself now is

```
void
inverse_wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=minm; m<=n; m<<=1)  inverse_wavelet_step(f, m, wf, t);
}
```

A readable source about wavelets is [246].

26.3 Moment conditions

As the wavelet conditions do not uniquely define the wavelet filters on can impose additional properties for the filters used. We require that, for an $2n$-tap wavelet filter, the first $n/2$ moments vanish:

$$\sum_j (-1)^j h_j \;=\; 0 \tag{26.3-1a}$$

$$\sum_j (-j)^k h_j \;=\; 0 \quad \text{where} \quad k = 1, 2, 3, \ldots, n/2-1 \tag{26.3-1b}$$

One motivation for these *moment conditions* is that for reasonably smooth signals (for which a polynomial approximation is good) the transform coefficients from the high-pass filter (the d_k) will be close to zero. With compression schemes that simply discard transform coefficients with small values this is a desirable property.

The class [FXT: `class wavelet_filter` in wavelet/waveletfilter.h] has a method to compute the moments of the filter:

```
static double moment(const double *h, ulong n, ulong x=0)
{
    if ( 0==x )
    {
        double v = 0.0;
        for (ulong k=0; k<n; k+=2)  v += h[k];
        for (ulong k=1; k<n; k+=2)  v -= h[k];
        return v;
    }

    double dk;

    double ve = 0;
    dk = 2.0;
    for (ulong k=2;  k<n;  k+=2, dk+=2.0)  ve += (pow(dk,x) * h[k]);

    double vo = 0;
    dk = 1.0;
    for (ulong k=1;  k<n;  k+=2, dk+=2.0)  vo += (pow(dk,x) * h[k]);
    return  ve - vo;
}

double moment(ulong x=0)  const  { return moment(h_, n_, x); }
```

Filter coefficients that satisfy the moment conditions are given in [FXT: wavelet/daubechies.cc]:

```
extern const double Daub1[] = {
+7.07106781186547524400844362104 8e-01,
+7.07106781186547524400844362104 8e-01 };

extern const double Daub2[] = {
+4.82962913144534143374871599864 4e-01,
+8.36516303737807905575293780916 8e-01,
+2.24143868042013381025972762240 4e-01,
-1.29409522551260381174449418812 0e-01 };

extern const double Daub3[] = {
+3.32670552950082615998511589139 0e-01,
+8.06891509311092576494493604088 7e-01,
+4.59877502118491570095151942147 6e-01,
-1.35011020010254588696389906699 3e-01,
-8.54412738820266616928191691817 7e-02,
+3.52262918857095366027406647155 1e-02 };

extern const double Daub4[] = {
+2.30377813308896500863291183044 0e-01,
+7.14846570552915647089921955273 9e-01,
+6.30880767929858907881716338300 6e-01,
-2.79837694168598542114137471800 7e-02,
-1.87034811719093084079570672789 0e-01,
+3.08413818355607636272193625349 5e-02,
+3.28830116668851997354075135492 4e-02,
-1.05974017850690321048832085240 2e-02 };

  [--snip--]
extern const double Daub38[] = {...}
```

The names reflect the number $n/2$ of vanishing moments. Reversing or negating the sequence of filter coefficients leads to trivial variants that also satisfies the moment conditions.

For the filters of length $n \geq 6$ there are solutions that are essentially different. For $n = 6$ there is one complex solution besides `Daub3[]`:

```
-0.09556007476957763 + 0.0508627772544*I
+0.08121662052705924 + 0.1525883317632*I
+0.72145023542906591 + 0.1017255545088*I
+0.72145023542906591 - 0.1017255545088*I
+0.08121662052705924 - 0.1525883317632*I
-0.09556007476957763 - 0.0508627772544*I
```

For $n = 8$ there is, besides `Daub4[]`, an additional real solution (left), and a complex one (right):

```
-0.07576571478950221                +0.02152475910155493 + 0.0184283603930*I
-0.02963552764600249                -0.06571356411493559 + 0.0176790547520*I
+0.49761866763277498                -0.1939761744607887 8 - 0.1319957453155*I
+0.80373875180513208                +0.2462766413907153 4 - 0.2801719341011*I
+0.29785779560530605                +0.8572304593176147 6 - 0.0921418019654*I
-0.09921954357663353                +0.5919931878573518 4 + 0.2064584925288*I
-0.01260396726203130                +0.0223277372281666 1 + 0.2057091868878*I
+0.03222310060405146                -0.0654494839465840 7 + 0.0560343868202*I
```

The numbers of solutions grows exponentially with n (the minimal polynomial of any tap value has degree 2^n). The filters given in [FXT: wavelet/daubechies.cc] are the filters for the so-called *Daubechies wavelets* (some closed form expressions for the filter coefficients are given in [80]).

Filter coefficients that satisfy the wavelet and the moment conditions can be found by a Newton iteration for zeros of the function $F : \mathbb{R}^n \to \mathbb{R}^n$, $F(\vec{h}) := \vec{w}$ where $w_i = F_i(\vec{h}) = F_i(h_0, h_1, \ldots, h_5)$. For example, with $n = 6$, the F_i are defined by

```
F[1]:  h0^2 + h1^2 + h2^2 + h3^2 + h4^2 + h5^2 - 1
F[2]:  h2*h0 + h3*h1 + h4*h2 + h5*h3
F[3]:  h4*h0 + h5*h1
F[4]:  -h0 + h1 + -h2 + h3 + -h4 + h5
F[5]:  h1 + -2*h2 + 3*h3 + -4*h4 + 5*h5
F[6]:  h1 + -4*h2 + 9*h3 + -16*h4 + 25*h5
```

The derivative is given by the *Jacobi matrix* J. It has the components $J_{r,c} := \dfrac{dF_r}{dh_c}$. Its rows are

```
J[1]= [2*h0, 2*h1, 2*h2, 2*h3, 2*h4, 2*h5]
J[2]= [h2, h3, h0 + h4, h1 + h5, h2, h3]
J[3]= [h4, h5, 0, 0, h0, h1]
J[4]= [-1, 1, -1, 1, -1, 1]
J[5]= [0, 1, -2, 3, -4, 5]
J[6]= [0, 1, -4, 9, -16, 25]
```

Now iterate (the equivalent to Newton's iteration, $x_{k+1} := x_k - f(x_k)/f'(x_k)$)

$$\vec{h}_{k+1} \quad := \quad \vec{h}_k - J^{-1}(\vec{h}_k)\, F(\vec{h}_k) \tag{26.3-2}$$

The computations have to be carried out with a rather great precision to avoid catastrophic loss of accuracy.

Part IV

Fast arithmetic

Chapter 27

Fast multiplication and exponentiation

The usual scheme for multiplication needs proportional N^2 operations for the multiplication of two N-digit numbers. This chapter describes multiplication algorithms that are asymptotically better than this, the Karatsuba algorithm, the Toom-Cook algorithms, and multiplication via FFTs. In addition, the left-to-right and right-to-left schemes for binary exponentiation are described.

27.1 Asymptotics of algorithms

An important feature of an algorithm is the number of operations that must be performed for the completion of a task of a certain size. For high precision computations one will take N as the length of the numbers counted in decimal digits or bits. For computations with square matrices one may take for N the number of rows, or the number of entries in the matrix. An operation is typically a (machine word) multiplication plus an addition, one could also simply count machine instructions.

We now consider the computational cost of a given algorithm. An algorithm is said to have some asymptotics $f(N)$ if it needs proportional $f(N)$ operations for a task of size N. We express proportionality with $f(N)$ as $\sim f(N)$. Some examples:

- Addition of two N-digit numbers needs $\sim N$ operations (here: machine word additions).
- Ordinary multiplication of two N-digit numbers needs $\sim N^2$ operations.
- Counting elements equal to a given value in an unsorted array of length N costs $\sim N$ operations. The operations are reads from memory and comparisons.
- Deciding whether a sorted array of length N contains a given value costs $\sim \log(N)$ operations when binary search is used.
- Computing the Fourier transform of a length-N sequence by definition, that is, as N sums each of length N, costs $\sim N^2$.
- The FFT algorithms compute the Fourier transform of a length-N sequence in $\sim N \log(N)$ operations
- Matrix multiplication (of two $N \times N$ matrices by the obvious algorithm) is $\sim N^3$ (N^2 sums, each of N products).
- When the problem size M for matrix multiplication is taken to be the number of elements of the matrix ($M = N^2$), then the cost is 'only' $\sim M^{3/2}$.

- Deciding whether a N-digit binary number is even or odd costs 1 operation (lookup whether the lowest bit is zero). The cost is independent of the problem size.

We have simplified the considerations by assuming that for a given algorithm $\sim f(N)$ there is a constant C so that

$$\lim_{N\to\infty} \frac{\text{work}(N)}{f(N)} \quad = \quad C \tag{27.1-1}$$

where work(N) is the actual cost for problem size N. The approximate cost of an algorithm is expressed by the symbol $\approx C \cdot f(N)$. The constant C is often referred to as the *hidden constant*.

The algorithm with the 'best' asymptotics wins for some, possibly huge, problem size N. For smaller N another algorithm will be superior. For the exact break-even point the hidden constants are, of course, important. For example, let the algorithm `mult1` take $\approx 1.0 \cdot N^2$ operations ($C = 1.0$), `mult2` take $\approx 8.0 \cdot N \log_2(N)$ operations ($C = 8.0$). Then for $N < 64$ the algorithm `mult1` is faster, while for $N > 64$ the algorithm `mult2` is faster. Completely different algorithms may be optimal for the same task at different problem sizes. The hidden constants can be so large that the asymptotically better algorithm is never practical for any feasible problem size.

With many algorithms it is only possible to give lower and upper (asymptotic) bounds, especially if the program flow depends on the processed data. Upper bounds are usually denoted with $O(f(N))$, for example, FFT multiplication is $O(N \log(N))$ and computations that take a constant number of operations independent of the problem size are $O(1)$.

The space requirements in all methods given is a constant times the problem size N. In-place algorithms (like the FFT) use space $\sim N$, the remaining cases (like radix sort) typically need $\sim 2\,N$. Note that it is possible to construct algorithms where the computational cost alone is not of much value: integer factorization (up to a certain maximal value) costs just one lookup if a precomputed table of factorizations is stored in an (insanely) big array.

For a more fine-grained approach to measuring the costs of algorithms see [89].

27.2 Splitting schemes for multiplication

Ordinary multiplication is $\sim N^2$. Assuming the hidden constant equals one the computation of the product of two million-digit numbers would require $\approx 10^{12}$ operations. On a machine that does 1 billion operations per second the multiplication would need 1000 seconds. The following schemes leads to algorithms with superior asymptotics.

27.2.1 2-way splitting: the Karatsuba algorithm

The following algorithm is due to A. Karatsuba and Y. Ofman, it was given 1963 in [147].

Split the numbers A and B (assumed to have approximately the same length) into two pieces

$$\begin{aligned} A &= x\,A_1 + A_0 \\ B &= x\,B_1 + B_0 \end{aligned} \tag{27.2-1}$$

where x is a power of the radix (for decimal numbers the radix is 10) close to the half length of A and B. The usual multiplication scheme needs 4 multiplications with half precision for one multiplication with full precision:

$$A\,B \quad = \quad A_0 \cdot B_0 + x\,(A_0 \cdot B_1 + B_0 \cdot A_1) + x^2\,A_1 \cdot B_1 \tag{27.2-2}$$

Only the multiplications $A_i \cdot B_j$ need to be considered. The multiplications by x, a power of the radix, are only shifts. If we use the relation

$$A\,B \;=\; (1+x)\,A_0 \cdot B_0 + x\,(A_1 - A_0) \cdot (B_0 - B_1) + (x + x^2)\,A_1 \cdot B_1 \tag{27.2-3}$$

we need 3 multiplications with half precision for one multiplication with full precision. Applying the scheme recursively until the numbers to multiply are of machine size we obtain an algorithm whose asymptotic cost is $\sim N^{\log_2(3)} \approx N^{1.585}$. An alternative form of relation 27.2-3 is

$$A\,B \;=\; (1-x)\,A_0 \cdot B_0 + x\,(A_1 + A_0) \cdot (B_0 + B_1) + (x^2 - x)\,A_1 \cdot B_1 \tag{27.2-4}$$

For squaring use either of the following schemes

$$A^2 \;=\; (1+x)\,A_0^2 - x\,(A_1 - A_0)^2 + (x + x^2)\,A_1^2 \tag{27.2-5a}$$
$$A^2 \;=\; (1-x)\,A_0^2 + x\,(A_1 + A_0)^2 + (x^2 - x)\,A_1^2 \tag{27.2-5b}$$

We compute $8231^2 = 67749361$ with the first relation (27.2-5a):

```
8231^2  ==  (100*82+31)^2
        ==  (1+100)*31^2  - 100*(82-31)^2 + (100+100^2)*82^2
        ==  (1+100)*[961] - 100*[2601]    + (100+100^2)*[6724]
        ==  961 + 96100   - 260100        + 672400 + 67240000
        ==  67749361
```

Assume that the hidden constant equals 2 as there is more bookkeeping overhead than with the usual algorithm. Computing the product of two million-digit numbers would require $\approx 2 \cdot (10^6)^{1.585} \approx 6.47 \cdot 10^9$ operations, taking about 6.5 seconds on our computer.

The Karatsuba scheme for polynomial multiplication is given in section 38.2 on page 799.

27.2.2 3-way splitting

One can extend the above idea by splitting U and V into more than two pieces each, the resulting method is called *Toom-Cook algorithm* (the method is called *Toom algorithm* in [88], and *Cook-Toom algorithm* in [2]).

27.2.2.1 Zimmermann's 3-way multiplication

```
A = a2*x^2 + a1*x + a0
B = b2*x^2 + b1*x + b0

S0 = a0 * b0
S1 = (a2+a1+a0) * (b2+b1+b0)
S2 = (4*a2+2*a1+a0) * (4*b2+2*b1+b0)
S3 = (a2-a1+a0) * (b2-b1+b0)
S4 = a2 * b2

T1 = 2*S3 + S2
T1 /= 3  \\ division by 3
T1 += S0
T1 /= 2
T1 -= 2*S4
T2 = (S1 + S3)/2
S1 -= T1
S2 = T2 - S0 - S4
S3 = T1 - T2

P = S4*x^4 + S3*x^3 + S2*x^2 + S1*x + S0
P - A*B  \\ == zero
```

Figure 27.2-A: Implementation of Zimmermann's 3-way multiplication scheme in pari/gp.

The good scheme for 3-way splitting is due to Paul Zimmermann. We compute the product $C = A \cdot B$ of two numbers, A and B

$$A = a_2\,x^2 + a_1\,x + a_0 \tag{27.2-6a}$$
$$B = b_2\,x^2 + b_1\,x + b_0 \tag{27.2-6b}$$
$$C = A \cdot B = c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0 \tag{27.2-6c}$$

by the following scheme (taken from [79]): set

$$S_0 := a_0 \cdot b_0 \tag{27.2-6d}$$
$$S_1 := (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0) \tag{27.2-6e}$$
$$S_2 := (4\,a_2 + 2\,a_1 + a_0) \cdot (4\,b_2 + 2\,b_1 + b_0) \tag{27.2-6f}$$
$$S_3 := (a_2 - a_1 + a_0) \cdot (b_2 - b_1 + b_0) \tag{27.2-6g}$$
$$S_4 := a_2 \cdot b_2 \tag{27.2-6h}$$

This costs 5 multiplications of length $N/3$. We already have found $c_0 = S_0$ and $c_4 = S_4$. We determine c_1, c_2, and c_3 by the following assignments (in the given order):

$$T_1 := 2\,S_3 + S_2 \qquad (= 18\,c_4 + 6\,c_3 + 6\,c_2 + 3\,c_0) \tag{27.2-6i}$$
$$T_1 := T_1/3 \qquad (= 6\,c_4 + 2\,c_3 + 2\,c_2 + c_0) \qquad \text{exact division by 3} \tag{27.2-6j}$$
$$T_1 := T_1 + S_0 \qquad (= 6\,c_4 + 2\,c_3 + 2\,c_2 + 2\,c_0) \tag{27.2-6k}$$
$$T_1 := T_1/2 \qquad (= 3\,c_4 + c_3 + c_2 + c_0) \tag{27.2-6l}$$
$$T_1 := T_1 - 2\,S_4 \qquad (= c_4 + c_3 + c_2 + c_0) \tag{27.2-6m}$$
$$T_2 := (S_1 + S_3)/2 \qquad (= c_4 + c_2 + c_0) \tag{27.2-6n}$$
$$S_1 := S_1 - T_1 \qquad (= c_1) \qquad \text{wrong in cited paper} \tag{27.2-6o}$$
$$S_2 := T_2 - S_0 - S_4 \qquad (= c_2) \tag{27.2-6p}$$
$$S_3 := T_1 - T_2 \qquad (= c_3) \tag{27.2-6q}$$

Now we have

$$C = A \cdot B = S_4\,x^4 + S_3\,x^3 + S_2\,x^2 + S_1\,x + S_0 \tag{27.2-6r}$$

The complexity of recursive multiplication based on this splitting scheme is $N^{\log_3(5)} \approx N^{1.465}$. Assume that the hidden constant again equals 2. Then the computation of the product of two million-digit numbers would require $\approx 2 \cdot (10^6)^{1.465} \approx 1.23 \cdot 10^9$ operations, taking about 1.2 seconds on our computer.

Note the division by 3 in relation 27.2-6j. A division by a constant (that is not a power of two) cannot be avoided in n-way splitting schemes for multiplication for $n \geq 3$. There are squaring schemes that do not involve such divisions.

27.2.2.2 3-way multiplication by Bodrato and Zanoni

An alternative algorithm for 3-way splitting is suggested in [47]: setup S_0, S_1, ..., S_4 as in relations 27.2-6d... 27.2-6h, then compute, in the given order,

$$S_2 := (S_2 - S_3)/3 \qquad (= 5\,c_4 + 3\,c_3 + c_2 + c_1) \qquad \text{exact division by 3} \tag{27.2-7a}$$
$$S_3 := (S_1 - S_3)/2 \qquad (= c_3 + c_1) \tag{27.2-7b}$$
$$S_1 := S_1 - S_0 \qquad (= c_4 + c_3 + c_2 + c_1) \tag{27.2-7c}$$
$$S_2 := (S_2 - S_1)/2 \qquad (= 2\,c_4 + c_3) \tag{27.2-7d}$$
$$S_1 := S_1 - S_3 - S_4 \qquad (= c_2) \tag{27.2-7e}$$
$$S_2 := S_2 - 2\,S_4 \qquad (= c_3) \tag{27.2-7f}$$
$$S_3 := S_3 - S_2 \qquad (= c_1) \tag{27.2-7g}$$

```
A = a2*x^2 + a1*x + a0
B = b2*x^2 + b1*x + b0

S0 = a0 * b0
S1 = (a2+a1+a0) * (b2+b1+b0)
S2 = (4*a2+2*a1+a0) * (4*b2+2*b1+b0)
S3 = (a2-a1+a0) * (b2-b1+b0)
S4 = a2 * b2

S2 = (S2 - S3)/3  \\ division by 3
S3 = (S1 - S3)/2
S1 = S1 - S0
S2 = (S2 - S1)/2
S1 = S1 - S3 - S4
S2 = S2 - 2*S4
S3 = S3 - S2

P = S4*x^4+ S2*x^3+ S1*x^2+ S3*x + S0

P - A*B \\ == zero
```

Figure 27.2-B: Implementation of the 3-way multiplication scheme of Bodrato and Zanoni.

Now we have (note the order of the coefficients S_i)

$$C \;=\; A \cdot B \;=\; S_4\,x^4 + S_2\,x^3 + S_1\,x^2 + S_3\,x + S_0 \tag{27.2-7h}$$

The scheme requires only one multiplication by two, Zimmermann's scheme involves two.

27.2.2.3 3-way squaring

The following scheme is taken from [79]. To compute the square $C = A^2$ of a number A

$$A \;=\; a_2\,x^2 + a_1\,x + a_0 \tag{27.2-8a}$$
$$C \;=\; A^2 \;=\; S_4\,x^4 + S_3\,x^3 + S_2\,x^2 + S_1\,x + S_0 \tag{27.2-8b}$$

set

$$S_0 \;:=\; a_0^2 \tag{27.2-8c}$$
$$S_1 \;:=\; (a_2 + a_1 + a_0)^2 \tag{27.2-8d}$$
$$S_2 \;:=\; (a_2 - a_1 + a_0)^2 \tag{27.2-8e}$$
$$S_3 \;:=\; 2\,a_1 \cdot a_2 \tag{27.2-8f}$$
$$S_4 \;:=\; a_2^2 \tag{27.2-8g}$$
$$\tag{27.2-8h}$$

This costs 4 squarings and 1 multiplication of length $N/3$. The quantities S_0, S_3, and S_4 are already correct. Determine S_1 and S_2 via

$$T_1 \;:=\; (S_1 + S_2)/2 \tag{27.2-8i}$$
$$S_1 \;:=\; S_1 - T_1 - S_3 \tag{27.2-8j}$$
$$S_2 \;:=\; T_1 - S_4 - S_0 \tag{27.2-8k}$$

27.2.3 4-way splitting

27.2.3.1 4-way multiplication

An elegant and clean scheme for 4-way splitting of a multiplication is given by Bodrato and Zanoni in [48]. A pari/gp implementation is shown in figure 27.2-C. The algorithm has asymptotics $\sim O(n^{\log_4(7)}) \approx O(n^{1.403})$. In general, an s-way splitting scheme will be $\sim O(n^{\log_s(2s+1)})$.

```
A = a3*x^3 + a2*x^2 + a1*x + a0
B = b3*x^3 + b2*x^2 + b1*x + b0

S1 = a3*b3
S2 = (8*a3+4*a2+2*a1+a0)*(8*b3+4*b2+2*b1+b0)
S3 = (+a3+a2+a1+a0)*(+b3+b2+b1+b0)
S4 = (-a3+a2-a1+a0)*(-b3+b2-b1+b0)
S5 = (+8*a0+4*a1+2*a2+a3)*(+8*b0+4*b1+2*b2+b3);
S6 = (-8*a0+4*a1-2*a2+a3)*(-8*b0+4*b1-2*b2+b3)
S7 = a0*b0


S2 += S5
S4 -= S3
S6 -= S5
S4 /= 2  \\
S5 -= S1
S5 -= (64*S7)
S3 += S4
S5 *= 2;  S5 += S6

S2 -= (65*S3)
S3 -= S1
S3 -= S7
S4 = -S4 \\
S6 = -S6 \\
S2 += (45*S3)
S5 -= (8*S3)
S5 /= 24  \\ division by 24

S6 -= S2
S2 -= (16*S4)
S2 /= 18  \\ division by 18
S3 -= S5
S4 -= S2
S6 += (30*S2)
S6 /= 60   \\ division by 60
S2 -= S6

P = S1*x^6 + S2*x^5 + S3*x^4 + S4*x^3 + S5*x^2 + S6*x + S7;

P - A*B \\ == zero
```

Figure 27.2-C: Implementation of the 4-way multiplication scheme in pari/gp.

27.2.3.2 4-way squaring

The following scheme is taken from [79]

$$A = a_3\,x^3 + a_2\,x^2 + a_1\,x + a_0 \tag{27.2-9a}$$
$$C = A^2 = c_6\,x^6 + c_5\,x^5 + c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0 \tag{27.2-9b}$$

Set

$$S_1 := a_0^2 \tag{27.2-9c}$$
$$S_2 := 2\,a_0 \cdot a_1 \tag{27.2-9d}$$
$$S_3 := (a_0 + a_1 - a_2 - a_3) \cdot (a_0 - a_1 - a_2 + a_3) \tag{27.2-9e}$$
$$S_4 := (a_0 + a_1 + a_2 + a_3)^2 \tag{27.2-9f}$$
$$S_5 := 2\,(a_0 - a_2) \cdot (a_1 - a_3) \tag{27.2-9g}$$
$$S_6 := 2\,a_3 \cdot a_2 \tag{27.2-9h}$$
$$S_7 := a_3^2 \tag{27.2-9i}$$

```
A = a3*x^3 + a2*x^2 + a1*x + a0

S1 = a0^2
S2 = 2 *a0 *a1
S3 = (a0 + a1 - a2 - a3) * (a0 - a1 - a2 + a3)
S4 = (a0 + a1 + a2 + a3 )^2
S5 = 2*(a0 - a2)*(a1 - a3)
S6 = 2*a3*a2
S7 = a3^2

T1 = S3 + S4
T2 = (T1 + S5)/2
T3 = S2 + S6
T4 = T2 - T3
T5 = T3 - S5
T6 = T4 - S3
T7 = T4 - S1
T8 = T6 - S7

P = S7 *x^6 + S6 *x^5 + T7 *x^4 + T5 *x^3 + T8 *x^2 + S2 *x + S1
P - A^2  \\ == zero
```

Figure 27.2-D: Implementation of the 4-way squaring scheme in pari/gp.

Then set, in the given order,

$$T_1 := S_3 + S_4 \tag{27.2-9j}$$
$$T_2 := (T_1 + S_5)/2 \tag{27.2-9k}$$
$$T_3 := S_2 + S_6 \tag{27.2-9l}$$
$$T_4 := T_2 - T_3 \tag{27.2-9m}$$
$$T_5 := T_3 - S_5 \tag{27.2-9n}$$
$$T_6 := T_4 - S_3 \tag{27.2-9o}$$
$$T_7 := T_4 - S_1 \tag{27.2-9p}$$
$$T_8 := T_6 - S_7 \tag{27.2-9q}$$

The square then equals

$$C = S_7\,x^6 + S_6\,x^5 + T_7\,x^4 + T_5\,x^3 + T_8\,x^2 + S_2\,x + S_1 \tag{27.2-9r}$$

27.2.4 5-way splitting

27.2.4.1 5-way multiplication

The scheme for 5-way splitting of a multiplication shown in figure 27.2-E is given in [48]. As for the 4-way multiplication scheme, no temporaries are used.

27.2.4.2 5-way squaring

We describe the 5-way squaring scheme given in [47]. Let

$$A = a_4\,x^4 + a_3\,x^3 + a_2\,x^2 + a_1\,x + a_0 \tag{27.2-10}$$
$$C = A^2 = c_8\,x^8 + c_7\,x^7 + c_6\,x^6 + c_5\,x^5 + c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0 \tag{27.2-11}$$

```
A = a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
B = b4*x^4 + b3*x^3 + b2*x^2 + b1*x + b0


S1 = a4*b4

S2 = (a0-2*a1+4*a2-8*a3+16*a4)*(b0-2*b1+4*b2-8*b3+16*b4)
S5 = (a0+2*a1+4*a2+8*a3+16*a4)*(b0+2*b1+4*b2+8*b3+16*b4)

S3 = (a4+2*a3+4*a2+8*a1+16*a0)*(b4+2*b3+4*b2+8*b1+16*b0)
S8 = (a4-2*a3+4*a2-8*a1+16*a0)*(b4-2*b3+4*b2-8*b1+16*b0)

S4 = (a0+4*a1+16*a2+64*a3+256*a4)*(b0+4*b1+16*b2+64*b3+256*b4)

S6 = (a0-a1+a2-a3+a4)*(b0-b1+b2-b3+b4)
S7 = (a0+a1+a2+a3+a4)*(b0+b1+b2+b3+b4)

S9 = a0*b0


S6 -= S7
S2 -= S5
S4 -= S9
S4 -= (2^16*S1)
S8 -= S3
S6 /= 2 \\
S5 *= 2;  S5 += S2
S2 = -S2 \\
S8 = -S8 \\
S7 += S6
S6 = -S6 \\
S3 -= S7
S5 -= (512*S7)
S3 *= 2;  S3 -= S8

S7 -= S1
S7 -= S9
S8 += S2
S5 += S3
S8 -= (80*S6)
S3 -= (510*S9)
S4 -= S2
S3 *= 3;  S3 += S5
S8 /= 180  \\ division by 180
S5 += (378*S7)
S2 /= 4
S6 -= S2
S5 /= (-72)  \\ division by -72
S3 /= (-360)  \\ division by -360

S2 -= S8
S7 -= S3
S4 -= (256*S5)
S3 -= S5
S4 -= (4096*S3)
S4 -= (16*S7)
S4 += (256*S6)
S6 += S2
S2 *= 180;  S2 += S4
S2 /= 11340  \\ division by 11340
S4 += (720*S6)
S4 /= (-2160)  \\ division by -2160
S6 -= S4
S8 -= S2


P = S1*x^8 + S2*x^7 + S3*x^6 + S4*x^5 + S5*x^4 + S6*x^3 + S7*x^2 + S8*x + S9;

P - A*B \\ == zero
```

Figure 27.2-E: Implementation of the 5-way multiplication scheme in pari/gp.

```
A = a4*x^4+ a3*x^3 + a2*x^2 + a1*x + a0

S1 = a0^2
S2 = a4^2
S3 = (a0 + a1 + a2 + a3 + a4)^2
S4 = (a0 - a1 + a2 - a3 + a4)^2
S5 = 2* (a0-a2+a4) * (a1-a3)
S6 = (a0 + a1 - a2 - a3 + a4) * (a0 - a1 - a2 + a3 + a4)
S7 = (a1 + a2 - a4) * (a1 - a2 - a4 + 2*(a0-a3))
S8 = 2*a0*a1
S9 = 2*a3*a4

S4 = (S4+S3)/2
S3 = S3-S4
S6 = (S6+S4)/2
S5 = (-S5+S3)/2
S4 = S4-S6
S3 = S3-S5-S8
S6 = S6-S2-S1
S5 = S5-S9
S7 = S7-S2-S8-S9+S6+S3
S4 = S4-S7

P = S2*x^8+S9*x^7+S4*x^6+S3*x^5+S6*x^4+S5*x^3+S7*x^2+S8*x+S1
P - A^2  \\ == zero
```

Figure 27.2-F: Implementation of the 5-way squaring scheme,

Set

$$
\begin{aligned}
S_1 &:= a_0^2 && \text{(27.2-12a)}\\
S_2 &:= a_4^2 && \text{(27.2-12b)}\\
S_3 &:= (a_0 + a_1 + a_2 + a_3 + a_4)^2 && \text{(27.2-12c)}\\
S_4 &:= (a_0 - a_1 + a_2 - a_3 + a_4)^2 && \text{(27.2-12d)}\\
S_5 &:= 2\,(a_0 - a_2 + a_4) \cdot (a_1 - a_3) && \text{(27.2-12e)}\\
S_6 &:= (a_0 + a_1 - a_2 - a_3 + a_4) \cdot (a_0 - a_1 - a_2 + a_3 + a_4) && \text{(27.2-12f)}\\
S_7 &:= (a_1 + a_2 - a_4) \cdot (a_1 - a_2 - a_4 + 2\,(a_0 - a_3)) && \text{(27.2-12g)}\\
S_8 &:= 2\,a_0 \cdot a_1 && \text{(27.2-12h)}\\
S_9 &:= 2\,a_3 \cdot a_4 && \text{(27.2-12i)}
\end{aligned}
$$

Further set, in the order given,

$$
\begin{aligned}
S_4 &= (S_4 + S_3)/2 \qquad (= c_0 + c_2 + c_4 + c_6 + c_8) && \text{(27.2-13a)}\\
S_3 &= S_3 - S_4 \qquad (= c_1 + c_3 + c_5 + c_7) && \text{(27.2-13b)}\\
S_6 &= (S_6 + S_4)/2 \qquad (= c_0 + c_4 + c_8) && \text{(27.2-13c)}\\
S_5 &= (-S_5 + S_3)/2 \qquad (= c_3 + c_7) && \text{(27.2-13d)}\\
S_4 &= S_4 - S_6 \qquad (= c_2 + c_6) && \text{(27.2-13e)}\\
S_3 &= S_3 - S_5 - S_8 \qquad (= c_5) && \text{(27.2-13f)}\\
S_6 &= S_6 - S_2 - S_1 \qquad (= c_4) && \text{(27.2-13g)}\\
S_5 &= S_5 - S_9 \qquad (= c_3) && \text{(27.2-13h)}\\
S_7 &= S_7 - S_2 - S_8 - S_9 + S_6 + S_3 \qquad (= c_2) && \text{(27.2-13i)}\\
S_4 &= S_4 - S_7 \qquad (= c_6) && \text{(27.2-13j)}
\end{aligned}
$$

Now we have (note the order of the coefficients S_i)

$$C \;=\; A^2 \;=\; S_2\,x^8 + S_9\,x^7 + S_4\,x^6 + S_3\,x^5 + S_6\,x^4 + S_5\,x^3 + S_7\,x^2 + S_8\,x + S_1 \qquad \text{(27.2-13k)}$$

```
A = a4*x^4+ a3*x^3 + a2*x^2 + a1*x + a0

S1 = a0^2
S2 = a4^2
[ ...S9,  as before]

T1 = S1 + 2*S2 - S7 + 2*S8 + S9
T2 = S3 - S4
T3 = 2*S5
T4 = T2 + T3
T5 = T2 - T3
T6 = T4/4
T7 = T5/4 - S9
T8 = T1 - T6 - S6
T9 = T6 - S8
T10 = S3 + S6
T11 = (T10 + S4 + S6)/4
T12 = T11 - S1 - S2
T13 = (T10 + S5)/2
T14 = T13 - T1

P = S2*x^8 + S9*x^7 + T8*x^6 + T9*x^5 +  T12*x^4 + T7*x^3 + T14*x^2 + S8*x + S1
P - A^2 \\ == zero
```

Figure 27.2-G: Implementation of the alternative 5-way squaring scheme in pari/gp. Definition of $S_1,\ldots,S_9$ as in figure 27.2-F.

27.2.4.3 Alternative 5-way squaring scheme

The following scheme is taken from [79], we correct some errors in the paper. Setup $S_1,\ldots,S_9$ as given by relations 27.2-12a...27.2-12i, then compute, in the given order,

$$\begin{aligned}
T_1 &:= S_1 + 2\,S_2 - S_7 + 2\,S_8 + S_9 && \text{(27.2-14a)}\\
T_2 &:= S_3 - S_4 && \text{(27.2-14b)}\\
T_3 &:= 2\,S_5 && \text{(27.2-14c)}\\
T_4 &:= T_2 + T_3 && \text{(27.2-14d)}\\
T_5 &:= T_2 - T_3 && \text{(27.2-14e)}\\
T_6 &:= T_4/4 && \text{(27.2-14f)}\\
T_7 &:= T_5/4 - S_9 && \text{(27.2-14g)}\\
T_8 &:= T_1 - T_6 - S_6 && \text{(27.2-14h)}\\
T_9 &:= T_6 - S_8 && \text{(27.2-14i)}\\
T_{10} &:= S_3 + S_6 && \text{(27.2-14j)}\\
T_{11} &:= (T_{10} + S_4 + S_6)/4 && \text{(27.2-14k)}\\
T_{12} &:= T_{11} - S_1 - S_2 \qquad \text{wrong in cited paper} && \text{(27.2-14l)}\\
T_{13} &:= (T_{10} + S_5)/2 && \text{(27.2-14m)}\\
T_{14} &:= T_{13} - T_1 && \text{(27.2-14n)}
\end{aligned}$$

We have (note that the coefficients for x^4 and x^2 are wrong in the cited paper):

$$C = S_2\,x^8 + S_9\,x^7 + T_8\,x^6 + T_9\,x^5 + T_{12}\,x^4 + T_7\,x^3 + T_{14}\,x^2 + S_8\,x + S_1 \qquad \text{(27.2-14o)}$$

27.3 Fast multiplication via FFT

Multiplication of two numbers is essentially a convolution of the sequences of their digits. The (linear) convolution of the two sequences $a_k, b_k, k = 0\ldots N-1$ is defined as the sequence c where

$$c_k := \sum_{i,j=0;\ i+j=k}^{N-1} a_i b_j \qquad k = 0\ldots 2N-2 \qquad \text{(27.3-1)}$$

A (n-digit) number written in radix R as

$$a_{n-1}\, a_{n-2}\, \ldots\, a_2\, a_1\, a_0 \tag{27.3-2}$$

denotes a quantity of

$$\sum_{i=0}^{n-1} a_i \cdot R^i \quad = \quad a_{n-1} \cdot R^{n-1} + a_{n-2} \cdot R^{n-2} + \ldots + a_1 \cdot R + a_0 \tag{27.3-3}$$

This means, the digits can be identified with coefficients of a polynomial in R. For example, with decimal numbers one has $R = 10$, and the number 578 equals $5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0$. The product of two numbers is almost the polynomial product

$$\sum_{k=0}^{2N-2} c_k\, R^k \quad := \quad \sum_{i=0}^{N-1} a_i\, R^i \cdot \sum_{j=0}^{N-1} b_j\, R^j \tag{27.3-4}$$

The c_k are found by comparing coefficients. One easily checks that the c_k must satisfy the convolution equation 27.3-1. As the c_k can be greater than 'nine' (that is, $R - 1$), the result has to be 'fixed' using *carry* operations: Go from right to left, replace c_k by $c'_k = c_k \bmod R$ and add $(c_k - c'_k)/R$ to its left neighbor.

An example: usually one would multiply the numbers 82 and 34 as follows:

	82	×	34	
	3	^{3}2	8	
2	^{2}4	6		
= 2	7	8	8	

We have seen that the carries can be delayed to the end of the computation:

	82	×	34	
		32	8	
	24	6		
	24	38	8	
= 2	^{2} 7	^{3} 8	8	

... which is really polynomial multiplication (which in turn is a convolution of the coefficients):

$(8\,x + 2)$	×	$(3\,x + 4)$	
	$32\,x$	8	
$24\,x^2$	$6\,x$		
= $24\,x^2$	$+38\,x$	$+8$	

The value of the polynomial $24\,x^2 + 38\,x + 8$ for $x = 10$ is 2788.

Convolution can be done efficiently using the Fast Fourier Transform (FFT): convolution is a simple (element-wise) multiplication in Fourier space. The FFT itself takes $\sim N \cdot \log N$ operations. Instead of the direct convolution ($\sim N^2$) one proceeds as follows:

- Compute the FFTs of both factors.
- Multiply the transformed sequences element-wise.
- Compute inverse transform of the product.

To understand why this actually works note that (1) the multiplication of two polynomials can be achieved by the (more complicated) scheme:

- evaluate both polynomials at sufficiently many points (at least one more point than the degree of the product polynomial c: $\deg c = \deg a + \deg b$)

- element-wise multiply the values found
- find the polynomial corresponding to those (product-)values

and (2) that *the FFT is an algorithm for the parallel evaluation of a given polynomial at many points, namely the roots of unity.* (3) the inverse FFT is an algorithm to find (the coefficients of) a polynomial whose values are given at the roots of unity.

You might be surprised if you always thought of the FFT as an algorithm for the 'decomposition into frequencies'. There is no problem with either of these notions.

Re-launching our example ($82 \cdot 34 = 2788$), we use the fourth roots of unity ± 1 and $\pm i$:

	$a = (8\,x + 2)$	$\times$	$b = (3\,x + 4)$	$c = a\,b$
$+1$	$+10$		$+7$	$+70$
$+i$	$+8i + 2$		$+3i + 4$	$+38i - 16$
-1	-6		$+1$	-6
$-i$	$-8i + 2$		$-3i + 4$	$-38i - 16$
				$c = (24\,x^2 + 38\,x + 8)$

This table has to be read as follows: first the given polynomials a and b are evaluated at the points given in the left column, thereby the columns below a and b are filled. Then the values are multiplied to fill the column below c, giving the values of c at the points. Finally, the actual polynomial c is found from those values, resulting in the lower right entry. You may find it instructive to verify that a 4-point FFT really evaluates a, b by transforming the sequences 0, 0, 8, 2 and 0, 0, 3, 4 by hand. The backward transform of 70, $38i - 16$, -6, $-38i - 16$ should produce the final result given for c.

The operation count is dominated by that of the FFTs (the element-wise multiplication is of course $\sim N$), so the whole fast convolution algorithm takes $\sim N \cdot \log N$ operations. The following carry operation is also $\sim N$ and can therefore be neglected when counting operations.

Assume the hidden constant equals five. Multiplying our million-digit numbers will need about

$$5 \cdot 10^6 \log_2(10^6) \quad \approx \quad 5 \cdot 10^6 \cdot 20 \;=\; 10^8 \;=\; 0.1 \cdot 10^9$$

operations, taking approximately a tenth second on our computer.

Strictly speaking $N \cdot \log N$ is not really the truth: it has to be $N \cdot \log N \cdot \log\log N$. This is because the sums in the convolutions have to be represented as exact integers. The biggest term C that can possibly occur is approximately $N\,R^2$ for a number with N digits (see next section). Therefore, working with some fixed radix R one has to compute the FFTs with $\log N$ bits precision, leading to an operation count of $N \cdot \log N \cdot \log N$. The slightly better $N \cdot \log N \cdot \log\log N$ can be obtained by recursive use of FFT multiplies. For realistic applications (where the sums in the convolution all fit into the machine type floating point numbers) it is safe to think of FFT multiplication being proportional $N \cdot \log N$.

For a survey of multiplication methods, some mathematical background and further references see [37]. Several alternative multiplication algorithms are given in [155, chapter 4.3.3]. See [162] on how far the idea "polynomials for numbers" can be carried and where it fails.

27.4 Radix/precision considerations with FFT multiplication

This section describes the dependencies between the radix of the number and the achievable precision when using FFT multiplication.

We use (unsigned) 16-bit words for the digits. Thereby the radix of the numbers can be in the range 2, 3, ..., 65536($= 2^{16}$). When working in base ten one will actually use 'super-digits' of base 10,000, the largest power of ten that fits into a 16-bit word. These super-digits are called `LIMB`s in hfloat.

With very large precision one cannot always use the greatest power of the desired base. This is due to the fact that the components of the convolution must be representable as integer numbers with the data type used for the FFTs: The cumulative sums c_k have to be represented precisely enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a c_k will appear in the middle of the product and when multiplicand and multiplier consist of 'nines' (that is $R-1$) only. For radix R and a precision of N `LIMB`s Let the maximal possible value be C, then

$$C \quad = \quad N\,(R-1)^2 \tag{27.4-1}$$

Note that with FFT based convolution the *absolute value* of the central term can in fact equal $|C| = N^2\,(R-1)^2$. But there is no need to distinguish that many integers. After dividing by N we are back at relation 27.4-1.

The number of bits to represent C exactly is the integer greater or equal to

$$\log_2(N\,(R-1)^2) \quad = \quad \log_2 N + 2\,\log_2(R-1) \tag{27.4-2}$$

Due to numerical errors there must be a few more bits for safety. If computations are made using double-precision floating point numbers (C-type `double`) one typically has a mantissa of 53 bits[1] then we need to have

$$M \quad \geq \quad \log_2 N + 2\,\log_2(R-1) + S \tag{27.4-3}$$

where M :=mantissa-bits and S :=safety-bits. Using $\log_2(R-1) < \log_2(R)$:

$$N_{max}(R) \quad = \quad 2^{M-S-2\,\log_2(R)} \tag{27.4-4}$$

Suppose we have $M = 53$ mantissa-bits and require $S = 3$ safety-bits. With base 2 numbers one could use radix $R = 2^{16}$ for precisions up to a length of $N_{max} = 2^{53-3-2\cdot 16} = 256k$ `LIMB`s. Corresponding are 4096 kilo bits and = 1024 kilo hex digits. For greater lengths smaller radices have to be used according to the following table (extra horizontal line at the 16 bit limit for `LIMB`s):

Radix R	max # `LIMB`s	max # hex digits	max # bits
$2^{10} = 1024$	$1048,576\,k$	$2621,440\,k$	$10240\,M$
$2^{11} = 2048$	$262,144\,k$	$720,896\,k$	$2816\,M$
$2^{12} = 4096$	$65,536\,k$	$196,608\,k$	$768\,M$
$2^{13} = 8192$	$16384\,k$	$53,248\,k$	$208\,M$
$2^{14} = 16384$	$4096\,k$	$14,336\,k$	$56\,M$
$2^{15} = 32768$	$1024\,k$	$3840\,k$	$15\,M$
$2^{16} = 65536$	$256\,k$	$1024\,k$	$4\,M$
$2^{17} = 128\,k$	$64\,k$	$272\,k$	$1062\,k$
$2^{18} = 256\,k$	$16\,k$	$72\,k$	$281\,k$
$2^{19} = 512\,k$	$4\,k$	$19\,k$	$74\,k$
$2^{20} = 1\,M$	$1\,k$	$5\,k$	$19\,k$
$2^{21} = 2\,M$	256	1300	5120

For decimal numbers:

Radix R	max # `LIMB`s	max # digits	max # bits
10^2	$110\,G$	$220\,G$	$730\,G$
10^3	$1100\,M$	$3300\,M$	$11\,G$
10^4	$11\,M$	$44\,M$	$146\,M$
10^5	$110\,k$	$550\,k$	$1826\,k$
10^6	$1\,k$	$6,597$	$22\,k$
10^7	11	77	255

Summarizing:

[1] Of which only the 52 least significant bits are physically present, the most significant bit is implied to be always set.

- For decimal digits and precisions up to 11 million LIMBs use radix 10,000. (corresponding to more than 44 million decimal digits), for even greater precisions choose radix 1,000.
- For hexadecimal digits and precisions up to 256,000 LIMBs use radix 65,536 (corresponding to more than 1 million hexadecimal digits), for even greater precisions choose radix 4,096.

27.5 The sum-of-digits test

With high-precision calculations it is mandatory to add a sanity check to the multiplication routines. That is, false results due to loss of accuracy should (with high probability) be detected via the *sum-of-digits test*:

When computing the product $c = a \cdot b$ with radix-R numbers then compute the values ('sums of digits') $s_a = a \bmod (R-1)$ and $s_b = b \bmod (R-1)$ after the multiplication compute $s_c = c \bmod (R-1)$ and compare s_c to $s_m = s_a \cdot s_b \bmod (R-1)$. If $s_c \neq s_m$ then an error has occurred in the computation of c.

The sum-of-digits function s_a can for a radix-R, length-n number a be computed as [FXT: mult/auxil.cc]:

```
ulong
sum_of_digits(const LIMB *a, ulong n, ulong nine, ulong s)
{
    for (ulong k=0; k<n; ++k)  s += a[k];
    s %= nine;
    return s;
}
```

where the variable `nine` has to be set to $R-1$ and `s` to zero.

The computation of $s_m = s_a \cdot s_b$ is done in

```
ulong
mult_sum_of_digits(const LIMB *a, ulong an,
                   const LIMB *b, ulong bn,
                   ulong nine)
{
    ulong qsa = sum_of_digits(a, an, nine, 0);
    ulong qsb = sum_of_digits(b, bn, nine, 0);
    ulong qsm = (qsa*qsb) % nine;
    return  qsm;
}
```

The checks in multiplication routine [FXT: mult/fxtmultiply.cc] can be outlined as:

```
fxt_multiply(const LIMB *a, ulong an,
             const LIMB *b, ulong bn,
             LIMB *c, ulong cn,
             uint rx)
{
    const ulong nine = rx-1;
    ulong qsm=0, qsp=0;
    qsm = mult_sum_of_digits(a, an, b, bn, nine);

    // Multiply: c=a*b
    // If carrying through c gives an additional (leading) digit,
    // then set cy to that value,  else set cy=0.

    qsp = sum_of_digits(g, n, nine, cy);
    if ( qsm!=qsp )  { /* FAILED */ }
}
```

If we assume that a failed multiplication produces 'random' digits in c then the probability that a failed multiplication goes unnoticed equals $1/R$.

Omitting the sum-of-digits test is *not* an option: the situation that some number contains mainly 'nines' in the course of a high-precision calculation is very common. Thereby insufficient precision in the FFT-multiplication will almost certainly result in an error.

The simplicity of the sum-of-digits test that uses the modulus $R-1$ can be seen from the polynomial identity

$$\sum_k a_k\, R^k \quad \equiv \quad \sum_k a_k \qquad \bmod R-1 \tag{27.5-1}$$

One can use other moduli, like

$$\sum_k a_k\, R^k \quad \equiv \quad \sum_k (-1)^k\, a_k \qquad \bmod R+1 \tag{27.5-2}$$

Moduli R^n-1 for small n are especially convenient:

$$\sum_k a_k\, R^k \quad \equiv \quad \sum_{k\equiv 0 \bmod 2} a_k + R \sum_{k\equiv 1 \bmod 2} a_k \qquad \bmod R^2-1 \tag{27.5-3a}$$

$$\equiv \quad \sum_{k\equiv 0 \bmod 3} a_k + R \sum_{k\equiv 1 \bmod 3} a_k + R^2 \sum_{k\equiv 2 \bmod 3} a_k \qquad \bmod R^3-1 \tag{27.5-3b}$$

$$\equiv \quad \sum_{U=0}^{n-1} R^U \left(\sum_{k\equiv U \bmod n} a_k \right) \qquad \bmod R^n-1 \tag{27.5-3c}$$

One can keep the sums corresponding to the powers R^U in separate variables. Note that the probability of an unrecognized error is reduced to approximately $1/R^n$. The multiplication of the residues involves additional work proportional to n^2.

27.6 Binary exponentiation

The *binary exponentiation* (or *binary powering*) scheme is a method to compute the e-th power of a number a using about $\log(e)$ multiplications and squarings. The term 'number' can be replaced by about anything one can multiply. That includes integers, floating point numbers, polynomials, matrices, integer remainders modulo some modulus, polynomials modulo a polynomial and so on. In fact, the given algorithms work for any group: we do not need commutativity but $a^n \cdot a^m = a^{n+m}$ must hold (power-associativity).

27.6.1 Right-to-left powering

The algorithm uses the binary expansion of the exponent: let $e \geq 0$, write e the base 2 as $e = [e_j, e_{j-1}, \ldots, e_1, e_0]$, $e_i \in \{0,1\}$. Then

$$a^e \quad = \quad a^{1\cdot e_0}\; a^{2\cdot e_1}\; a^{4\cdot e_2}\; a^{8\cdot e_3}\; \cdots\; a^{2^j\, e_j} \tag{27.6-1a}$$

$$= \quad 1\,(a^1)^{e_0}\,(a^2)^{e_1}\,(a^4)^{e_2}\,\cdots\,(a^{2^j})^{e_j} \tag{27.6-1b}$$

We initialize a variable t by one, generate the powers $s_i = a^{2^i}$ by successive squarings $s_i = s_{i-1}^2 = (a^{2^{i-1}})^2$ and multiply t by s_i if e_i equals one. The following C++ code computes the e-th power of the (double precision) number a:

```
double power_r2l(double a, ulong e)
{
    double t = 1;
    if ( e )
    {
        double s = a;
        while ( 1 )
        {
            if ( e & 1 )  t *= s;
            e /= 2;
```

```
            if ( 0==e )  break;
            s *= s;
        }
    }
    return t;
}
```

A trivial optimization is to avoid the multiplication by 1 if the exponent is a power of two:

```
double power_r2l(double a, ulong e)
{
    if ( 0==e )  return 1;
    double s = a;
    while ( 0==(e&1) )
    {
        s *= s;
        e /= 2;
    }
    a = s;
    while ( 0!=(e/=2) )
    {
        s *= s;
        if ( e & 1 )  a *= s;
    }
    return  a;
}
```

The program [FXT: arith/power-r2l-demo.cc] shows the quantities that occur with the computation of $p = 2^{38}$:

```
    arg 1: 2 == a  [number to exponentiate]  default=2
    arg 2: 38 == e  [exponent]  default=38
     e = 1..11.
    0                           2                         2
    1                           4                         2
    1                          16                        64
    0                         256                        64
    0                       65536                        64
    1                  4294967296             274877906944
    p=a**e = 274877906944
```

In the right-to-left powering scheme the exponent is scanned starting from the lowest bit.

27.6.2 Left-to-right powering

The left-to-right binary powering algorithm scans the exponent starting from the highest bits. We use the facts that $a^{2k} = (a^k)^2$ and $a^{2k+1} = (a^k)^2\, a$. Implementation is simple:

```
double power_l2r(double a, ulong e)
{
    if ( 0==e )  return 1;
    double s = a;
    ulong b = highest_bit(e);
    while ( b>1 )
    {
        b >>= 1;
        s *= s;
        if ( e & b )  s *= a;
    }
    return s;
}
```

The program [FXT: arith/power-l2r-demo.cc] shows the quantities that occur with the computation of $p = 2^{38}$ when the left-to-right scan is used:

```
    arg 1: 2 == a  [number to exponentiate]  default=2
    arg 2: 38 == e  [exponent]  default=38
     e = 1..11.
    1                           2
    0                           4                         4
    0                          16                        16
    1                         256                       512
    1                      262144                    524288
    0                274877906944             274877906944
    p=a**e = 274877906944
```

All multiplications apart from the squarings happen with the unchanged value of a. This is an advantage

if a is a small (integer) value so that the multiplications are cheap. As a slightly extreme example, if one computes $7^{7^7} \approx +0.3759823526783 \cdot 10^{695975}$ to full precision, then the left-to-right powering is about 3 times faster. If a is a full-precision number (and multiplication is done via FFTs) then the FFT of a only need to be computed once. Thereby all multiplications except for the first count a squarings. This technique is called *FFT caching*.

The given powering algorithms are good enough for most applications. There are schemes that improve further. For repeated power computations, especially for very large exponents the schemes based on the so-called *addition chains* lead to better algorithms, see [155]. The so-called 'flexible window powering method' is described and analyzed in [86]. A readable survey of exponentiation methods is given in [121]. An algorithm for the efficient computation of products of powers (of different numbers) is described in [38]. These optimized exponentiation algorithms are mainly used with cryptographic applications.

Techniques for accelerating computations of factorials and binomial coefficients are described in [151].

27.6.3 Cost of binary exponentiation of full-precision numbers

e :	e (radix 2)	#S	#M	#F	#C
1:	`.....1`	0	0	0	
2:	`....1.`	1	0	2	
3:	`....11`	1	1	5	
4:	`...1..`	2	0	4	
5:	`...1.1`	2	1	7	
6:	`...11.`	2	1	7	
7:	`...111`	2	2	10	9
8:	`..1...`	3	0	6	
9:	`..1..1`	3	1	9	
10:	`..1.1.`	3	1	9	
11:	`..1.11`	3	2	12	11
12:	`..11..`	3	1	9	
13:	`..11.1`	3	2	12	11
14:	`..111.`	3	2	12	11
15:	`..1111`	3	3	15	13
16:	`.1....`	4	0	8	
17:	`.1...1`	4	1	11	
18:	`.1..1.`	4	1	11	
19:	`.1..11`	4	2	14	13
20:	`.1.1..`	4	1	11	

e :	e (radix 2)	#S	#M	#F	#C
21:	`.1.1.1`	4	2	14	13
22:	`.1.11.`	4	2	14	13
23:	`.1.111`	4	3	17	15
24:	`.11...`	4	1	11	
25:	`.11..1`	4	2	14	13
26:	`.11.1.`	4	2	14	13
27:	`.11.11`	4	3	17	15
28:	`.111..`	4	2	14	13
29:	`.111.1`	4	3	17	15
30:	`.1111.`	4	3	17	15
31:	`.11111`	4	4	20	17
32:	`1.....`	5	0	10	
33:	`1....1`	5	1	13	
34:	`1...1.`	5	1	13	
35:	`1...11`	5	2	16	15
36:	`1..1..`	5	1	13	
37:	`1..1.1`	5	2	16	15
38:	`1..11.`	5	2	16	15
39:	`1..111`	5	3	19	17
40:	`1.1...`	5	1	13	

Figure 27.6-A: Cost of binary powering of full-precision numbers for small exponents e in terms of squarings (#S), multiplications (#M) and FFTs (#F). If the left-to-right exponentiation algorithm with FFT caching needs less FFTs then the number is given under (#C).

With full-precision numbers the cost of binary powering is the same for both the left-to-right and the right-to-left algorithm. As an example, to raise x to the 26-th power, note that $e = 26 = 11010_2$ and we can write

$$x^{26} \;=\; x^{16} \cdot x^8 \cdot x^2 \;=\; (((x^2)^2)^2)^2 \cdot ((x^2)^2)^2 \cdot (x^2) \tag{27.6-2}$$

Here we need four squarings and two multiplications. In general one needs $\lfloor \log_2 e \rfloor$ squarings and $h(e)-1$ multiplications were $h(e)$ is the number of set bits in the binary expansion of e. Figure 27.6-A lists the cost of the exponentiation for small exponents e in terms of squarings and multiplications and, assuming a squaring costs two FFTs and multiplication three, in terms of FFTs. The table was created with the program [FXT: arith/power-costs-demo.cc].

Chapter 28

Root extraction

We describe methods to compute the inverse, square root, and higher roots of a given number. The computation of any of these costs just the equivalent of a few full-precision multiplications.

28.1 Division, square root and cube root

28.1.1 Inverse and division

The ordinary division algorithm is far too expensive for numbers of extreme precision. Instead one replaces the division $\frac{a}{d}$ by the multiplication of a with the inverse of d. The inverse of d is computed by finding a starting approximation $x_0 \approx \frac{1}{d}$ and then iterating

$$x_{k+1} \quad = \quad x_k + x_k\,(1 - d\,x_k) \tag{28.1-1}$$

until the desired precision is reached. The convergence is quadratic (second order), which means that the number of correct digits is doubled with each step: if $x_k = \frac{1}{d}(1+e)$ then $x_{k+1} = \frac{1}{d}\,\left(1-e^2\right)$.

Moreover, each step only requires computations with twice the number of digits that were correct at its beginning. Still better: the multiplication $x_k(\dots)$ needs only to be done with half of the current precision as it computes the correcting digits (which alter only the less significant half of the digits). Thus, at each step we have 1.5 multiplications of the current precision: one full precision multiplication for $d\,x_k$ plus a half precision multiplication for $x_k(\dots)$. The total work amounts to $1.5 + 1.5/2 + 1.5/4 + \dots = 1.5 \cdot \sum_{n=0}^{N} \frac{1}{2^n}$ which is less than 3 full precision multiplications. The cost of a multiplication is set to $\sim N$ for the estimates made here, this gives a realistic picture for large N. Together with the final multiplication a division costs as much as 4 multiplications.

The numerical example given in figure 28.1-A shows the first steps of the computation of an inverse starting from a two-digit initial approximation.

The achieved precision can be determined by the absolute value of $(1-d\,x_k)$. In hfloat, when the achieved precision is below a certain limit a third order correction is used to assure maximum precision at the last step:

$$x_{k+1} \quad = \quad x_k + x_k\,(1 - d\,x_k) + x_k\,(1 - d\,x_k)^2 \tag{28.1-2}$$

One should in general *not* use algebraically equivalent forms like $x_{k+1} = 2\,x_k - d\,x_k^2$ (for the second order iteration) because computationally there is a difference: cancellation can occur and the information on the achieved precision is not found easily.

$$
\begin{aligned}
d &:= 3.1415926 \\
x_0 &:= 0.31 \qquad \text{[initial 2-digit approximation for } 1/d\text{]} \\
\\
d \cdot x_0 &:= 3.141 \cdot 0.3100 = 0.9737 \\
y_0 &:= 1.000 - d \cdot x_0 = 0.02629 \\
x_0 \cdot y_0 &:= 0.3100 \cdot 0.02629 = 0.0081(49) \\
x_1 &:= x_0 + x_0 \cdot y_0 = 0.3100 + 0.0081 = 0.3181 \\
\\
d \cdot x_1 &:= 3.1415926 \cdot 0.31810000 = 0.9993406 \\
y_1 &:= 1.0000000 - d \cdot x_1 = 0.0006594 \\
x_1 \cdot y_1 &:= 0.31810000 \cdot 0.0006594 = 0.0002097(5500) \\
x_2 &:= x_1 + x_1 \cdot y_1 = 0.31810000 + 0.0002097 = 0.31830975 \\
\\
d \cdot x_2 &:= 3.1415926 \cdot 0.31830975 = 0.99999955 \\
y_2 &:= 1.0000000 - d \cdot x_2 = 0.00000014 \\
x_2 \cdot y_2 &:= 0.31830975 \cdot 0.00000014 = 0.000000044 \\
x_3 &:= x_2 + x_2 \cdot y_2 = 0.31830975 + 0.000000044 = 0.31830979399
\end{aligned}
$$

Figure 28.1-A: First steps of the computation of the inverse of π.

28.1.2 Inverse square root

Computation of inverse square roots can be done using a similar scheme: find a starting approximation $x_0 \approx \frac{1}{\sqrt{d}}$ then iterate

$$x_{k+1} \quad = \quad x_k + x_k \frac{(1 - d\,x_k^2)}{2} \tag{28.1-3}$$

until the desired precision is reached. Convergence is again second order: if $x_k = \frac{1}{\sqrt{d}}(1 + e)$ then

$$x_{k+1} \quad = \quad \frac{1}{\sqrt{d}} \left(1 - \frac{3}{2}e^2 - \frac{1}{2}e^3\right) \tag{28.1-4}$$

When the achieved precision is below a certain limit a third order correction should be applied:

$$x_{k+1} \quad = \quad x_k + x_k \frac{(1 - d\,x_k^2)}{2} + x_k \frac{3\,(1 - d\,x_k^2)^2}{8} \tag{28.1-5}$$

To compute the square root first compute $\frac{1}{\sqrt{d}}$ then a final multiply with d gives $\sqrt{d}$.

Similar considerations as above (with squaring considered as expensive as multiplication[1]) give an operation count of 4 multiplications for computing $\frac{1}{\sqrt{d}}$ and 5 for $\sqrt{d}$. Note that this algorithm is considerably better than iterating $x_{k+1} := \frac{1}{2}\,(x_k + \frac{d}{x_k})$ because no long divisions are involved.

A unified routine that implements the computation of the inverse a-th roots is given in [hfloat: src/hf/itiroot.cc]. The general form of the divisionless iteration for the a-th root of d is, up to third

[1] Indeed it costs about $\frac{2}{3}$ of a multiplication.

order:

$$x_{k+1} \quad = \quad x_k \left(1 + \frac{(1 - d\,x_k^a)}{a} + \frac{(1+a)\,(1 - d\,x_k^a)^2}{2\,a^2}\right) \tag{28.1-6}$$

The initial approximation is obtained using ordinary floating point numbers (type `double`) with special precautions to avoid overflow with exponents that cannot be represented with `double`s. Third order corrections are made whenever the achieved precision falls below a certain limit.

28.1.3 Cube root extraction

We use the relation $d^{1/3} = d\,(d^2)^{-1/3}$. That is, we compute the inverse third root of d^2 using the iteration

$$x_{k+1} \quad = \quad x_k + x_k \frac{(1 - d^2\,x_k^3)}{3} \tag{28.1-7}$$

and finally multiply with d. Convergence is second order: if $x_k = \frac{1}{\sqrt[3]{d}}(1+e)$ then

$$x_{k+1} \quad = \quad \frac{1}{\sqrt[3]{d}} \left(1 - 2e^2 - \frac{4}{3}e^3 - \frac{1}{3}e^4\right) \tag{28.1-8}$$

28.1.4 Improved iteration for the square root

Actually, the 'simple' version of the square root iteration ($x_{k+1} := \frac{1}{2}\,(x_k + \frac{d}{x_k})$) can be used for practical purposes when rewritten as a *coupled iteration* for both $\sqrt{d}$ and its inverse. Using for $\sqrt{d}$ the iteration

$$x_{k+1} \quad = \quad x_k - \frac{(x_k^2 - d)}{2\,x_k} \tag{28.1-9}$$

$$\quad = \quad x_k - v_{k+1}\,\frac{(x_k^2 - d)}{2} \quad \text{where} \quad v \approx 1/x \tag{28.1-10}$$

and for the auxiliary $v \approx 1/\sqrt{d}$ the iteration

$$v_{k+1} = v_k + v_k\,(1 - x_k\,v_k) \tag{28.1-11}$$

where one starts with approximations

$$x_0 \quad \approx \quad \sqrt{d} \tag{28.1-12}$$

$$v_0 \quad \approx \quad 1/x_0 \tag{28.1-13}$$

and the v-iteration step precedes that for x. When carefully implemented this method turns out to be significantly more efficient than the computation via the inverse root. An implementation is given in [hfloat: src/hf/itsqrt.cc]. The idea is due to Schönhage.

28.1.5 A different view on the iterations

Let p be a prime and assume you know the inverse x_0 of a given number d modulo p. With (the iteration for the inverse, relation 28.1-1 on page 541) $\Phi(x) := x\,(1 + (1 - d\,x))$ the number $x_1 := \Phi(x_0)$ is the inverse of d modulo p^2. Modulo p^2 we know that $x_0\,d \equiv (1 + k\,p)$ so we can write $x_0 \equiv 1/d\,(1 + k\,p)$, thereby

$$\Phi(x_0) \quad = \quad \Phi\left(\frac{1}{d}\,(1 + k\,p)\right) \;=\; \frac{1}{d}\,\left(1 - k\,p^2\right) \;\equiv\; \frac{1}{d} \bmod p^2 \tag{28.1-14}$$

The very same computation (with $x_1 = 1/d\,(1 + j\,p^2)$) shows that for $x_2 := \Phi(x_1)$ one has $x_2 \equiv 1/d \bmod p^4$. Each application of Φ doubles the exponent of the modulus.

The equivalent scheme works for root extraction. An example for the inverse square root: with $p = 17$ and $x_0 = 3$ we have $x_0^2\,2 \equiv 1$ ($x_0 \equiv 1/\sqrt{d} \bmod p$ where $d = 2$). Now use the iteration $\Phi(x) := x\,(1+(1-d\,x)/2)$ to compute $x_1 = \Phi(x_0)$ to obtain $x_1 = -45/2 \equiv 122 \bmod p$ and observe that $x_1^2\,d \equiv 1 \bmod p^2$. Compute $x_2 = \Phi(x_1)$ to obtain $x_2 = -1815665 \equiv 21797 \bmod p^2$ and check that $x_2^2\,d \equiv 1 \bmod p^4$.

We will not go into the details (of the theory of p-adic numbers) but note that pari/gp can work with them:

```
? 1/sqrt(2+O(17^5))
  3 + 7*17 + 7*17^2 + 4*17^3 + 11*17^4 + O(17^5)
\\ Note that 21797 = 3 + 7*17 + 7*17^2 + 4*17^3 + 11*17^4
\\       and   122 = 3 + 7*17
```

Section 1.22 on page 55 describes the case $p = 2$. The computation of a square root modulo p^x, given a square root modulo p, is described in section 37.9 on page 751.

28.2 Root extraction for rationals

We give expression for the extraction of the a-th root of a rational quantity.

28.2.1 Extraction of the square root

A general formula for an k-th order ($k \geq 2$) iteration for $\sqrt{d}$ is

$$\Phi_k(x) \;=\; \sqrt{d}\,\frac{\left(x+\sqrt{d}\right)^k + \left(x-\sqrt{d}\right)^k}{\left(x+\sqrt{d}\right)^k - \left(x-\sqrt{d}\right)^k} = \sqrt{d}\,\frac{\left(p+q\sqrt{d}\right)^k + \left(p-q\sqrt{d}\right)^k}{\left(p+q\sqrt{d}\right)^k - \left(p-q\sqrt{d}\right)^k} \tag{28.2-1}$$

where $x = p/q$. All $\sqrt{d}$ vanish when expanded:

$$\Phi_2(x) \;=\; \frac{x^2+d}{2\,x} \;=\; \frac{p^2+d\,q^2}{2\,p\,q} \tag{28.2-2a}$$

$$\Phi_3(x) \;=\; x\,\frac{x^2+3\,d}{3\,x^2+d} \;=\; \frac{p}{q}\,\frac{p^2+3\,d\,q^2}{3\,p^2+d\,q^2} \tag{28.2-2b}$$

$$\Phi_4(x) \;=\; \frac{x^4+6\,d\,x^2+d^2}{4\,x^3+4\,d\,x} \;=\; \frac{p^4+6\,d\,p^2\,q^2+d^2\,q^4}{4\,p^3\,q+4\,d\,p\,q^3} \tag{28.2-2c}$$

$$\Phi_5(x) \;=\; x\,\frac{x^4+10\,d\,x^2+5\,d^2}{5\,x^4+10\,d\,x^2+d^2} \;=\; \frac{p}{q}\,\frac{p^4+10\,d\,p^2\,q^2+5\,d^2\,q^4}{5\,p^4+10\,d\,p^2\,q^2+d^2\,q^4} \tag{28.2-2d}$$

$$\Phi_k(x) \;=\; x\,\frac{\sum_{j=0}^{\lfloor k/2\rfloor}\binom{k}{2\,j}\,x^{k-2j}\,d^j}{\sum_{j=0}^{\lfloor k/2\rfloor}\binom{k}{2\,j+1}\,x^{k-2j-1}\,d^j} \tag{28.2-2e}$$

The denominators and numerators of Φ_k are terms of the second order recurrence

$$a_k \;=\; 2\,x\,a_{k-1} - \left(x^2-d\right)\,a_{k-2} \tag{28.2-3}$$

with initial terms $a_0 = 1$, $a_1 = x$ for the denominators, and $a_0 = 0$, $a_1 = 1$ for the numerators (that is, $\Phi_0 = 0/1$, $\Phi_1 = x/1$). There is a nice expression for the error behavior of the k-th order iteration:

$$\Phi_k\left(\sqrt{d}\cdot\frac{1+e}{1-e}\right) \;=\; \sqrt{d}\cdot\frac{1+e^k}{1-e^k} \tag{28.2-4}$$

The following composition law holds:

$$\Phi_m(\Phi_n(x)) \;=\; \Phi_{m\,n}(x) \tag{28.2-5}$$

28.2.2 Extraction of the a-th root

A second order iteration for $\sqrt[a]{z}$ is given by

$$\Phi_2(x) \quad = \quad x + \frac{d - x^a}{a\,x^{a-1}} = \frac{(a-1)\,x^a + d}{a\,x^{a-1}} = \frac{1}{a}\left((a-1)\,x + \frac{d}{x^{a-1}}\right) \tag{28.2-6}$$

A third order iteration for $\sqrt[a]{d}$ is

$$\Phi_3(x) \quad = \quad x \cdot \frac{\alpha\,x^a + \beta\,d}{\beta\,x^a + \alpha\,d} \;=\; \frac{p}{q} \cdot \frac{\alpha\,p^a + \beta\,q^a d}{\beta\,p^a + \alpha\,q^a d} \tag{28.2-7}$$

where $x = p/q$, $\alpha = a - 1$ and $\beta = a + 1$.

28.2.3 More iterations via Padé approximants *

The iterations can also be obtained using Padé-approximants. Let $P_{[i,j]}(z)$ be the Padé-expansion of $\sqrt{z}$ around $z = 1$ of order $[i, j]$. An iteration of order $i+j+1$ is given by $x\,P_{[i,j]}(\frac{d}{x^2})$. Different combinations of i and j result in alternative iterations:

$$[i,j] \quad \mapsto \quad x\,P_{[i,j]}\left(\frac{d}{x^2}\right) \tag{28.2-8a}$$

$$[1,0] \quad \mapsto \quad \frac{x^2 + d}{2x} \tag{28.2-8b}$$

$$[0,1] \quad \mapsto \quad \frac{2x^3}{3x^2 - d} \tag{28.2-8c}$$

$$[1,1] \quad \mapsto \quad x\,\frac{x^2 + 3d}{3x^2 + d} \tag{28.2-8d}$$

$$[2,0] \quad \mapsto \quad \frac{3x^4 + 6dx^2 - 3d^2}{8x^3} \tag{28.2-8e}$$

$$[0,2] \quad \mapsto \quad \frac{8x^5}{15x^4 - 10dx^2 + 3d^2} \tag{28.2-8f}$$

Still other forms are obtained by using $\frac{d}{x}\,P_{[i,j]}(\frac{x^2}{d})$:

$$[i,j] \quad \mapsto \quad \frac{d}{x}\,P_{[i,j]}\left(\frac{x^2}{d}\right) \tag{28.2-9a}$$

$$[1,0] \quad \mapsto \quad \frac{x^2 + d}{2x} \tag{28.2-9b}$$

$$[0,1] \quad \mapsto \quad \frac{2d^2}{3dx - x^3} \tag{28.2-9c}$$

$$[1,1] \quad \mapsto \quad \frac{d\,(d + 3x^3)}{x\,(3d + x^2)} \tag{28.2-9d}$$

$$[2,0] \quad \mapsto \quad \frac{-x^4 + 6dx^2 + 3d^2}{8xd} \tag{28.2-9e}$$

$$[0,2] \quad \mapsto \quad \frac{8d^3}{3x^4 - 10dx^2 + 15d^2} \tag{28.2-9f}$$

In section 29.5 on page 569 the Padé idea is pursued for arbitrary functions f.

28.2.4 A product involving $\Phi_k(y)$ for $d = 1$ *

A product expression involving the rational iterations of order 2^n for $d = 1$ is given by

$$y \left(\frac{y^2+1}{2y}\right)^{1/2} \left(\frac{y^4+6y^2+1}{4y^3+4y}\right)^{1/4} \ldots \left(\Phi_{2^k}(y)\right)^{1/2^k} \ldots = \left(\frac{y+1}{2}\right)^2 \tag{28.2-10}$$

where $\Phi_1(y) = y$. The relation can be deduced from the following relation (given by R.W.Gosper) by setting $x = \operatorname{arccoth}(y)$:

$$\coth(x)\, \coth(2x)^{1/2}\, \coth(4x)^{1/4} \ldots \coth(2^k x)^{1/2^k} \ldots = \frac{\exp(2x)}{4\, \sinh^2(x)} \tag{28.2-11}$$

An equivalent relation is obtained by setting $y = (1+e)/(1-e)$ and using relation 28.2-4 on page 544:

$$\prod_{k=0}^{\infty} \left(\frac{1+e^{2^k}}{1-e^{2^k}}\right)^{1/2^k} = \frac{1}{(1-e)^2} \tag{28.2-12}$$

Compare with relation 36.1-14a on page 693. More generally, one has

$$\prod_{k=0}^{\infty} \left(\frac{\sum_{j=0}^{n-1} e^{j\cdot n^k}}{1-e^{n^k}}\right)^{1/2^k} = \frac{1}{(1-e)^2} \tag{28.2-13}$$

28.3 Divisionless iterations for the inverse a-th root

There is a nice general formula that gives iterations with arbitrary order of convergence for $1/\sqrt[a]{d} = d^{-1/a}$ that involve no long division.

One uses the identity

$$d^{-1/a} = x\, (1-(1-x^a\, d))^{-1/a} \tag{28.3-1}$$

$$= x\,(1-y)^{-1/a} \quad \text{where} \quad y := (1-x^a\, d) \tag{28.3-2}$$

Taylor expansion gives

$$d^{-1/a} = x \sum_{k=0}^{\infty} (1/a)^{\overline{k}}\, y^k \tag{28.3-3}$$

where $z^{\overline{k}} := z\,(z+1)\,(z+2)\,\ldots\,(z+k-1)$ (and $z^{\overline{0}} := 1$, $z^{\overline{k}}$ is called the rising factorial power). Written out:

$$d^{-1/a} = x\, \frac{1}{\sqrt[a]{1-y}} = x \left(1 + \frac{y}{a} + \frac{(1+a)\, y^2}{2\, a^2} + \frac{(1+a)(1+2a)\, y^3}{6\, a^3} + \right. \tag{28.3-4}$$
$$\left. + \frac{(1+a)(1+2a)(1+3a)\, y^4}{24\, a^4} + \cdots + \frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\, a^n}\, y^n + \ldots \right)$$

A n-th order iteration for $d^{-1/a}$ is obtained by truncating the above series after the $(n-1)$-th term:

$$\Phi_n(x) := x \sum_{k=0}^{n-1} (1/a)^{\overline{k}}\, y^k \tag{28.3-5}$$

$$x_{k+1} = \Phi_n(x_k) \tag{28.3-6}$$

Convergence is n-th order:

$$\Phi_n(d^{-1/a}(1+e)) = d^{-1/a}(1+O(e^n)) \tag{28.3-7}$$

For example, the second order iteration is

$$\Phi_2(x) \quad := \quad x + x\frac{(1-d\,x^a)}{a} \tag{28.3-8}$$

Convergence is indeed quadratic: if $x = \frac{1}{\sqrt[a]{d}}\,(1+e)$ then

$$\Phi_2(x) \quad = \quad \frac{1}{\sqrt[a]{d}}\left((1+e)\left[(1+e)^a - (a+1)\right]\right) \tag{28.3-9}$$

$$= \quad \frac{1}{\sqrt[a]{d}}\left(1 - \frac{a+1}{2}e^2 + O(e^3)\right) \tag{28.3-10}$$

28.3.1 Iterations for the inverse

Set $a = 1$, $y = 1 - d\,x$ to compute the inverse of d.

$$\frac{1}{d} \quad = \quad x\,\frac{1}{1-y} \tag{28.3-11a}$$

$$\Phi_k(x) \quad = \quad x\,\left(1 + y + y^2 + y^3 + y^4 + \cdots + y^{k-1}\right) \tag{28.3-11b}$$

$\Phi_2(x) = x\,(1+y)$ is the second order iteration 28.1-1 on page 541.

Composition is particularly simple with the iterations for the inverse:

$$\Phi_{n\,m} \quad = \quad \Phi_n(\Phi_m) \tag{28.3-12}$$

There are simple closed forms for this iteration:

$$\Phi_k \quad = \quad \frac{1-y^k}{d} = x\,\frac{1-y^k}{1-y} \tag{28.3-13a}$$

$$\Phi_\infty \quad = \quad 1 + x + x^2 + x^3 + x^4 + \ldots \tag{28.3-13b}$$

$$= \quad x\,(1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots \tag{28.3-13c}$$

$$= \quad x\,(1+y+y^2)\,(1+y^3+y^6)\,(1+y^9+y^{18})\,\ldots \tag{28.3-13d}$$

The expression for the convergence of the k-th order iteration is

$$\Phi_k\left(\frac{1}{d}\,(1+e)\right) \quad = \quad \frac{1}{d}\,\left(1 - (-e)^k\right) \tag{28.3-14}$$

The iteration converges if one has $|e| < 1$ for the start value $x_0 = \frac{1}{d}\,(1+e)$. That is, the basin of attraction is the open disc of radius $r = 1/d$ around the point $1/d$, independent of the order k. For other iterations, the basin of attraction usually has a fractal boundary and further depends on the order.

28.3.2 Iterations for the inverse square root

Set $a = 2$, $y = 1 - d\,x^2$ to compute the inverse square root of d.

$$\frac{1}{\sqrt{d}} \quad = \quad x\,\frac{1}{\sqrt{1-y}} \tag{28.3-15a}$$

$$= \quad x\,\left(1 + \frac{y}{2} + \frac{3\,y^2}{8} + \frac{5\,y^3}{16} + \frac{35\,y^4}{128} + \cdots + \frac{\binom{2k}{k}\,y^k}{4^k} + \ldots\right) \tag{28.3-15b}$$

$$\Phi_{k+1}(x) \quad = \quad x\,\left(1 + \frac{y}{2} + \frac{3\,y^2}{8} + \ldots + \frac{\binom{2k}{k}\,y^k}{4^k}\right) \tag{28.3-15c}$$

$\Phi_2(x) = x\,(1 + y/2)$ is the second order iteration 28.1-3 on page 542.

28.3.3 Computation of the a-th root

```
? default(realprecision,55);
? n=5;d=3;
? f=x^n-d
   x^5 - 3
? phi(x)=(x+(x/n-x^(n+1)/(n*d)))
? phi(x)
   -1/15*x^6 + 6/5*x
? y=real(polroots(f)[1])
   1.2457309396155173259666803366403050809393099993068779811
? y*=(1.01); \\ <--= initial approximation within 1%
? for(k=0,7,t=phi(y);print(k,":  ",y);y=t;);
0:  1.2581882490116724992263471400067081317487030929994676 09
1:  1.2453521998882091612922815042363613525213873439226820 49
2:  1.2457305943106651323381267600848321408508336218806673 03
3:  1.2457309396152301803405533431627934257835002801383562 72
4:  1.2457309396155173259666801380758928589406244031248749 62
5:  1.2457309396155173259666803366403050809393099930686848 60
6:  1.2457309396155173259666803366403050809393099930687798 11
7:  1.2457309396155173259666803366403050809393099930687798 11
? y^n
   3.0000000000000000000000000000000000000000000000000000000
```

Figure 28.3-A: Quantities occurring in the iterative computation of $\sqrt[5]{3}$.

The following (second order) iteration computes $\sqrt[a]{d}$ directly:

$$\Phi(x) \;=\; x + \frac{1}{a}\left(x - \frac{x^{a+1}}{d}\right) \tag{28.3-16}$$

Figure 28.3-A shows the quantities occurring in the iterative computation of $\sqrt[5]{3}$. The iteration is involves no long division for small (rational) d.

To compute the a-root of a full-precision number d one can use the iteration for the inverse root and invert afterwards. Another possibility is to compute the inverse a-th root of d^{a-1} and multiply with d afterwards:

$$\left[(d)^{a-1}\right]^{-1/a} d \;=\; d^{(1-a)/a}\, d \;=\; d^{1/a} \tag{28.3-17}$$

If a is small the cost is lower than with the final iteration for the inverse (which costs about 3 Multiplications or 9 FFTs). The powering-method is not more expensive than inversion if the rightmost column in figure 27.6-A on page 539 for $e = a - 1$ is less or equal 6. In case the iteration for the inverse involves a loss of precision the method might be preferred also if its cost is higher.

28.3.4 Error expressions for inverse square root iterations *

An expression for the error behavior of the n-th order iteration similar to relation 28.2-4 on page 544 is

$$F_n \;:=\; \Phi_n(d^{-1/2}\frac{1+e}{1-e})/d^{-1/2} \tag{28.3-18a}$$

$$\;=\; \frac{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k - \sum_{k=n+1}^{2n+1}\binom{2n-1}{k}(-e)^k}{(1-e)^{2n-1}} \tag{28.3-18b}$$

Now define $c := \frac{F_n - 1}{F_n + 1}$, then

$$F_n \;=\; \frac{1+c}{1-c} \quad \text{where} \quad c \;=\; e^n\,\frac{\sum_{k=0}^{n}\binom{2n-1}{n-k}(-e)^k}{\sum_{k=0}^{n}\binom{2n-1}{k}(-e)^k} \tag{28.3-18c}$$

For example, with $n = 2$ we obtain

$$F_2 := \Phi_2(d^{-1/2}\frac{1+e}{1-e}) \,/\, d^{-1/2} \tag{28.3-19a}$$
$$= \frac{1-3\,e-3\,e^2+e^3}{1-3\,e+3\,e^2-e^3} \tag{28.3-19b}$$
$$= 1-6\,e^2-16\,e^3-30\,e^4-48\,e^5-70\,e^6-\ldots \tag{28.3-19c}$$
$$= -1+\frac{6}{(e-1)^2}+\frac{4}{(e-1)^3} \tag{28.3-19d}$$
$$F_2 = \frac{1+c}{1-c} \quad \text{where} \quad c = e^2\,\frac{e-3}{1-3\,e} \tag{28.3-19e}$$

The coefficients of the Taylor expansion of F_n in e are always integers.

For $n = 4$:

$$F_4 := \Phi_4(d^{-1/2}\frac{1+e}{1-e}) \,/\, d^{-1/2} \tag{28.3-20a}$$
$$= \frac{1-7\,e+21\,e^2-35\,e^3-35\,e^4+21\,e^5-7\,e^6+e^7}{1-7\,e+21\,e^2-35\,e^3+35\,e^4-21\,e^5+7\,e^6-e^7} \tag{28.3-20b}$$
$$= 1-70\,e^4-448\,e^5-1680\,e^6-4800\,e^7-11550\,e^8-\ldots \tag{28.3-20c}$$
$$= -1+\frac{70}{(e-1)^4}+\frac{168}{(e-1)^5}+\frac{140}{(e-1)^6}+\frac{40}{(e-1)^7} \tag{28.3-20d}$$
$$F_4 = \frac{1+c}{1-c} \quad \text{where} \quad c = e^4\,\frac{e^3-7\,e^2+21\,e-35}{1-7\,e+21\,e^2-35\,e^3} \tag{28.3-20e}$$

Two curious formulas related to the error behavior of Φ_2 are

$$\Phi_2\left(\frac{1}{\sqrt{d}}\left[e+\frac{1}{e}\right]\right) = \frac{1}{\sqrt{d}}\left[-\frac{1}{2}\cdot\left(e^3+\frac{1}{e^3}\right)\right] \tag{28.3-21}$$
$$\Phi_2\left(\frac{1}{\sqrt{d}}\left[e-\frac{2}{3}\,\frac{1}{e}\right]\right) = \frac{1}{\sqrt{d}}\left[+\frac{1}{2}\cdot\left(e^3-\frac{2}{3}\,\frac{1}{e^3}\right)\right] \tag{28.3-22}$$

28.4 Initial approximations for iterations

With the iterative schemes one alway needs an initial approximation for the root that is to be computed. Assume we want to compute $f(d)$, for example, $f(d) = \sqrt{d}$ or $f(d) = \exp(d)$. One can convert the high precision number d to a machine floating point number and use the FPU to compute an initial approximation. However, when d cannot be represented with a machine float, the method fails. The method will also fail if the result causes an overflow, which is likely to happen with $f(d) = \exp(d)$.

28.4.1 Inverse roots

With $f(d) = d^{1/a}$ one can use the following technique. Write d in the form

$$d = M \cdot R^X \tag{28.4-1}$$

were M is the mantissa, R the radix and X the exponent. We have $0 \le M < 1$ and $X \in \mathbb{Z}$. Now use

$$d^{1/a} = M^{1/a} \cdot R^{X/a} = M^{1/a} \cdot R^{Y/a} \cdot R^Z \tag{28.4-2}$$

where $Z = \lfloor X/a \rfloor$ and $Y = X - a \cdot Z$ (so $X = a \cdot Z + Y$).

The algorithm computes the three quantities in relation 28.4-2 separately and finally computes the product as result. A C++ implementation is given in [hfloat: src/hf/itiroot.cc]:

```
void
approx_invpow(const hfloat &d, hfloat &c, long a)
{
    double dd;
    dt_mantissa_to_double(*(d.data()), dd);
    dd = pow(dd, 1.0/(double)a);  // M^(1/a)

    long  Z = d.exp() / a;    // Z = X / a
    long  Y = d.exp() - a*Z;  // Y = X % a

    double tt = pow((double)d.radix(),(double)Y/a);  // R^(Y/a)
    dd *= tt;         // M^(1/a) * R^(Y/a)

    d2hfloat(dd, c);  // c =  M^(1/a) * R^(Y/a)
    c.exp( c.exp()+Z );   // c *= R^(Z)
}
```

One can also subtract $a \cdot Z$ from the exponent before the iteration and add Z to the exponent afterwards: $\left(R^{X-a\,Z}\right)^{1/a} = R^{Y/a} = R^{X/a}/R^{Z}$. In that case the initial approximation can be computed via the straight forward approach.

28.4.2 Exponential function

With $f(d) = \exp(d)$ write

$$\exp(d) \quad = \quad M \cdot R^{X}$$

Then use $X = \lfloor d/\log(R) \rfloor$ and $M = \exp(d - X \cdot \log R)$. Note that d must fit into a machine float which is not a real restriction as the $\exp(d)$ will not fit into a `hfloat` type already with smaller values as the exponent of the result would overflow.

A C++ implementation is given in [hfloat: src/tz/itexp.cc]:

```
void
approx_exp(const hfloat &d, hfloat &c)
{
    double dd;
    hfloat2d(d,dd);

    double lr = log( hfloat::radix() );
    double X = floor( dd/lr );
    double M = exp( dd-X*lr );

    d2hfloat(M,c);
    c.exp( c.exp()+(long)X );
}
```

The iteration for computation of the exponential function is given in section 31.2 on page 603.

28.5 Some applications of the matrix square root

We give applications of the iteration for the (inverse) square root to compute re-orthogonalized matrices, the polar decomposition, the sign decomposition, and the pseudo-inverse of a matrix.

28.5.1 Re-orthogonalization

A task from graphics applications: a rotation matrix A that deviates from being orthogonal[2] shall be transformed to the closest orthogonal matrix E. One has (see [205]):

$$E \quad = \quad A\,(A^T A)^{-\frac{1}{2}} \tag{28.5-1}$$

[2]Typically due to cumulative errors resulting from many multiplications with rotation matrices.

With the division-free iteration for the inverse square root

$$\Phi(x) \quad = \quad x\left(1+\frac{1}{2}\left(1-dx^2\right)+\frac{3}{8}\left(1-dx^2\right)^2+\frac{5}{16}\left(1-dx^2\right)^3+\ldots\right) \tag{28.5-2}$$

the given task is pretty easy: as A^TA is close to unity (the identity matrix) we can use the (second order) iteration with $d = A^TA$ and $x = 1$

$$(A^TA)^{-\frac{1}{2}} \quad \approx \quad \left(1+\frac{1-A^TA}{2}\right) \tag{28.5-3}$$

and multiply by A to get a 'closer-to-orthogonal' matrix A_+:

$$A_+ \quad = \quad A\left(1+\frac{1-A^TA}{2}\right) \quad \approx E \tag{28.5-4}$$

The step can be repeated with A_+ (or higher orders can be used) if necessary. Note the identical equation would be obtained when trying to compute the inverse square root of 1:

$$x_+ \quad = \quad x\left(1+\frac{1-x^2}{2}\right) \quad \to 1 \tag{28.5-5}$$

It is instructive to write things down in the *singular value decomposition* (SVD) representation

$$A \quad = \quad U\,\Omega\,V^T \tag{28.5-6}$$

where U and V are orthogonal and Ω is a diagonal matrix with non-negative entries, see [251]. We note that the SVD is *not* unique, using the 1×1 matrix $[-2] = [-1]\,[2]\,[1] = [1]\,[2]\,[-1]$. The SVD is a decomposition of the action of the matrix as: rotation – element-wise stretching – rotation. Now

$$A^TA \quad = \quad \left(V\Omega U^T\right)\,\left(U\Omega V^T\right) = V\Omega^2V^T \tag{28.5-7}$$

Thereby (using the fact that $(V\Omega V^T)^n = V\Omega^nV^T$)

$$(A^TA)^{-\frac{1}{2}} \quad = \quad \left(\left(V\Omega U^T\right)\left(U\Omega V^T\right)\right)^{-\frac{1}{2}} = \left(V\Omega^2V^T\right)^{-\frac{1}{2}} = V\Omega^{-1}V^T \tag{28.5-8}$$

and we have

$$A\,(A^TA)^{-\frac{1}{2}} \quad = \quad \left(U\Omega V^T\right)\,\left(V\Omega^{-1}V^T\right) = U\,V^T \tag{28.5-9}$$

that is, the 'stretching part' was removed.

A numerical example: Let

$$A \quad = \quad \begin{bmatrix} +1.0000000 & +1.0000000 & +0.7500000 \\ -0.5000000 & +1.5000000 & +1.0000000 \\ +0.7500000 & +0.5000000 & -1.0000000 \end{bmatrix} \tag{28.5-10}$$

then

$$E \quad = \quad \begin{bmatrix} +0.803114165 & +0.291073143 & +0.519888513 \\ -0.486897253 & +0.823533541 & +0.291073143 \\ +0.343422053 & +0.486897252 & -0.803114166 \end{bmatrix} \tag{28.5-11}$$

and $E\,E^T = 1$.

28.5.2 Polar decomposition

The so-called *polar decomposition* of a matrix A is a representation of the form

$$A \;=\; E\,R \tag{28.5-12}$$

where the matrix E is orthogonal and $R = R^T$. It is an analogue to the representation of a complex number $z \in \mathbb{C}$ as $z = e^{i\,\phi}\,r$ (identify $R \sim r$ and $E \sim e^{i\,\phi}$).

The polar decomposition can be defined by

$$A \;=\; E\,R \;:=\; \Big(A(A^TA)^{-1/2}\Big)\,\Big((A^TA)^{1/2}\Big) \tag{28.5-13}$$

where R is the (unique, positive semidefinite) square root $R = (A^TA)^{1/2}$ and $E = A(A^TA)^{-1/2}$.

The matrix E is computed as before:

$$E \;=\; A\cdot\left(1+\frac{1-A^TA}{2}\right)\cdot\left(1+\frac{1-A_+^TA_+}{2}\right)\cdot\ldots \tag{28.5-14}$$

The matrix R equals $E^{-1}A = E^TA$, that is

$$\begin{aligned} A &= E\,R \;=\; E\,\big(E^TA\big) && (28.5\text{-}15)\\ &= U\,V^T\,\big(V\,\Omega\,V^T\big) && (28.5\text{-}16)\end{aligned}$$

Compute the polar decomposition as

$$\begin{aligned} E_0 &= A && (28.5\text{-}17a)\\ Y_k &= \left(1+\frac{1-E_k^T\,E_k}{2}\right) && (28.5\text{-}17b)\\ E_{k+1} &= E_k\,Y_k \quad \to E && (28.5\text{-}17c)\\ R_{k+1} &= E_{k+1}^TA \quad \to R && (28.5\text{-}17d)\\ E_{k+1}\,R_{k+1} &\to A && (28.5\text{-}17e)\end{aligned}$$

Higher orders can be added in the computation of Y_k. If you prefer $z = r\,e^{i\,\phi}$ over $e^{i\,\phi}\,r$ then iterate as above but set $R' = A\,E^T$ so that

$$\begin{aligned} A &= R'\,E \;=\; \big(AE^T\big)\,E && (28.5\text{-}18)\\ &= \big(U\,\Omega\,U^T\big)\,U\,V^T && (28.5\text{-}19)\end{aligned}$$

Numerical example: Let

$$A \;=\; \begin{bmatrix} +1.00000 & +1.00000 & +0.75000\\ -0.50000 & +1.50000 & +1.00000\\ +0.75000 & +0.50000 & -1.00000\end{bmatrix} \tag{28.5-20}$$

then

$$\begin{aligned} A &= E\,R && (28.5\text{-}21a)\\ &= \begin{bmatrix} +0.80311 & +0.29107 & +0.51988\\ -0.48689 & +0.82353 & +0.29107\\ +0.34342 & +0.48689 & -0.80311\end{bmatrix}\begin{bmatrix} +1.30412 & +0.24447 & -0.22798\\ +0.24447 & +1.76982 & +0.55494\\ -0.22798 & +0.55494 & +1.48410\end{bmatrix} && (28.5\text{-}21b)\\ A &= R'\,E && (28.5\text{-}21c)\\ &= \begin{bmatrix} +1.48410 & +0.55494 & +0.22798\\ +0.55494 & +1.76982 & -0.24447\\ +0.22798 & -0.24447 & +1.30412\end{bmatrix}\begin{bmatrix} +0.80311 & +0.29107 & +0.51988\\ -0.48689 & +0.82353 & +0.29107\\ +0.34342 & +0.48689 & -0.80311\end{bmatrix} && (28.5\text{-}21d)\end{aligned}$$

28.5.3 Sign decomposition

The *sign decomposition* can be defined as

$$A \;=\; S\,N \;=\; \Big(A(A^2)^{-1/2}\Big)\;\Big((A^2)^{1/2}\Big) \tag{28.5-22}$$

where $N = (A^2)^{-1/2}$ and $S = A\,(A^2)^{-1/2}$. The square root has to be chosen such that all its eigenvalues have positive real parts. The sign decomposition is undefined if A has eigenvalues on the imaginary axis. The matrix S is its own inverse (its eigenvalues are ± 1). The matrices A, S and N commute pairwise: $SN = NS$, $AN = NA$ and $AS = SA$.

Use

$$\begin{aligned} S_0 &= A && (28.5\text{-}23a)\\ Y_k &= \left(1 + \frac{1 - S_k^2}{2}\right) && (28.5\text{-}23b)\\ S_{k+1} &= S_k\,Y_k \quad \to S && (28.5\text{-}23c)\\ N_{k+1} &= S_{k+1}\,A \quad \to N && (28.5\text{-}23d) \end{aligned}$$

Numerical example: Let

$$A \;=\; \begin{bmatrix} +1.00000 & +1.00000 & +0.75000 \\ -0.50000 & +1.50000 & +1.00000 \\ +0.75000 & +0.50000 & -1.00000 \end{bmatrix} \tag{28.5-24}$$

then

$$\begin{aligned} A &= S\,N && (28.5\text{-}25a)\\ &= \begin{bmatrix} +0.90071 & -0.01706 & +0.29453 \\ -0.24065 & +0.95862 & +0.71389 \\ +0.62679 & +0.10775 & -0.85933 \end{bmatrix} \begin{bmatrix} +1.13014 & +1.02237 & +0.36392 \\ -0.18454 & +1.55423 & +0.06423 \\ -0.07158 & +0.35875 & +1.43718 \end{bmatrix} && (28.5\text{-}25b) \end{aligned}$$

where $S\,S = 1$. See [133] and also [134].

28.5.4 Pseudo-inverse

While we are at it: define a matrix A^+ as

$$A^+ \;:=\; (A\,A^T)^{-1}\,A^T = \big(V\,\Omega^{-2}\,V^T\big)\;\big(V\,\Omega\,U^T\big) = V\,\Omega^{-1}\,U^T \tag{28.5-26}$$

This looks suspiciously like the inverse of A. In fact, this is the *pseudo-inverse* of A:

$$A^+\,A \;=\; \big(V\,\Omega^{-1}\,U^T\big)\;\big(U\,\Omega\,V^T\big) = 1 \quad \text{but wait} \tag{28.5-27}$$

A^+ has the nice property to exist even if A^{-1} does not. If A^{-1} exists, it is identical to A^+. If not, $A^+\,A \neq 1$ but A^+ will give the best possible (in a least-square sense) solution $x^+ = A^+ b$ of the equation $A\,x = b$ (see [89, p.770]). To find $(A\,A^T)^{-1}$ use the iteration for the inverse:

$$\Phi(x) = x\;\big(1 + (1 - dx) + (1 - dx)^2 + \ldots\big) \tag{28.5-28}$$

with $d = A\,A^T$ and the start value $x_0 = 2 - n\,(A\,A^T)/\;||A\,A^T||^2$ where n is the dimension of A.

A pari/gp implementation of the pseudo-inverse using the SVD:

```
? A
  [+1.00 +1.00 +0.75 +2.00]
  [-0.50 +1.50 +1.00 +3.00]
  [+0.75 +0.50 -1.00 -3.00]

? t=matSVD(A); U=t[1]; d=t[2]; V=t[3];
? U
  [+0.644401153492 +0.438818890 +0.6262468643]
  [-0.695372132379 +0.676976586 +0.2411644651]
  [-0.318126941467 -0.590881276 +0.7413869203]

? d
  [+0.95641003 0 0]
  [0 +5.09161169 0]
  [0 0 +1.74234618]

? V
  [+0.787833655771 -0.067332385426 +0.60935354299]
  [-0.583139548860 +0.227598489665 +0.77980313700]
  [+0.110889336609 +0.313647647860 -0.01752654388]
  [+0.164225309908 +0.919396775264 -0.14243646797]

? Ax=matpseudoinv(A)
  [+0.744034618880 -0.497415792829 +0.005046325813]
  [-0.093004327360 +0.562176974103 +0.499369209273]
  [+0.095446097914 -0.041347314730 -0.080741213017]
  [+0.138692567521 -0.016875347613 -0.221929812661]

? A*Ax
  [+1.0000000000000 +3.78653234 E-29 -4.41762106 E-29]
  [+2.52435489 E-29 +1.0000000000000 -2.52435489 E-29]
  [-2.52435489 E-29 -2.52435489 E-29 +1.0000000000000]

? Ax*A
  [+0.9965272596551 +0.0004340925431 +0.0555638455173 -0.0193171181681]
  [+0.0004340925431 +0.9999457384321 -0.0069454806896 +0.0024146397710]
  [+0.0555638455173 -0.0069454806896 +0.1109784717229 +0.3090738906900]
  [-0.0193171181681 +0.0024146397710 +0.3090738906900 +0.8925485301897]
```

Figure 28.5-A: Numerical example for the pseudo-inverse computed by the SVD. We use a 3×4 matrix which is definitely not invertible. A working precision of 25 decimal digits was used, so $A\,A^{+} = 1$ to within that precision. On the other hand, $A^{+}\,A$ is not close to the unit matrix.

```
matpseudoinv(A)=
\\ Return pseudo-inverse of A
{
    local(t, x, U, d, V);
    t = matSVD(A);
    U = t[1];  d = t[2];  V = t[3];
    for (k=1, matsize(d)[1],
        x=d[k,k];  if (x>1e-15, d[k,k]=1/x, d[k,k]=0);
    );
    return( V*d*U~ );
}
```

Where the SVD is computed with the help of a routine (`qfjacobi()`) that returns the eigenvectors of a real symmetric matrix:

```
matSVDcore(A)=
\\ Singular value decomposition:
\\ Return [U, d, V] so that U*d*V~==A
\\ d is a diagonal matrix
\\ U, V are orthogonal
{
    local(U, d, V);  \\ returned quantities
    local(t, R, d1);
    R = conj(A~)*A;  \\ R==V*d^2*V~
    t = qfjacobi( R );  \\ fails with eigenvalues==zero
    V = t[2];
    d = real(sqrt(t[1]));
    d1 = d;
    for (k=1, length(d1), t=d1[k]; if (abs(t)>1e-16, t=1/t, t=0); d1[k]=t );
    d1 = matdiagonal(d1):
```

```
    d = matdiagonal(d);
    U = (A*V*d1);
    return( [U, d, V] );
}
```

The core routine is always called with a matrix A whose number of rows is greater or equal to its number of rows. Thereby we make sure that when the main routine with argument A computes U, d, V so that $A = U\,d\,V^T$ then with argument A^T we obtain X, d, Y where $A^T = X\,d\,Y^T$, $X = V$, and $Y = U$.

```
matSVD(A)=
{
    local(tq, t, U, d, V);
    t = matsize(A);
    tq=0;  if ( t[1]<t[2],  tq=1; A=A~; );
    t = matSVDcore(A);
    d = t[2];
    if ( tq,
        U = t[3];  V=t[1];
    , /* else */
        U = t[1];  V=t[3];
    );
    return( [U, d, V] );
}
```

For a numerical example see figure 28.5-A. The connection between the SVD of a matrix a and the eigenvectors of $A^T\,A$ is described in [145].

28.6 Goldschmidt's algorithm

A framework for the so-called *Goldschmidt algorithm* can be stated as follows. Initialize

$$x_0 \;=\; d^A, \qquad E_0 \;=\; d^B \tag{28.6-1a}$$

then iterate

$$P_{k+1} \;=\; 1 + \frac{1 - E_k}{a} \tag{28.6-1b}$$

$$x_{k+1} \;=\; x_k\,P_k^b \quad \to d^{A - B\,b/a} \tag{28.6-1c}$$

$$E_{k+1} \;=\; E_k\,P_k^a \quad \to 1 \tag{28.6-1d}$$

The algorithm converges quadratically. The updates for x and E (last two relations) can be computed independently. The iteration is not self-correcting, so the computations have to be carried out with full precision in all steps.

An invariant of the algorithm is given by x_k^a / E_k^b:

$$\frac{x_{k+1}^a}{E_{k+1}^b} \;=\; \frac{(x_k \cdot P_k^b)^a}{(E_k \cdot P_k^a)^b} \;=\; \frac{x_k^a}{E_k^b} \tag{28.6-2a}$$

Using

$$\frac{x_0^a}{E_0^b} \;=\; \frac{d^{A\,a}}{d^{B\,b}} \;=\; d^{A\,a - B\,b} \tag{28.6-2b}$$

and, as E converges to 1, we find that

$$\frac{x_\infty^a}{E_\infty^b} \;=\; x_\infty^a \;=\; \frac{x_0^a}{E_0^b} \tag{28.6-2c}$$

That is,

$$x_\infty \;=\; \left(\frac{x_0^a}{E_0^b}\right)^{1/a} \;=\; \frac{x_0}{E_0^{b/a}} \;=\; \frac{d^A}{d^{b\,B/a}} \;=\; d^{A - B\,b/a} \tag{28.6-2d}$$

One can now look for interesting special cases, b is set to one in what follows.

28.6.1 Algorithm for the a-th root

Solving $A - B/a = 1/a$ gives $B = A\,a - 1$ and especially $A = 0,\ B = 1$. That is, set

$$x_0 \;=\; d, \qquad E_0 \;=\; d^{a-1} \tag{28.6-3a}$$

then iterate

$$P_k \;:=\; 1 + \frac{1 - E_k}{a} \quad \to 1 \tag{28.6-3b}$$

$$x_{k+1} \;:=\; x_k \cdot P_k \tag{28.6-3c}$$

$$E_{k+1} \;:=\; E_k \cdot P_k^a \qquad \to 1 \tag{28.6-3d}$$

until x close enough to $x_\infty = d^{\frac{1}{a}}$

Computation of the square root

For $a = 2$ one obtains an algorithm for the computation of the square root:

$$\sqrt{d} \;=\; d \prod_{k=0}^{\infty} \frac{3 - E_k}{2} \tag{28.6-4}$$

where $E_0 = d$, $E_{k+1} := E_k \,(\frac{3-E_k}{2})^2$.

28.6.2 Algorithm for the inverse a-th root

Solving $A - B/a = -1/a$ gives $B = A\,a + 1$ and especially $A = 1,\ B = a - 1$. That is, set

$$x_0 \;=\; 1, \qquad E_0 \;=\; d \tag{28.6-5}$$

then iterate as in formulas 28.6-3b..28.6-3d until x close enough to $x_\infty = d^{-\frac{1}{a}}$.

Computation of the inverse

Setting $a = 1$ one obtains an algorithm for the inverse ($P_k = 1 + (1 - E_k) = 2 - E_k$):

$$\frac{1}{d} \;=\; \prod_{k=0}^{\infty} (2 - E_k) \tag{28.6-6}$$

where $E_0 = d$, $E_{k+1} := E_k \,(2 - E_k)$.

Computation of the inverse square root

For $a = 2$ one obtains an algorithm for the inverse square root ($P_k = 1 + (1 - E_k)/2 = (3 - E_k)/2$):

$$\frac{1}{\sqrt{d}} \;=\; \prod_{k=0}^{\infty} \frac{3 - E_k}{2} \tag{28.6-7}$$

where $E_0 = d$, $E_{k+1} := E_k \,(\frac{3-E_k}{2})^2$.

28.6.3 Higher order algorithms for the inverse a-th root

Higher order iterations are obtained by appending higher terms to the expression $\left(1+\frac{1-E_k}{a}\right)$ in the definitions of P_{k+1} as suggested by equation 28.3-4 on page 546 and the identification $y = 1 - E$:

$$
\begin{aligned}
E_{k+1} &= E_k\, P_k^a \quad \text{where} && \text{(28.6-8)}\\
P_k &= 1 + \frac{1-E_k}{a} \qquad \text{[second order]} && \text{(28.6-9)}\\
&\quad + \frac{(1+a)\,(1-E_k)^2}{2\,a^2} \qquad \text{[third order]}\\
&\quad + \frac{(1+a)\,(1+2a)\,(1-E_k)^3}{6\,a^3} \qquad \text{[fourth order]}\\
&\quad + \ \ldots\ +\\
&\quad + \frac{(1+a)\,(1+2a)\ldots(1+(n-1)\,a)\,(1-E_k)^n}{n!\,a^n} \qquad \text{[order } (n+1)\text{]}
\end{aligned}
$$

$x_0 = 1$ $E_0 = 2.0$ $P_0 = 0.90625$ $b_0 = 0.0$
$x_1 = 0.90625$ $E_1 = 1.3490314483642578125$ $P_1 = 0.93177697415069360431516543030738830566640625$ $b_1 = 1.5185$
$x_2 = 0.84442288282406607891061867121607065200805664062 5$ $E_2 = 1.01688061936626457410433320872039209492188542219092968$ $P_2 = 0.99582436942565084187883150540345532399967189056293558 21$ $b_2 = 5.8884$
$x_3 = 0.84089688481686585202128466050063877105975287186278309 56$ $E_3 = 1.00000223363323283559583877024921007574068879685671957$ $P_3 = 0.99999944159247134069773211917093099758090030134706071 62$ $b_3 = 18.772$
$x_4 = 0.84089641525371454412926831199731186378490804857313364 97$ $E_4 = 1.00000000000000000522367709431971479704373188248422463 7$ $P_4 = 0.99999999999999999869408072642007130500262990214776549 07$ $b_4 = 57.409$
$x_5 = 0.84089641525371454303112547623321489504003426235678452 49$ $E_5 = 1.00000000000000000000000000000000000000000000000000000 067$ $P_5 = 0.99999999999999999999999999999999999999999999999999999 9833$ $b_5 = 173.32$
$1/\sqrt[4]{2} = 0.84089641525371454303112547623321489504003426235678451 08\ldots$

Figure 28.6-A: Numerical quantities occurring in the computation of $1/\sqrt[4]{2}$ using a third order Goldschmidt algorithm.

As an example, the inverse fourth root of $d = 2$ can be obtained via the third order algorithm

$$
\begin{aligned}
x_0 &= 1 && \text{(28.6-10a)}\\
E_0 &= d \ = \ 2 && \text{(28.6-10b)}\\
E_{k+1} &= E_k\, P_k^4 \ = \ E_k \left(\frac{45 - 18\,E_k + 5\,E_k^2}{32}\right)^4 && \text{(28.6-10c)}\\
x_{k+1} &= x_k\, P_k && \text{(28.6-10d)}
\end{aligned}
$$

Figure 28.6-A shows the numerical values of x_k, E_k and P_k up to step $k = 6$. The approximate precision in bits of x_k is computed as $b_k = -\log(|1 - E_k|)/\log(2)$.

28.7 Products for the a-th root

Rewrite the well-known product form

$$\frac{1}{1-y} \;=\; (1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots \tag{28.7-1}$$

as

$$\frac{1}{1-y} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \quad Y_1 \;:=\; y, \qquad Y_{k+1} \;:=\; Y_k^2 \tag{28.7-2}$$

We give product forms for a-th roots and their inverses that generalize the relations above.

28.7.1 Second order products

Products for the square root and its inverse

For the inverse square root use $1/\sqrt{1-y} = (1+y/2)\cdot 1/\sqrt{1-y^2/4\,(3+y)}$, thereby

$$\frac{1}{\sqrt{1-y}} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \quad Y_1 \;:=\; \frac{y}{2}, \qquad Y_{k+1} \;:=\; Y_k^2\left(\frac{3}{2}+Y_k\right) \tag{28.7-3}$$

For the square root use $\sqrt{1-y} = (1-y/2)\cdot\sqrt{1-(y/(y-2))^2}$, so

$$\sqrt{1-y} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \quad Y_1 \;:=\; -\frac{y}{2}, \qquad Y_{k+1} \;:=\; -\frac{1}{2}\left(\frac{Y_k}{1+Y_k}\right)^2 \tag{28.7-4}$$

Products for the a-th root and its inverse

The relation for the inverse a-th root is

$$\frac{1}{\sqrt[a]{1-y}} \;=\; (1-y)^{-1/a} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \tag{28.7-5a}$$

$$Y_1 \;:=\; \frac{y}{a}, \qquad Y_{k+1} \;:=\; \frac{1}{a}\left(1-(1-a\,Y_k)\,(Y_k+1)^a\right) \tag{28.7-5b}$$

Alternatively,

$$\frac{1}{\sqrt[a]{d}} \;=\; x\,(1-y)^{-1/a} \;=\; x\,\prod_{k>0}(1+Y_k) \tag{28.7-6}$$

with $y := 1 - d\,x^a$ and the definitions 28.7-5b for Y_k. For the a-th root we obtain

$$\sqrt[a]{1-y} \;=\; (1-y)^{1/a} \;=\; \prod_{k>0}(1+Y_k) \quad \text{where} \tag{28.7-7a}$$

$$Y_1 \;:=\; \frac{y}{a}, \qquad Y_{k+1} \;:=\; \frac{1}{a}\,\frac{(1+a\,Y_k)-(1+Y_k)^a}{(1+Y_k)^a} \tag{28.7-7b}$$

28.7.2 Products of arbitrary order

We want to obtain an n-th order product for the inverse a-th root

$$\frac{1}{\sqrt[a]{1-y}} \;=\; \prod_{k>0}\left(1+T(Y_k)\right) \quad \text{where} \quad Y_1 \;=\; y, \qquad Y_{k+1} \;=\; N(Y_k) \tag{28.7-8}$$

The functions T and N have to be determined. Set

$$[1-y]^{-\frac{1}{a}} \;=\; \left(1+T(Y_1)\right)\,\left[\left(1+T(Y_1)\right)^a\,(1-y)\right]^{-\frac{1}{a}} \tag{28.7-9a}$$

$$=:\; \left(1+T(Y_1)\right)\,[1-Y_2]^{-\frac{1}{a}} \tag{28.7-9b}$$

where $1+T(Y_1)$ is the Taylor expansion

$$[1-y]^{-\frac{1}{a}} \;=\; 1+\frac{y}{a}+\frac{(1+a)\,y^2}{2\,a^2}+\frac{(1+a)(1+2a)\,y^3}{6\,a^3}+\ldots+\frac{\prod_{k=1}^{n-1}(1+k\,a)}{n!\,a^n}\,y^n+\ldots \tag{28.7-10}$$

up to order $n-1$. The Taylor expansion of Y_2 starts with a term $\sim y^n$. Using $Y_{k+1}=N(Y_k)$ as suggested by the relation between Y_2 and Y_1 gives a product with n-th order convergence. For example, for a third order product for $1/\sqrt{1-y}$, set

$$T(Y) \;:=\; \frac{1}{2}\,Y+\frac{3}{8}\,Y^2 \tag{28.7-11}$$

Now solve $\left(1+T(Y_1)\right)^2(1-y)=(1-Y_2)$ for Y_2 to obtain

$$Y_2 \;=\; \frac{5\,y^3}{8}+\frac{15\,y^4}{64}+\frac{9\,y^5}{64} \;=:\; N(y) \tag{28.7-12}$$

Then, finally, ($Y_1:=y$ and)

$$\frac{1}{\sqrt{1-y}} \;=\; \prod_{k>0}\left(1+T(Y_k)\right) \quad \text{where} \tag{28.7-13a}$$

$$T(Y) \;:=\; \frac{1}{2}\,Y+\frac{3}{8}\,Y^2 \tag{28.7-13b}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; \frac{Y_k^3}{64}\,\left(40+15\,Y_k+9\,Y_k^2\right) \tag{28.7-13c}$$

Replace relation 28.7-13c by $Y_{k+1}=1-\left(1+T(Y_k)\right)^a\,(1-Y_k)$ to obtain the general formula for the inverse a-th root.

The second order products lead to expressions that are especially nice:

$$\frac{1}{\sqrt[a]{1-y}} \;=\; \prod_{k>0}\left(1+T(Y_k)\right) \quad \text{where} \quad T(y) \;:=\; +\frac{y}{a} \qquad \text{and} \tag{28.7-14a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; 1-\left(1+\frac{y}{a}\right)^a\,(1-y) \tag{28.7-14b}$$

$$\frac{1}{\sqrt[a]{1+y}} \;=\; \prod_{k>0}\left(1+T(Y_k)\right) \quad \text{where} \qquad T(y) \;:=\; -\frac{y}{a} \qquad \text{and} \tag{28.7-15a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; \left(1-\frac{y}{a}\right)^a\,(1+y)-1 \tag{28.7-15b}$$

$$\sqrt[a]{1-y} \;=\; \prod_{k>0}\left(1+T(Y_k)\right) \quad \text{where} \quad T(y) \;:=\; -\frac{y}{a} \qquad \text{and} \tag{28.7-16a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; \frac{(a-Y_k)^a-(1-Y_k)\,a^a}{(a-Y_k)^a} \;=\; 1-\frac{(1-Y_k)\,a^a}{(a-Y_k)^a} \tag{28.7-16b}$$

$$\sqrt[a]{1+y} \;=\; \prod_{k>0}\,(1+T(Y_k)) \quad \text{where} \quad T(y) \;:=\; +\frac{y}{a} \qquad \text{and} \tag{28.7-17a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; \frac{(1+Y_k)\,a^a-(a+Y_k)^a}{(a+Y_k)^a} \;=\; \frac{(1+Y_k)\,a^a}{(a+Y_k)^a}-1 \tag{28.7-17b}$$

The third order product for $\frac{1}{\sqrt[a]{1-y}}$ is

$$\frac{1}{\sqrt[a]{1-y}} \;=\; \prod_{k>0}\,(1+T(Y_k)) \quad \text{where} \quad T(y) \;=\; +\frac{y}{a}+\frac{y^2\,(1+a)}{2\,a^2} \qquad \text{and} \tag{28.7-18a}$$

$$Y_{k+1} \;=\; N(Y_k) \;:=\; 1-\left(1+\frac{y}{a}+\frac{y^2\,(1+a)}{2\,a^2}\right)^a\,(1-y) \tag{28.7-18b}$$

$$\;=\; 1-(1+T(y))^a\,(1-y) \tag{28.7-18c}$$

28.7.3 Third order product for the a-th root

```
? default(realprecision,55);
? a=3;d=2;al=a-1;be=a+1;
? F(x)=(al*x^a+be*d)/(be*x^a+al*d)  \\ == (x^3 + 4)/(2*x^3 + 2)
? p=99.0;  \\ very bad approximation to the root
? for(k=0,25,p*=F(p);print("  ",p););
  49.500153045449860867772853756570132948572606410388539 63
  24.750688696322535793295398076760659038190058852964934 60
  12.377792780128382599987309228382889563737720225044012 95
  6.1986817294679733089803948935359831901917837320165830 57
  3.1382160956525775164587135127175212698603091937687217 90
  1.7166434303974992394555143292360395395658208028534524 86
  1.2833235143227843328303771164010152645995928582804900 32
  1.2599262845711532794913599247538658268681639208667671 05
  1.2599210498948732250077509793665642207537327503965189 86
  1.2599210498948731647672106072782283505702514647015997 90
  1.2599210498948731647672106072782283505702514647015079 80
  1.2599210498948731647672106072782283505702514647015079 80
```

Figure 28.7-A: Computation of $\sqrt[3]{2}$ with a very bad initial approximation.

The third order iteration given as relation 28.2-7 on page 545 gives a simple product for $\sqrt[a]{d}$. Let

$$P_k \;:=\; \prod_{j=0}^{k} Y_k \tag{28.7-19a}$$

where Y_0 is sufficiently near to $\sqrt[a]{d}$ and $Y_k = F(P_{k-1})$ where

$$F(x) \;:=\; \frac{\alpha\,x^a+\beta\,d}{\beta\,x^a+\alpha\,d} \tag{28.7-19b}$$

with $\alpha = a-1$ and $\beta = a+1$. Then $P_\infty = \sqrt[a]{d}$. Figure 28.7-A shows the numerical quantities with the computation of $\sqrt[3]{2}$ with a starting value $Y_0 = 99$ that is not at all close to the root. We have $F(x) = \frac{x^3+4}{2x^3+2}$ which is $\approx \frac{1}{2}$ for large values of x. Therefore the big initial values are repeatedly halved before the third order convergence begins.

28.8 Divisionless iterations for polynomial roots

Let $f(x)$ be a polynomial in x with simple roots only, then

$$\Phi(x) \;:=\; x-p(x)\,f(x) \quad \text{where} \quad p(x) := f'(x)^{-1} \bmod f(x) \tag{28.8-1}$$

is a second order iteration for the roots of $f(x)$. The iteration involves no long division if all coefficients are small rationals. Instead of dividing by $f'(x)$ a multiplication by the modular inverse $p(x)$ is used. As $\deg(p) < \deg(f)$ we have $\deg(\Phi) \leq 2\,\deg(f) - 1$.

For example, for $f(x) = a\,x^2 + b\,x + c$ we obtain

$$\Phi(x) \quad = \quad x - \frac{2\,a\,x + b}{\Delta}\,f(x) \quad \text{where} \quad \Delta = b^2 - 4\,a\,c \tag{28.8-2}$$

The general expressions for polynomials of orders > 2 get complicated. However, for fixed polynomial coefficients the iteration is more manageable. For example, with $f(x) = x^3 + 5\,x + 1$ we obtain

$$\Phi(x) \quad = \quad x + \frac{f(x)}{527}\,\left(30\,x^2 - 9\,x + 100\right) \tag{28.8-3}$$

For the polynomial $x^n - d$ we have $p = x/(n\,d)$ and the iteration is (relation 28.3-16 on page 548):

$$\Phi(x) \quad = \quad x - \frac{x}{n\,d}\,(x^n - d) \;=\; x + \frac{1}{n}\,\left(x - \frac{x^{n+1}}{d}\right) \tag{28.8-4}$$

The construction is given in [136] where a method to construct divisionless iterations of arbitrary order is given: let $p\,f' + q\,f \equiv 1$, and

$$\Phi_1 \quad := \quad x - p_1\,f, \qquad p_1 \;:=\; p \tag{28.8-5a}$$

$$p_r \quad := \quad p\,p'_{r-1} - (r-1)\,q\,p_{r-1} \tag{28.8-5b}$$

$$\Phi_r \quad := \quad \Phi_{r-1} + (-1)^r\,p_r\,f^r/r! \tag{28.8-5c}$$

then Φ_r is an iteration of order $r + 1$.

Chapter 29

Iterations for the inversion of a function

In this chapter we study some general expressions for iterations for the zero of a function. Two schemes that give (arbitrary order, one-point) iterations, Householder's formula and Schröder's formula, are given. Several methods to construct alternative iterations are described. Moreover, iterations that also converge for multiple roots and a technique to turn a linear iteration into a super-linear one are presented.

29.1 Iterations and their rate of convergence

An *iteration* for a zero r (or root, $f(r) = 0$) of a function $f(x)$ are themselves functions $\Phi(x)$ that, when used like

$$x_{k+1} \quad = \quad \Phi(x_k) \tag{29.1-1}$$

will make x_k converge towards the root: $x_\infty = r$. Convergence is subject to the condition that x_0 was chosen close enough to r. The function $\Phi(x)$ must (and can) be constructed so that it has an attracting fixed point where $f(x)$ has a zero:

$$\Phi(r) \quad = \quad r \qquad \text{(fixed point)} \tag{29.1-2}$$

$$|\Phi'(r)| \quad < \quad 1 \qquad \text{(attracting)} \tag{29.1-3}$$

This type of iteration is called a *one-point iteration*. There are also *multi-point iterations*, these are of the form $x_{k+1} = \Phi(x_k, x_{k-1}, \ldots, x_{k-j}), j \geq 1$. An example is the two-point iteration known as the *secant method*

$$x_{k+1} \quad = \quad \Phi(x_k, x_{k-1}) \;=\; x_k - f(x_k)\,\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \tag{29.1-4}$$

We are mainly concerned with one-point iterations in what follows.

The order of convergence (or simply *order*) of a given iteration can be defined as follows: let $x = r \cdot (1+e)$ with $|e| \ll 1$ and $\Phi(x) = r \cdot (1 + \alpha e^n + O(e^{n+1}))$, then the iteration Φ is called *linear* (or first order) if $n = 1$ (and $|\alpha| < 1$). A linear iteration improves the result by (roughly) adding a constant amount of correct digits with every step.

A *super-linear* iteration does better than that: The number of correct digits grows exponentially (to the base n) at each step. Super-linear convergence of order n should better be called *exponential of order n*.

Iterations of second order ($n = 2$) are often called *quadratic* (or *quadratically convergent*), those of third order *cubic* iterations. Fourth, fifth and sixth order iterations are called *quartic*, *quintic* and *sextic* and so on. We note that the two-point iteration relation 29.1-4 has order $(\sqrt{5}+1)/2 \approx 1.618$, see [137, p.152].

It is conceivable to find iterations that do converge better than linear but less than exponential to any base: imagine an iteration that produces proportional k^2 digits at step k (this is *not* quadratical convergence which produces proportional 2^k correct digits at step k). That case is not covered by the 'order-n' notion just introduced. However, those (super-linear but sub-exponential) iterations are not usually encountered and we actually won't meet one. In fact, the constructions used in this chapter cannot produce such an iteration. For a more fine-grained definition of the concept of order see [59, p.21].

For $n \geq 2$ the iteration function Φ has a *super-attracting fixed point* at r: $\Phi'(r) = 0$. For an iteration of order n one has

$$\Phi'(r) = 0, \quad \Phi''(r) = 0, \quad \ldots, \quad \Phi^{(n-1)}(r) = 0 \tag{29.1-5}$$

There is no standard term for emphasizing the number of derivatives vanishing at the fixed point: *super-attracting of order n* might be appropriate.

To any iteration of order n for a function f one can add a term $f(x)^n \cdot \varphi(x)$ (where $\varphi(x)$ is an arbitrary function that is analytic in a neighborhood of the root) without changing the order of convergence. That term is assumed to be zero in what follows. The statement can easily be checked by verifying that the first $n-1$ derivatives of $\Phi_n(x) + f(x)^n \cdot \varphi(x)$, evaluated at the root r, equal zero.

Any two one-point iterations of the same order n differ by a term $f(x)^n \cdot \varphi(x)$.

Any two iterations of the same order n differ by a term $(x-r)^n \, \nu(x)$ where $\nu(x)$ is a function that is finite at r [137, p.174, ex.3].

Any one-point iteration of order n must explicitly evaluate f, f', ..., $f^{(n-1)}$ [229, p.98]. For methods to find zeros and extrema without evaluating derivatives see [59].

29.2 Schröder's formula

For $n \geq 2$ then the expression

$$S_n(x) \quad := \quad x + \sum_{t=1}^{n-1} (-1)^t \, \frac{f(x)^t}{t!} \left(\frac{1}{f'(x)} \, \frac{\partial}{\partial x} \right)^{t-1} \, \frac{1}{f'(x)} \tag{29.2-1}$$

gives a n-th order iteration for a (simple) root r of f [210, p.13]. That is,

$$\begin{aligned} S \quad := \quad S_\infty(x) \; = \; x \quad & - \frac{f}{1!\, f'} \quad - \frac{f^2}{2!\, f'^3} \cdot f'' \quad - \frac{f^3}{3!\, f'^5} \cdot \left(3f''^2 - f' f''' \right) \\ & - \frac{f^4}{4!\, f'^7} \cdot \left(15 f''^3 - 10 f' f'' f''' + f'^2 f'''' \right) \\ & - \frac{f^5}{5!\, f'^9} \cdot \left(105 f''^4 - 105 f' f''^2 f''' + 10 f'^2 f'''^2 + 15 f'^2 f'' f'''' - f'^3 f''''' \right) - \ldots \end{aligned} \tag{29.2-2}$$

The second order iteration is the well-known Newton iteration. The third order iteration, obtained upon truncation after the third term on the right hand side, and written as

$$S_3 \quad = \quad x - \frac{f}{f'} \left(1 + \frac{f f''}{2 f'^2} \right) \tag{29.2-3}$$

is sometimes referred to as *Householder's method*. Approximating the second term on the right hand side gives *Halley's formula*:

$$H_3 \quad = \quad x - \frac{f}{f'}\left(1 - \frac{ff''}{2f'^2}\right)^{-1} \tag{29.2-4}$$

Write

$$S \quad = \quad x - U_1\frac{f}{1!\,f'} - U_2\frac{f^2}{2!\,f'^3} - U_3\frac{f^3}{3!\,f'^5} - \ldots - U_n\frac{f^n}{n!\,f'^{2n-1}} - \ldots \tag{29.2-5}$$

then $U_1 = 1$, $U_2 = f''$, $U_3 = 3f''^2 - f'f'''$, and we have the recursion

$$U_n \quad = \quad (2n-3)f''U_{n-1} - f'U'_{n-1} \tag{29.2-6}$$

An alternative recursion is given in [229, p.83], write

$$S \quad = \quad x - Y_1\left(\frac{f}{f'}\right) - Y_2\left(\frac{f}{f'}\right)^2 - Y_3\left(\frac{f}{f'}\right)^3 - \ldots - Y_t\left(\frac{f}{f'}\right)^t - \ldots \tag{29.2-7}$$

then $Y_1 = 1$ and

$$Y_t \quad = \quad \frac{1}{t}\left(2\,(t-1)\,\frac{f''}{2f'}\,Y_{t-1} - Y'_{t-1}\right) \tag{29.2-8}$$

Relation 29.2-1 on the preceding page with $f(x) = 1/x^a - d$ gives the 'division-free' iteration 28.3-5 on page 546 for arbitrary order. For $f(x) = \log(x) - d$ one obtains the iteration 31.2-9a on page 603.

For $f(x) = x^2 - d$ one obtains

$$S(x) \quad = \quad x - \left(\frac{x^2-d}{2x} + \frac{\left(x^2-d\right)^2}{8x^3} + \frac{\left(x^2-d\right)^3}{16x^5} + \frac{5\left(x^2-d\right)^4}{128x^7} + \ldots\right) \tag{29.2-9a}$$

$$= \quad x - 2x\cdot\left(Y + Y^2 + 2\,Y^3 + 5\,Y^4 + 14\,Y^5 + 42\,Y^6 + \ldots\right) \quad \text{where} \quad Y := \frac{x^2-d}{(2x)^2} \tag{29.2-9b}$$

The coefficients of the powers of Y are the Catalan numbers, see section 13.3 on page 306.

A simple derivation of Schröder's formula *

The starting point is the Taylor series of a function f around x_0:

$$f(x) \quad = \quad \sum_{k=0}^{\infty}\frac{1}{k!}f^{(k)}(x_0)\,(x-x_0)^k \tag{29.2-10a}$$

$$= \quad f(x_0) + f'(x_0)\,(x-x_0) + \frac{1}{2}f''(x_0)\,(x-x_0)^2 + \frac{1}{6}f'''(x_0)\,(x-x_0)^3 + \ldots \tag{29.2-10b}$$

Now let $f(x_0) = y_0$ and r be the zero of f (that is, $f(r) = 0$). We expand the inverse $g = f^{-1}$ around y_0:

$$g(0) \quad = \quad \sum_{k=0}^{\infty}\frac{1}{k!}g^{(k)}(y_0)\,(0-y_0)^k \tag{29.2-11a}$$

$$= \quad g(y_0) + g'(y_0)\,(0-y_0) + \frac{1}{2}g''(y_0)\,(0-y_0)^2 + \frac{1}{6}g'''(y_0)\,(0-y_0)^3 + \ldots \tag{29.2-11b}$$

Using $x_0 = g(y_0)$ and $g(0) = r$ we obtain

$$r = x_0 - g'(y_0)\,f(x_0) + \frac{1}{2}g''(y_0)\,f(x_0)^2 - \frac{1}{6}g'''(y_0)\,f(x_0)^3 + \dots \tag{29.2-12}$$

Remains to express the derivatives of the inverse g in terms of (derivatives of) f. Set

$$f \circ g = \mathrm{id}\,, \qquad \text{that is:} \quad f(g(x)) = x \tag{29.2-13}$$

and derive the equation (chain rule) to obtain $g'(f(x))\,f'(x) = 1$, so $g'(y) = \frac{1}{f'(x)}$. Derive $f(g(x)) - x$ multiple times to obtain (arguments y of g and x of f are omitted for readability):

$$1 = f'g' \tag{29.2-14a}$$
$$0 = g'f'' + f'^2 g'' \tag{29.2-14b}$$
$$0 = g'f''' + 3f'f''g'' + f'^3 g''' \tag{29.2-14c}$$
$$0 = g'f'''' + 4f'g''f''' + 3f''^2 g'' + 6f'^2 f''g''' + f'^4 g'''' \tag{29.2-14d}$$

This system of linear equations in the derivatives of g can be solved successively for g', g'', g''', ...:

$$g' = \frac{1}{f'} \tag{29.2-15a}$$
$$g'' = -\frac{f''}{f'^3} \tag{29.2-15b}$$
$$g''' = \frac{1}{f'^5}\left(3f''^2 - f'f'''\right) \tag{29.2-15c}$$
$$g'''' = \frac{1}{f'^7}\left(10f'f''f''' - 15f''^3 - f'^2 f''''\right) \tag{29.2-15d}$$
$$g''''' = \frac{1}{f'^9}\left(105f''^4 - f'^3 f''''' - 105f'f''^2 f''' + 15f'^2 f''f'''' + 10f'^2 f'''^2\right) \tag{29.2-15e}$$

Thereby equation 29.2-12 can be written as

$$\begin{aligned} r &= x - \frac{1}{f'}\,f + \frac{1}{2}\left(-\frac{f''}{f'^3}\right)\,f^2 - \frac{1}{6}\left(\frac{1}{f'^5}\left(3f''^2 - f'f'''\right)\right)\,f^3 + \dots \\ &= x - \frac{f}{1!\,f'} - \frac{f^2}{2!\,f'^3}\cdot f'' - \frac{f^3}{3!\,f'^5}\cdot\left(3f''^2 - f'f'''\right) - \dots \end{aligned} \tag{29.2-16}$$

which is Schröder's iteration, equation 29.2-2 on page 564.

29.3 Householder's formula

The following expression gives for $n \geq 2$ a n-th order iteration for a (simple) root r of f [137, p.169]:

$$H_n(x) := x + (n-1)\,\frac{\left(\frac{1}{f(x)}\right)^{(n-2)}}{\left(\frac{1}{f(x)}\right)^{(n-1)}} \tag{29.3-1}$$

We have

$$H_2 = x - \frac{f}{f'} \tag{29.3-2a}$$

$$H_3 = x - \frac{2ff'}{2f'^2 - ff''} \tag{29.3-2b}$$

$$H_4 = x - \frac{3f(ff'' - 2f'^2)}{6ff'f'' - 6f'^3 - f^2f'''} \tag{29.3-2c}$$

$$H_5 = x + \frac{4f\left(6f'^3 - 6ff'f'' + f^2f'''\right)}{f^3f'''' - 24f'^4 + 36ff'^2f'' - 8f^2f'f''' - 6f^2f''^2} \tag{29.3-2d}$$

The second order variant is Newton's formula, the third order iteration is Halley's formula.

Kalantari and Gerlach [144] define the iteration

$$B_m = x - f\,\frac{D_{m-2}}{D_{m-1}} \tag{29.3-3a}$$

where $m \geq 2$, $D_0 = 1$, $D_1 = f'$, and

$$D_m = \det\begin{pmatrix} f' & \frac{f''}{2!} & \cdots & \frac{f^{(m-1)}}{(m-1)!} & \frac{f^{(m)}}{m!} \\ f & f' & \ddots & \ddots & \frac{f^{(m-1)}}{(m-1)!} \\ 0 & f & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \frac{f''}{2!} \\ 0 & 0 & \ddots & f & f' \end{pmatrix} \tag{29.3-3b}$$

The iteration turns out to be identical to the one of Householder ($B_n = H_n$). A recursive definition for D_m is given by

$$D_m = \sum_{i=1}^{m} (-1)^{i-1} f^{i-1} \frac{f^{(i)}}{i!} D_{m-i} \tag{29.3-4}$$

The derivation of Halley's formula by applying Newton's formula to $f/\sqrt{f'}$ can be generalized to produce m-order iterations as follows: Let $F_1 = f$ and for $m \geq 2$ let

$$F_m = \frac{F_{m-1}}{\sqrt[m]{F'_{m-1}}} \tag{29.3-5a}$$

$$H_m = x - \frac{F_{m-1}}{F'_{m-1}} \tag{29.3-5b}$$

That this recursion indeed gives the Householder iteration is shown in [144].

An alternative recursive formulation (also given in [144], ascribed to Ford/Pennline) is

$$Q_2 = 1 \tag{29.3-6a}$$

$$Q_m = f'\,Q_{m-1} - \frac{1}{m-2}\,f\,Q'_{m-1} \tag{29.3-6b}$$

$$H_m = x - f\,\frac{Q_m}{Q_{m-1}} \tag{29.3-6c}$$

The Taylor series of the k-th order Householder iteration around $f = 0$ up to order $k-1$ gives the k-th order Schröder iteration.

29.4 Dealing with multiple roots

The iterations given so far will not converge at the stated order if f has a multiple root at r. As an example consider the function

$$f(x) \quad = \quad (x^2 - d)^m \quad \text{where} \quad m \in \mathbb{N} \tag{29.4-1}$$

The iteration $\Phi(x) = x - f/f'$ is

$$\Phi_?(x) \quad = \quad x - \frac{x^2 - d}{m2x} \tag{29.4-2}$$

Its convergence is only linear for $m > 1$: $\Phi(\sqrt{d}(1+e)) = \sqrt{d}(1 + \frac{m-1}{m}e + O(e^2))$

Householder [137, p.161, ex.6] gives a second order iteration for a root of known multiplicity m as

$$\Phi_2(x) = x - m \cdot \frac{f}{f'} \tag{29.4-3}$$

Note that with the example above we obtain a quadratic iteration.

For roots of unknown multiplicity use the general expressions for iterations with $F := f/f'$ instead of f. Both F and f have the same set of roots, but all roots of F are simple. To illustrate this consider a function f that has a root of multiplicity m at r: $f(x) := (x-r)^m\, h(x)$ with $h(r) \neq 0$. Then

$$f'(x) \quad = \quad m\,(x-r)^{m-1}\, h(x) + (x-r)^m\, h'(x) \tag{29.4-4a}$$

$$\quad = \quad (x-r)^{m-1}\, \Big(m\, h(x) + (x-r)\, h'(x)\Big) \tag{29.4-4b}$$

and

$$F(x) = f(x)/f'(x) \quad = \quad (x-r)\, \frac{h(x)}{m\, h(x) + (x-r)\, h'(x)} \tag{29.4-5}$$

The fraction on the right hand side does not vanish at the root r.

Plugging $F = f/f'$ into Householder's formula (relation 29.3-1) we get the following iterations denoted by $H_k^{\%}$, the iterations H_k are given for comparison:

$$H_2 \quad = \quad x - \frac{f}{f'} \tag{29.4-6a}$$

$$H_2^{\%} \quad = \quad x - \frac{ff'}{f'^2 - ff''} \tag{29.4-6b}$$

$$H_3 \quad = \quad x - \frac{2ff'}{2f'^2 - ff''} \tag{29.4-6c}$$

$$H_3^{\%} \quad = \quad x + \frac{2f^2 f'' - 2ff'^2}{2f'^3 - 3ff'f'' + f^2 f'''} \tag{29.4-6d}$$

$$H_4 \quad = \quad x + \frac{3f^2 f'' - 6ff'^2}{6f'^3 - 6ff'f'' + f^2 f'''} \tag{29.4-6e}$$

$$H_4^{\%} \quad = \quad x + \frac{6ff'^3 + 3f^3 f''' - 9f^2 f' f''}{f^3 f'''' - 6f'^4 + 12ff'^2 f'' - 4f^2 f' f''' - 3f^2 f''^2} \tag{29.4-6f}$$

$$H_5 \quad = \quad x + \frac{24ff'^3 + 4f^3 f''' - 24f^2 f' f''}{f^3 f'''' - 24f'^4 + 36ff'^2 f'' - 8f^2 f' f''' - 6f^2 f''^2} \tag{29.4-6g}$$

The terms in the numerators and denominators of $H_k^{\%}$ and H_{k+1} are identical up to the integral constants. A alternative form for $H_k^{\%}$ is given by

$$H_k^{\%} \quad = \quad x + (k-1)\, \frac{(\log(f))^{(k-1)}}{(\log(f))^{(k)}} \tag{29.4-7}$$

Schröder's formula (relation 29.2-1), when inserting f/f', becomes:

$$\begin{aligned} S^{\%} \;=\; & x - \frac{f f'}{(f'^2 - f f'')} - \frac{f^2 f' \left(f f' f''' - 2 f f''^2 + f'^2 f''\right)}{2\left(f f'' - f'^2\right)^3} - \\ & - \frac{f^3 f' \left(2 f f'^3 f'' f''' \pm \ldots - 3 f^2 f'^2 f'''^2\right)}{6\left(f f'' - f'^2\right)^5} - \frac{f^4 f' \left(3 f'^8 f'''' \pm \ldots - 36 f^3 f'^2 f''^2 f'''^2\right)}{24\left(f f'' - f'^2\right)^7} - \\ & - \ldots - \frac{f^k f' (\ldots)}{k!\left(f f'' - f'^2\right)^{2k-1}} \end{aligned} \tag{29.4-8}$$

Checking convergence with the example function (relation 29.4-1) we began with: the second order iteration is

$$\Phi_2^{\%}(x) \;=\; S_2 \;=\; H_2 \;=\; x + x\frac{d - x^2}{d + x^2} = \frac{2\,d\,x}{x^2 + d} \tag{29.4-9}$$

Convergence is indeed second order, as

$$\Phi_2^{\%}(\sqrt{d}\cdot\frac{1-e}{1+e}) \;=\; \sqrt{d}\cdot\frac{1-e^2}{1+e^2} \tag{29.4-10}$$

which holds independent of m. The expression is similar to relation 28.2-4 on page 544. In general, with $H_k^{\%}$ one obtains for the square root:

$$\Phi_k^{\%}(\sqrt{d}\cdot\frac{1-e}{1+e}) \;=\; \sqrt{d}\cdot\frac{1-e^k}{1+e^k} \tag{29.4-11}$$

Using Schröder's third order formula for f/f' with f as in 29.4-1 we obtain a beautiful *fourth* order iteration for $\sqrt{d}$:

$$S_3^{\%}(x) \;=\; x + x\,\frac{d - x^2}{d + x^2} + x\,d\,\frac{(d - x^2)^2}{(d + x^2)^3} \tag{29.4-12a}$$

$$S_3^{\%}(\sqrt{d}\,\frac{1-e}{1+e}) \;=\; \sqrt{d}\,\frac{1 + 3e^2 - 3e^4 - e^6}{1 + 3e^2 + 3e^4 + e^6} \tag{29.4-12b}$$

$$\;=\; \sqrt{d}\,\frac{1-c}{1+c} \quad \text{where} \quad c = e^4\,\frac{e^2+3}{3e^2+1} \tag{29.4-12c}$$

In general, the $(1 + a\,k)$-th order Schröder iteration for $1/\sqrt[a]{d}$ obtained through f/f' has an order of convergence that exceeds the expected order by one. The third order Schröder iteration for $f(x) = 1 - d\,x^2$ is

$$S_3^{\%}(x) \;=\; x + x\,\frac{1 - dx^2}{1 + dx^2} + x\,\frac{(1 - dx^2)^2}{(1 + dx^2)^3} \tag{29.4-13}$$

The iteration also has 4th order convergence and the error expression $S_3^{\%}(\frac{1}{\sqrt{d}}\,\frac{1-e}{1+e})$ is the same (replacing $\sqrt{d}$ by $1/\sqrt{d}$) as in relation 29.4-12b.

29.5 More iterations

Rational iterations from Padé approximants

The $[i, j]$-th Padé approximant of Φ_n in f gives an iteration of order $p = i + j + 1$ (if $n \geq p$). Write $P_{[i,j]}$ for an iteration (of order $i + j + 1$) that is obtained using the Padé approximant $[i, j]$. For the second order (where the Newton iteration is $P_{[1,0]}(x) = x - \frac{f}{f'}$) this method gives one alternative form, namely

$$P_{[0,1]}(x) \;=\; x^2\,\frac{f'}{f + x\,f'} \;=\; x - \frac{x\,f}{f + x\,f'} \;=\; x - \frac{x\,f}{(x\,f)'} \;=\; x\left(1 + \frac{f}{x\,f'}\right)^{-1} \tag{29.5-1}$$

For the third order we find $P_{[2,0]}(x) = S_3(x)$, $P_{[1,1]}(x) = H_3(x)$, and

$$\begin{aligned} P_{[0,2]}(x) &= \frac{2x^3 f'^3}{2f^2 f' + 2xff'^2 + xf^2 f'' + 2x^2 f'^3} && (29.5\text{-}2a)\\ &= x - \frac{xf\left(2ff' + xff'' + 2xf'^2\right)}{\left(2f^2 f' + 2xff'^2 + xf^2 f'' + 2x^2 f'^3\right)} && (29.5\text{-}2b)\\ &= x\left(1 + \frac{f}{xf'} + \frac{f^2}{x^2 f'^2} + \frac{f^2 f''}{2xf'^3}\right)^{-1} && (29.5\text{-}2c)\\ &= x\left(1 + \frac{f}{(x\,f')} + \frac{f^2(x^2\,f'')}{2(x\,f')^3} + \frac{f^2}{(x\,f')^2}\right)^{-1} && (29.5\text{-}2d) \end{aligned}$$

Alternatively one can use the Padé approximant $A_{[i,j]}$ of $(\Phi(x)-x)/f$ in f where $\Phi(x)$ is a given iteration of order $\geq i+j+2$. Then $\Phi^+_{[i,j]} := x + f \cdot A_{[i,j]}$ is an iteration which has order $n = i+j+2$.

$$\begin{aligned} \Phi^+_{[0,1]}(x) &= x - \frac{2ff'}{2f'^2 - ff''} \;=\; H_3(x) && (29.5\text{-}3a)\\ \Phi^+_{[1,0]}(x) &= x - \frac{f\left(f\,f'' + 2\,f'^2\right)}{2\,f'^3} \;=\; S_3(x) && (29.5\text{-}3b) \end{aligned}$$

The iterations Φ^+ of order n are expressions in $x, f, f', \ldots, f^{(n-1)}$. Fourth order iterations are

$$\begin{aligned} \Phi^+_{[2,0]}(x) &= x - \frac{f\left(6\,f'^4 + 3\,f\,f'^2\,f'' - f^2\,f'\,f''' + 3\,f^2\,f''^2\right)}{6\,f'^5} && (29.5\text{-}4a)\\ &= x - \frac{f}{f'} - \frac{f^2}{2\,f'^3}\cdot f'' - \frac{f^3}{6\,f'^5}\cdot\left(3f''^2 - f'\,f'''\right) \;=\; S_4(x) && (29.5\text{-}4b)\\ &= x - \frac{f^2}{f'^2}\left(\frac{f'}{f} + \frac{f''}{2\,f'} + \frac{f\left(3\,f''^2 - f'\,f'''\right)}{6\,f'^3}\right) && (29.5\text{-}4c)\\ \Phi^+_{[1,1]}(x) &= x - \frac{f\left(2\,f\,f'\,f''' - 3\,f\,f''^2 + 6\,f'^2\,f''\right)}{f'\left(2\,f\,f'\,f''' - 6\,f\,f''^2 + 6\,f'^2\,f''\right)} && (29.5\text{-}4d)\\ \Phi^+_{[0,2]}(x) &= x - \frac{12\,f\,f'^3}{\left(12\,f'^4 - 6\,f\,f'^2\,f'' + 2\,f^2\,f'\,f''' - 3\,f^2\,f''^2\right)} && (29.5\text{-}4e)\\ &= x - \left(\frac{f'}{f} - \frac{f''}{2\,f'} - \frac{f\left(3\,f''^2 - 2\,f'\,f'''\right)}{12\,f'^3}\right)^{-1} && (29.5\text{-}4f) \end{aligned}$$

The iteration $\Phi^+_{[n,0]}$ always coincides with Schröder's iteration. In general one obtains $n-1$ additional forms of iterations using the approximants $[0, n-2]$, $[1, n-3]$, $\ldots$, $[n-3, 1]$.

Neglecting terms that contain the third derivative in relation 29.5-4d we obtain the third order iteration

$$\Phi_3 \;=\; x - \frac{f\left(2\,f'^2 - f\,f''\right)}{f'\left(2\,f'^2 - 2\,f\,f''\right)} \qquad (29.5\text{-}5)$$

An iteration involving radicals

By directly solving the truncated Taylor expansion

$$f(r) \quad = \quad f(x) + f'(x)\,(r-x) + \frac{1}{2} f''(x)\,(r-x)^2 \tag{29.5-6}$$

of $f(r) = 0$ around x one obtains the following third order iteration:

$$\Phi_3 \quad = \quad x - \frac{1}{f''}\left(f' \pm \sqrt{f'^2 - 2ff''}\right) = x - \frac{f'}{f''}\left(1 \pm \sqrt{1 - 2\,\frac{f\,f''}{f'^2}}\right) \tag{29.5-7}$$

For $f(x) = a\,x^2 + b\,x + c$ it gives the two solutions of the quadratic equation $f(x) = 0$; for other functions one obtains an iterated square root expression for the roots.

The following form, given in [229, p.94], avoids possible cancellation:

$$\Phi_3 \quad = \quad x - \frac{2\,u}{1 + \sqrt{1 - 4\,A\,u}} \quad \text{where} \quad u = f/f' \quad \text{and} \quad A = f''/(2f) \tag{29.5-8}$$

It can be obtained by observing that

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad = \quad \frac{-2c}{b + \sqrt{b^2 - 4ac}} \tag{29.5-9}$$

Iterations from iterations

Alternative rational forms can also be obtained in a way that generalizes the method used for multiple roots: if we emphasize the so far notationally omitted dependency from the function f as $\Phi\{f\}$. The iteration $\Phi\{f\}$ has fixed points where f has a root r, so $x - \Phi\{f\}$ again has a root at r. Hence we can build more iterations that will converge to those roots as $\Phi\,\{x - \Phi\{f\}\}$. For dealing with multiple roots we used $\Phi\,\{x - \Phi_2\{f\}\}_k = \Phi\,\{f/f'\}$. An iteration $\Phi_k\,\{x - \Phi_j\{f\}\}$ can only be expected to have a k-th order convergence.

29.6 Improvements by the delta squared process

Given a sequence of partial sums x_k the so-called *delta squared process* computes a new sequence x_k^* of extrapolated sums:

$$x_k^* \quad = \quad x_{k+2} - \frac{(x_{k+2} - x_{k+1})^2}{x_{k+2} - 2\,x_{k+1} + x_k} \tag{29.6-1}$$

The method is due to Aitken. The name delta squared is due to the fact that the formula can be written symbolically as

$$x^* = x - \frac{(\Delta x)^2}{(\Delta^2 x)} \tag{29.6-2}$$

where Δ is the difference operator.

Note that the mathematically equivalent form

$$x_k^* \quad = \quad \frac{x_k\,x_{k+2} - x_{k+1}^2}{x_{k+2} - 2\,x_{k+1} + x_k} \tag{29.6-3}$$

sometimes given should be avoided with numerical computations due to possible cancellation.

If $x_k = \sum_{i=0}^{k} a_i$ and the ratio of consecutive summands a_i is approximately constant (that is, a is close to a geometric series) then x^* converges significantly faster to x_∞ than x. Let us partly rewrite the formula using $x_k - x_{k-1} = a_k$:

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} \tag{29.6-4}$$

Then for a geometric series with $a_{k+1}/a_k = q$

$$x_k^* = x_{k+2} - \frac{(a_{k+2})^2}{a_{k+2} - a_{k+1}} = x_{k+2} - \frac{\left(a_0\, q^{k+2}\right)^2}{a_0\,(q^{k+2} - q^{k+1})} \tag{29.6-5a}$$

$$= a_0\,\frac{1 - q^{k+3}}{1 - q} + a_0\, q^{k+2} \cdot \frac{q^{k+2}}{q^{k+1} - q^{k+2}} = \frac{a_0}{1 - q}\left(1 - q^{k+3} + q^{k+3}\right) \tag{29.6-5b}$$

$$= \frac{a_0}{1 - q} \tag{29.6-5c}$$

which is the exact sum. Now consider the sequence

$$x_0, \quad x_1 = \Phi(x_0), \quad x_2 = \Phi(x_1) = \Phi(\Phi(x_0)), \quad \ldots \tag{29.6-6}$$

of successively better approximations to some root r of a function f. Think of the x_k as partial sums of a series whose sum is the root r. Apply the idea to define an improved iteration Φ^* from a given one Φ:

$$\Phi^*(x) = \Phi(\Phi(x)) - \frac{[\Phi(\Phi(x)) - \Phi(x)]^2}{\Phi(\Phi(x)) - 2\,\Phi(x) + x} \tag{29.6-7}$$

The good news is that $\Phi*$ will give quadratic convergence even if Φ only has linear convergence. As an example, take $f(x) = (x^2 - d)^2$, forget that its root $\sqrt{d}$ is a double root and happily define $\Phi(x) = x - f(x)/f'(x) = x - (x^2 - d)/(4x)$. Convergence is only linear:

$$\Phi(\sqrt{d}\cdot(1 + e)) = \sqrt{d}\cdot\left(1 + \frac{e}{2} + \frac{e^2}{4} + O(e^3)\right) \tag{29.6-8}$$

Then try

$$\Phi^*(x) = \frac{d\,(7\,x^2 + d)}{x\,(3\,x^2 + 5\,d)} \tag{29.6-9}$$

and find that it has quadratic convergence

$$\Phi(\sqrt{d}\cdot(1 + e)) = \sqrt{d}\cdot\left(1 - \frac{e^2}{4} + \frac{e^3}{16} + O(e^4)\right) \tag{29.6-10}$$

In general, if Φ_n has convergence of order $n > 1$ then Φ_n^* will be of order $2n - 1$, but linear convergence ($n = 1$) is turned into second order, see [137, p.165].

Chapter 30

The arithmetic-geometric mean (AGM)

The arithmetic-geometric mean (AGM) is the basis for fast algorithms for the computation of π to high precision. We give AGM based algorithms for the computation of certain hypergeometric functions. AGM-based algorithms for the computation of the logarithm are given in section 31.1.1 on page 597, and for the exponential function in section 31.2.1 on page 603.

30.1 The AGM

The *arithmetic-geometric mean* (AGM) plays a central role in the high precision computation of logarithms and π. The AGM(a, b) is defined as the limit of the iteration

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{30.1-1a}$$

$$b_{k+1} = \sqrt{a_k\, b_k} \tag{30.1-1b}$$

starting with $a_0 = a$ and $b_0 = b$. Both of the values converge quadratically to a common limit. The related quantity c_k used in many AGM based computations is defined as

$$c_k^2 = a_k^2 - b_k^2 = (a_{k-1} - a_k)^2 \tag{30.1-2}$$

An alternative way for the computation for the AGM iteration is

$$c_{k+1} = \frac{a_k - b_k}{2} \tag{30.1-3a}$$

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{30.1-3b}$$

$$b_{k+1} = \sqrt{a_{k+1}^2 - c_{k+1}^2} \tag{30.1-3c}$$

Schönhage gives the most economic variant of the AGM, which, apart from the square root, only needs

one squaring per step:

$$\begin{aligned}
A_0 &= a_0^2 && (30.1\text{-}4a)\\
B_0 &= b_0^2 && (30.1\text{-}4b)\\
t_0 &= 1-(A_0-B_0) && (30.1\text{-}4c)\\
S_k &= \frac{A_k+B_k}{4} && (30.1\text{-}4d)\\
b_k &= \sqrt{B_k} \qquad \text{[square root]} && (30.1\text{-}4e)\\
a_{k+1} &= \frac{a_k+b_k}{2} && (30.1\text{-}4f)\\
A_{k+1} &= a_{k+1}^2 \qquad \text{[squaring]} && (30.1\text{-}4g)\\
&= \left(\frac{\sqrt{A_k}+\sqrt{B_k}}{2}\right)^2 = \frac{A_k+B_k}{4}+\frac{\sqrt{A_k\,B_k}}{2} && (30.1\text{-}4h)\\
B_{k+1} &= 2\,(A_{k+1}-S_k) \;=\; b_{k+1}^2 && (30.1\text{-}4i)\\
c_{k+1}^2 &= A_{k+1}-B_{k+1} \;=\; a_{k+1}^2-b_{k+1}^2 && (30.1\text{-}4j)\\
t_{k+1} &= t_k-2^{k+1}\,c_{k+1}^2 && (30.1\text{-}4k)
\end{aligned}$$

Starting with $a_0=A_0=1$, $B_0=1/2$ one has $\pi\approx(2\,a_n^2)/t_n$. The importance of the AGM is related to the fact that it can be used to compute certain hypergeometric functions fast. Indeed, one has

$$F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|1-\frac{b^2}{a^2}\right) \;=\; \frac{a}{\mathrm{AGM}(a,b)} \;=\; \frac{1}{\mathrm{AGM}(1,b/a)} \tag{30.1-5}$$

The relation is usually written as

$$F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|k\right) \;=\; \frac{1}{\mathrm{AGM}(1,\sqrt{1-k})} \tag{30.1-6}$$

corresponding to $\mathrm{AGM}(1,k)$, that is

$$a_0=1, \qquad b_0=k, \quad \text{and} \quad c_0=\sqrt{1-k^2} \tag{30.1-7}$$

The quantity

$$R'(k) \;:=\; 1-\frac{1}{2}\sum_{n=0}^{\infty}2^n c_n^2 \tag{30.1-8}$$

together with the AGM leads to a fast algorithm for the function $F\left(\left.\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\right|k\right)$, see section 30.2.

Combining two steps of the AGM iteration leads to the fourth order AGM iteration:

$$\begin{aligned}
\alpha_0 &= \sqrt{a_0} && (30.1\text{-}9a)\\
\beta_0 &= \sqrt{b_0} && (30.1\text{-}9b)\\
\alpha_{k+1} &= \frac{\alpha_k+\beta_k}{2} && (30.1\text{-}9c)\\
\beta_{k+1} &= \left(\frac{\alpha_k\,\beta_k\,(\alpha_k^2+\beta_k^2)}{2}\right)^{1/4} && (30.1\text{-}9d)\\
\gamma_k^4 &= \alpha_k^4-\beta_k^4 \qquad = c_{2k}^2 && (30.1\text{-}9e)
\end{aligned}$$

We have $\alpha_k = \sqrt{a_{2k}}$, $\beta_k = \sqrt{b_{2k}}$. An alternative formulation of the iteration is:

$$\gamma_{k+1} = \frac{\alpha_k - \beta_k}{2} \tag{30.1-10a}$$

$$\alpha_{k+1} = \frac{\alpha_k + \beta_k}{2} \tag{30.1-10b}$$

$$\beta_{k+1} = \left(\alpha_{k+1}^4 - \gamma_{k+1}^4\right)^{1/4} \tag{30.1-10c}$$

$$c_{2k}^2 + 2\,c_{2k+1}^2 = \alpha_{k-1}^4 - \left(\alpha_k^2 - \gamma_k^2\right)^2 \tag{30.1-10d}$$

Compute R' via

$$R'(k) = 1 - \sum_{n=0}^{\infty} 4^n \left(\alpha_n^4 - \left(\frac{\alpha_n^2 + \beta_n^2}{2}\right)^2\right) \tag{30.1-11}$$

30.2 The elliptic functions K and E

The elliptic functions $K(k)$ and $E(k)$ can be computed via the AGM which gives super-linear convergence. The logarithmic singularity of $K(k)$ at the point $k = 1$ (relation 30.2-4, see also relation 31.1-1a on page 597) is the key to the fast computation of the logarithm. The exponential function could be computed by inverting the logarithm but also as described in section 31.2.1 on page 603. For computations with very high precision the algorithms based on the elliptic functions are the fastest known today for the logarithm, the number π, and the exponential function.

30.2.1 Elliptic K

The function K can be defined as

$$K(k) = \int_0^{\pi/2} \frac{d\,\vartheta}{\sqrt{1 - k^2\,\sin^2\vartheta}} = \int_0^1 \frac{d\,t}{\sqrt{(1-t^2)\,(1-k^2t^2)}} \tag{30.2-1}$$

One has

$$K(k) = \frac{\pi}{2}\,F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,k^2\right) \tag{30.2-2a}$$

$$= \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2 k^{2i} = \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{\binom{2\,i}{i}}{4^i}\right)^2 k^{2i} \tag{30.2-2b}$$

$$= \frac{\pi}{2}\left(1 + \left(\frac{1}{2}\right)^2 k^2 + \left(\frac{1\cdot 3}{2\cdot 4}\right)^2 k^4 + \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)^2 k^6 + \ldots\right) \tag{30.2-2c}$$

$$= \frac{\pi}{2}\left(1 + \frac{1}{4}\,k^2 + \frac{9}{64}\,k^4 + \frac{25}{256}\,k^6 + \frac{1225}{16384}\,k^8 + \frac{3969}{65536}\,k^{10} + \ldots\right) \tag{30.2-2d}$$

See section 35.2.7 on page 677 for transformations in terms of hypergeometric functions. Special values are $K(0) = \frac{\pi}{2}$ and $\lim_{k\to 1-} K(k) = +\infty$.

The computational interesting form is $F\left(\begin{matrix}\frac{1}{2},\frac{1}{2}\\ 1\end{matrix}\,\middle|\,z\right) = 1/\operatorname{AGM}(1,\sqrt{1-z})$ (see section 30.1 on page 573):

$$K(k) = \frac{\pi}{2\,\operatorname{AGM}(1,k')} = \frac{\pi}{2\,\operatorname{AGM}(1,\sqrt{1-k^2})} \tag{30.2-3a}$$

One defines $k' = \sqrt{1-k^2}$ and $K'(k)$ as $K(k')$:

$$K'(k) := K(\sqrt{1-k^2}) = \frac{\pi}{2\,\operatorname{AGM}(1,k)} \tag{30.2-3b}$$

A C++ implementation of the AGM based computation is given in [hfloat: src/tz/elliptic-k.cc].

For k close to 1 we have

$$K(k) \quad \approx \quad \log \frac{4}{\sqrt{1-k^2}} \tag{30.2-4}$$

The following estimate is given in [52, p.11]:

$$\left| K'(k) - \log \frac{4}{k} \right| \quad \leq \quad 4\,k^2\,(8 + \log k) \quad \text{where} \quad 0 < k \leq 1 \tag{30.2-5}$$

Product forms for K and K' that are also candidates for fast computations are, for $0 < k_0 \leq 1$,

$$\frac{2}{\pi}\,K'(k_0) \quad = \quad \prod_{n=0}^{\infty} \frac{2}{1+k_n} \;=\; \prod_{n=1}^{\infty} 1 + k'_n \quad \text{where} \quad k_{n+1} \;:=\; \frac{2\sqrt{k_n}}{1+k_n}, \quad k_\infty = 1 \tag{30.2-6a}$$

$$\frac{2}{\pi}\,K'(k_0) \quad = \quad \prod_{n=0}^{\infty} \frac{1}{\sqrt{k_n}} \quad \text{where} \quad k_{n+1} \;:=\; \frac{1+k_n}{2\sqrt{k_n}}, \quad k_\infty = 1 \tag{30.2-6b}$$

The second form is computationally especially attractive since, apart from the multiplication with the main product, only a inverse square root needs to be computed per step. The product formulas follow directly from relation 30.2-3b (and $\mathrm{AGM}(a,b) = a\ \mathrm{AGM}(1,b/a) = b\ \mathrm{AGM}(a/b,1)$):

$$\frac{1}{\mathrm{AGM}(1,k)} \quad = \quad \left[\mathrm{AGM}\left(\frac{1+k}{2}, \sqrt{k}\right)\right]^{-1} \tag{30.2-7a}$$

$$= \quad \left[\frac{1+k}{2}\ \mathrm{AGM}\left(1, \frac{2\sqrt{k}}{1+k}\right)\right]^{-1} \qquad \text{(first form)} \tag{30.2-7b}$$

$$= \quad \left[\sqrt{k}\ \mathrm{AGM}\left(\frac{1+k}{2\sqrt{k}}, 1\right)\right]^{-1} \qquad \text{(second form)} \tag{30.2-7c}$$

Similarly, for $0 < k_0 \leq 1$,

$$\frac{2}{\pi}\,K(k_0) \quad = \quad \prod_{n=0}^{\infty} \frac{2}{1+k'_n} \;=\; \prod_{n=1}^{\infty} 1 + k_n \quad \text{where} \quad k_{n+1} \;:=\; \frac{1-k'_n}{1+k'_n}, \quad k_\infty = 0 \tag{30.2-8}$$

Certain values of the gamma function can be expressed in K (taken from [233, p.12]):

$$\Gamma\left(\frac{1}{3}\right) \quad = \quad \frac{\pi^{1/3}\,2^{7/9}}{3^{1/12}}\,K\left(\frac{\sqrt{3}-1}{2\sqrt{2}}\right)^{1/3} \tag{30.2-9a}$$

$$\Gamma\left(\frac{1}{4}\right) \quad = \quad \pi^{1/4}\,2\,K\left(\frac{1}{\sqrt{2}}\right)^{1/2} \tag{30.2-9b}$$

$$\Gamma\left(\frac{1}{8}\right) \quad = \quad \pi^{1/8}\,2^{17/8}\,K\left(\frac{1}{\sqrt{2}}\right)^{1/4} K\left(\sqrt{2}-1\right)^{1/2} \tag{30.2-9c}$$

30.2.2 Elliptic E

The function E can be defined as

$$E(k) \quad = \quad \int_0^{\pi/2} \sqrt{1 - k^2\,\sin^2\vartheta}\,d\,\vartheta \;=\; \int_0^1 \frac{\sqrt{1-k^2t^2}}{\sqrt{1-t^2}}\,d\,t \tag{30.2-10}$$

One has

$$E(k) = \frac{\pi}{2}\,F\left(\begin{matrix}-\frac{1}{2},\frac{1}{2}\\1\end{matrix}\,\middle|\,k^2\right) \tag{30.2-11a}$$

$$= \frac{\pi}{2}\left(-\sum_{i=0}^{\infty}\left(\frac{(2\,i-1)!!}{2^i\,i!}\right)^2\frac{k^{2i}}{2i-1}\right) = \frac{\pi}{2}\sum_{i=0}^{\infty}\left(\frac{\binom{2\,i}{i}}{4^i}\right)^2\frac{k^{2i}}{2\,i-1} \tag{30.2-11b}$$

$$= \frac{\pi}{2}\left(1-\left(\frac{1}{2}\right)^2k^2-\left(\frac{1\cdot3}{2\cdot4}\right)^2\frac{k^4}{3}-\left(\frac{1\cdot3\cdot5}{2\cdot4\cdot6}\right)^2\frac{k^6}{5}-\ldots\right) \tag{30.2-11c}$$

$$= \frac{\pi}{2}\left(1-\frac{1}{4}\,k^2-\frac{3}{64}\,k^4-\frac{5}{256}\,k^6-\frac{175}{16384}\,k^8-\frac{441}{65536}\,k^{10}-\ldots\right) \tag{30.2-11d}$$

Special values are $E(0)=\frac{\pi}{2}$ and $E(1)=1$. The latter leads to a (slowly converging) series for $2/\pi$:

$$\frac{2}{\pi} = F\left(\begin{matrix}-\frac{1}{2},\frac{1}{2}\\1\end{matrix}\,\middle|\,1\right) \tag{30.2-12}$$

Similarly as for K', one defines E' as

$$E'(k) := E(k') = E(\sqrt{1-k^2}) \tag{30.2-13}$$

The key to fast computation of E is the relation

$$\frac{E}{K} = 1-\frac{1}{2}\sum_{n=0}^{\infty}2^n{c'}_n^2 \tag{30.2-14}$$

The terms c' in the sum occur naturally during the computation of the AGM, see relation 30.1-2 on page 573. One defines

$$R := \frac{E}{K}, \qquad R' := \frac{E'}{K'} \tag{30.2-15}$$

Then E can be computed via

$$E(k) = R(k)\,K(k) = \frac{\pi}{2\ \mathrm{AGM}(1,\sqrt{1-k^2})}\cdot\left(1-\sum_{n=0}^{\infty}2^{n-1}{c'}_n^2\right) \tag{30.2-16}$$

Legendre's relation between K and E is (arguments omitted for readability, choose your favorite form):

$$\frac{E}{K}+\frac{E'}{K'}-1 = \frac{\pi}{2\,K\,K'} \tag{30.2-17a}$$

$$E\,K'+E'\,K-K\,K' = \frac{\pi}{2} \tag{30.2-17b}$$

Equivalently,

$$\mathrm{AGM}(1,k) = \frac{E/K}{(1-E'/K')} = \frac{R}{(1-R')} \tag{30.2-18}$$

For $k=\frac{1}{\sqrt{2}}=:s$ we have $k=k'$, thereby $K=K'$ and $E=E'$, so

$$\frac{K(s)}{\pi}\left(\frac{2\,E(s)}{\pi}-\frac{K(s)}{\pi}\right) = \frac{1}{2\,\pi} \tag{30.2-19}$$

As expressions 30.2-3a and 30.2-16 provide a fast AGM based computation of $\frac{K}{\pi}$ and $\frac{E}{\pi}$ the above formula can be used to compute π.

Using $E - K = k\,k'\,\frac{dK}{dk} - k^2\,K$ one can express the derivative of K in terms of E and K and thereby compute that quantity fast:

$$\frac{dK}{dk} = \frac{E - k'^2\,K}{k\,k'^2} \tag{30.2-20}$$

For the derivative of E we have

$$\frac{dE}{dk} = \frac{E - K}{k} \tag{30.2-21}$$

We note the following generalization of Legendre's relation in terms of hypergeometric functions (see section 35.2 on page 663):

$$\begin{aligned}\frac{\Gamma\left(1+a+b\right)\,\Gamma\left(1+c+b\right)}{\Gamma\left(\frac{3}{2}+a+b+c\right)\,\Gamma\left(\frac{1}{2}+b\right)} = \\ +F\left(\begin{matrix}+\frac{1}{2}+a,\ -\frac{1}{2}-c\\ 1+a+b\end{matrix}\middle|\,z\right)\ F\left(\begin{matrix}+\frac{1}{2}-a,\ +\frac{1}{2}+c\\ 1+c+b\end{matrix}\middle|\,1-z\right)+ \\ +F\left(\begin{matrix}+\frac{1}{2}+a,\ +\frac{1}{2}-c\\ 1+a+b\end{matrix}\middle|\,z\right)\ F\left(\begin{matrix}-\frac{1}{2}-a,\ +\frac{1}{2}+c\\ 1+c+b\end{matrix}\middle|\,1-z\right)- \\ -F\left(\begin{matrix}+\frac{1}{2}-a,\ +\frac{1}{2}-c\\ 1+a+b\end{matrix}\middle|\,z\right)\ F\left(\begin{matrix}+\frac{1}{2}-a,\ +\frac{1}{2}+c\\ 1+c+b\end{matrix}\middle|\,1-z\right)\end{aligned} \tag{30.2-22}$$

This equation is given in [15, p.138]. For $a = b = c = 0$ one obtains Legendre's relation.

30.3 AGM-type algorithms for hypergeometric functions

We give AGM based algorithms for $F\left(\begin{matrix}1/2-s,\,1/2+s\\ 1\end{matrix}\middle|\,z\right)$ where $s \in \{0,\,1/6,\,1/4,\,1/3\}$, and $F\left(\begin{matrix}1/4-t,\,1/4+t\\ 1\end{matrix}\middle|\,z\right)$ where $t \in \{1/12,\,1/6\}$. These are taken from [55] and [117], both papers are recommended for further studies. See also [175], [54], [75], and [74]. The limit of a three-term iteration as a generalized hypergeometric function is determined in [159]. A four-term iteration is considered in [53].

The following transformations can be applied to the functions, these are special cases of relations 35.2-31a and 35.2-31b on page 667:

$$F\left(\begin{matrix}\frac{1}{2}+s,\ \frac{1}{2}-s\\ 1\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}\frac{1}{4}+\frac{s}{2},\ \frac{1}{4}-\frac{s}{2}\\ 1\end{matrix}\middle|\,4z(1-z)\right) \quad \text{where} \quad |z| < \frac{1}{2} \tag{30.3-1a}$$

$$F\left(\begin{matrix}\frac{1}{4}+t,\ \frac{1}{4}-t\\ 1\end{matrix}\middle|\,z\right) = F\left(\begin{matrix}\frac{1}{2}+2t,\ \frac{1}{2}-2t\\ 1\end{matrix}\middle|\,\frac{1-\sqrt{1-z}}{2}\right) \tag{30.3-1b}$$

Algorithms for $F\left(\begin{matrix}1/2,\,1/2\\ 1\end{matrix}\middle|\,z\right)$ $[1/2 \pm 0]$

The following is relation 30.2-3a on page 575, the classical AGM algorithm which has quadratic convergence:

$$F\left(\begin{matrix}1/2,\ 1/2\\ 1\end{matrix}\middle|\,z\right) = 1/M\left(1,\,\sqrt{1-z}\right) \quad \text{where} \tag{30.3-2a}$$

$$M(a,b) := \left[(a+b)/2,\,\sqrt{a\,b}\right] \tag{30.3-2b}$$

We write the AGM as $M := [f(a,b),\,g(a,b)]$ in the obvious way.

A fourth order algorithm obtained by combining two steps of the classical AGM:

$$F\left(\begin{matrix}1/2,\,1/2\\1\end{matrix}\middle|\,z\right) \;=\; 1/M\left(1,\,\sqrt[4]{1-z}\right)^2 \quad \text{where} \tag{30.3-3a}$$

$$M(a,b) \;:=\; \left[(a+b)/2,\;\sqrt{a\,b\,(a^2+b^2)/2}\right] \tag{30.3-3b}$$

For comparison, we give the quadratic transform for the hypergeometric function

$$F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\1\end{matrix}\middle|\,z'\right) \;=\; (1+z)\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\1\end{matrix}\middle|\,z^2\right) \tag{30.3-4a}$$

where

$$z \;=\; \frac{1-(1-z')^{1/2}}{1+(1-z')^{1/2}} \tag{30.3-4b}$$

$$z' \;=\; 1-\left(\frac{1-z}{1+z}\right)^2 \tag{30.3-4c}$$

It is the special case $a=1/2$ and $b=1/2$ of the transformation

$$F\left(\begin{matrix}a,\,b\\2b\end{matrix}\middle|\,\frac{4z}{(1+z)^2}\right) \;=\; F\left(\begin{matrix}a,\,b\\2b\end{matrix}\middle|\,1-\left(\frac{1-z}{1+z}\right)^2\right) \;=\; (1+z)^{2a}\,F\left(\begin{matrix}a,\;a-b+\frac{1}{2}\\b+\frac{1}{2}\end{matrix}\middle|\,z^2\right) \tag{30.3-5}$$

Algorithms for $F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right)$ $[1/2 \pm 1/6]$

A third order algorithm:

$$F\left(\begin{matrix}1/3,\,1/3\\1\end{matrix}\middle|\,z\right) \;=\; 1/M\left(\sqrt[3]{1-z},\,1\right) \;=\; \sqrt[3]{1-z}\,F\left(\begin{matrix}2/3,\,2/3\\1\end{matrix}\middle|\,z\right) \quad \text{where} \tag{30.3-6a}$$

$$M(a,b) \;:=\; \left[(a+2\,b)/3,\;\sqrt[3]{b\,(a^2+a\,b+b^2)\,/3}\right] \tag{30.3-6b}$$

One further has

$$F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right) \;=\; 1/M\left(1,\,\sqrt[3]{1-z}\right) \tag{30.3-7}$$

A quadratic algorithm:

$$F\left(\begin{matrix}1/3,\,2/3\\1\end{matrix}\middle|\,z\right) \;=\; 1/M\left(1,\,\sqrt[3]{1-z}\right) \quad \text{where} \tag{30.3-8a}$$

$$M(a,b) \;:=\; \left[\frac{1}{2}\left(\sqrt[3]{2p-a^3}+\sqrt[3]{2m-a^3}\right),\;\frac{1}{2}\left(\sqrt[3]{p}+\sqrt[3]{m}\right)\right] \qquad \text{and} \tag{30.3-8b}$$

$$p \;:=\; b^3+t, \qquad m \;:=\; b^3-t, \qquad t \;:=\; \sqrt{b^6-a^3\,b^3} \tag{30.3-8c}$$

And again (see relation 30.3-6a):

$$F\left(\begin{matrix}1/3,\,1/3\\1\end{matrix}\middle|\,z\right) \;=\; 1/M\left(\sqrt[3]{1-z},\,1\right) \;=\; \sqrt[3]{1-z}\,F\left(\begin{matrix}2/3,\,2/3\\1\end{matrix}\middle|\,z\right) \tag{30.3-9}$$

We note the following hypergeometric transformation due to Ramanujan:

$$F\left(\begin{matrix}\frac{1}{3},\,\frac{2}{3}\\1\end{matrix}\middle|\,z'\right) \;=\; (1+2\,z)\,F\left(\begin{matrix}\frac{1}{3},\,\frac{2}{3}\\1\end{matrix}\middle|\,z^3\right) \tag{30.3-10a}$$

where

$$z = \frac{1-(1-z')^{1/3}}{1+2\,(1-z')^{1/3}} \tag{30.3-10b}$$

$$z' = 1-\left(\frac{1-z}{1+2z}\right)^3 \tag{30.3-10c}$$

The general form is given in [36]:

$$F\left(\begin{matrix}c,\ c+\frac{1}{3}\\ \frac{3c+1}{2}\end{matrix}\middle|\ 1-\left(\frac{1-z}{1+2z}\right)^3\right) = (1+2z)^{3c}\,F\left(\begin{matrix}c,\ c+\frac{1}{3}\\ \frac{3c+5}{6}\end{matrix}\middle|\ z^3\right) \tag{30.3-11}$$

For $c = 1/3$ one obtains relation 30.3-10a. A computer algebra proof that relation 30.3-11 is the only possible generalization of relation 30.3-10c is given in [158].

An alternative quadratic algorithm is

$$F\left(\begin{matrix}1/3,\ 2/3\\ 1\end{matrix}\middle|\ z\right) = 1/M\,(1,\,W) \quad \text{where} \tag{30.3-12a}$$

$$M(a,b) := \left[(a+b)/2,\ \left(3\,\sqrt{b\,(b+2a)/3}-b\right)/2\right] \quad \text{and} \tag{30.3-12b}$$

$$W := \frac{1-R+R^2}{R},\quad R := \left[\sqrt{u^2-1}+u\right]^{1/3},\quad u := 1-2\,z \tag{30.3-12c}$$

It is given in the form

$$F\left(\begin{matrix}1/3,\ 2/3\\ 1\end{matrix}\middle|\ (1-x)\,(1+x/2)^2\right) = 1/M\,(1,\,x) \tag{30.3-13}$$

A product form can be derived from [35, Theorem 6.1]: Let

$$\alpha(z) := \frac{z\,(3+z)^2}{2\,(1+z)^3} \tag{30.3-14a}$$

$$p(z) := \frac{r^2-r+1}{r} \quad \text{where} \quad r := \left[2z+2\sqrt{z^2-z}-1\right]^{1/3} \tag{30.3-14b}$$

then, with $t_0 := 1-z$ and $t_{k+1} := \alpha(p(t_k))$,

$$F\left(\begin{matrix}1/3,\ 2/3\\ 1\end{matrix}\middle|\ z\right) = \left[\prod_{k=0}^{\infty}\frac{1+p(t_k)}{2}\right]^{-1} \tag{30.3-14c}$$

Convergence is quadratic. The function $p(z)$ is the real solution of $p(\beta(z)) = z$ where $\beta(z) := (z^2(3+z))/4$.

Algorithms for $F\left(\begin{matrix}1/4,\,3/4\\ 1\end{matrix}\middle|\ z\right)$ $[1/2 \pm 1/4]$

A quadratic algorithm:

$$F\left(\begin{matrix}1/4,\ 3/4\\ 1\end{matrix}\middle|\ z\right) = 1/M\left(1,\,\sqrt{1-z}\right)^{1/2} \quad \text{where} \tag{30.3-15a}$$

$$M(a,b) := \left[(a+3b)/4,\ \sqrt{b\,(a+b)/2}\right] \tag{30.3-15b}$$

One further has (note the swapped arguments in the mean)

$$F\left(\begin{matrix}1/4,\ 1/4\\ 1\end{matrix}\middle|\ z\right) = 1/M\left(\sqrt{1-z},\,1\right)^{1/2} = \sqrt{1-z}\,F\left(\begin{matrix}3/4,\ 3/4\\ 1\end{matrix}\middle|\ z\right) \tag{30.3-16}$$

Now set $A_k := \sqrt{(a_k + b_k)/2}$ and $B_k := \sqrt{b_k}$, then

$$A_{k+1} = \frac{1}{2}\,(A_k + B_k) \tag{30.3-17a}$$

$$B_{k+1} = \sqrt{A_k\,B_k} \tag{30.3-17b}$$

This is the iteration of the classical AGM. Thereby

$$M(a,b)^{1/2} = \text{AGM}\left(\sqrt{\frac{a+b}{2}},\,\sqrt{b}\right) \tag{30.3-18}$$

and one can employ the AGM scheme of Schönhage (relations 30.1-4a… 30.1-4k on page 574). Equivalently,

$$F\left(\begin{matrix}1/4,\,3/4\\1\end{matrix}\middle|\,z\right) = 1/\,\text{AGM}\left(\sqrt{\frac{1+\sqrt{1-z}}{2}},\,\sqrt[4]{1-z}\right) \tag{30.3-19}$$

We also have

$$F\left(\begin{matrix}1/4,\,1/4\\1\end{matrix}\middle|\,z\right) = 1/\,\text{AGM}\left(\sqrt{\frac{1+\sqrt{1-z}}{2}},\,1\right) \tag{30.3-20}$$

For comparison, we give the hypergeometric transformation

$$F\left(\begin{matrix}\frac{1}{4},\,\frac{3}{4}\\1\end{matrix}\middle|\,z'\right) = \sqrt{1+3\,z}\;F\left(\begin{matrix}\frac{1}{4},\,\frac{3}{4}\\1\end{matrix}\middle|\,z^2\right) \tag{30.3-21a}$$

where

$$z = \frac{1-(1-z')^{1/2}}{1+3\,(1-z')^{1/2}} \tag{30.3-21b}$$

$$z' = 1-\left(\frac{1-z}{1+3\,z}\right)^2 \tag{30.3-21c}$$

is the special case $d = 1/4$ of the transformation

$$F\left(\begin{matrix}d,\,d+\frac{1}{2}\\\frac{4d+2}{3}\end{matrix}\middle|\,z'\right) = (1+3\,z)^{2d}\;F\left(\begin{matrix}d,\,d+\frac{1}{2}\\\frac{4d+5}{6}\end{matrix}\middle|\,z^2\right) \tag{30.3-22}$$

This is taken from [118] where various such transformations and their generalizations are given.

Algorithm for $F\left(\begin{matrix}1/6,\,1/3\\1\end{matrix}\middle|\,z\right)$ $[1/4 \pm 1/12]$

A quadratic algorithm is

$$F\left(\begin{matrix}1/6,\,1/3\\1\end{matrix}\middle|\,z\right) = 1/M\,(1,\,W)^{1/2} \quad \text{where} \tag{30.3-23a}$$

$$M(a,b) := \Big[(a+3b)/4,\,(\sqrt{ab}+b)/2\Big] \qquad \text{and} \tag{30.3-23b}$$

$$W := \frac{1+R+R^2}{3\,R},\quad R := \Big[\sqrt{z^2-1}+z\Big]^{1/3} \tag{30.3-23c}$$

It is given [55, p.515] as

$$\left[F\left(\begin{matrix}1/6,\,1/3\\1\end{matrix}\middle|\,27\,x^2\,(1-x)/4\right)\right]^2 = 1/M\,(1,\,x) \tag{30.3-24}$$

One can solve for the argument of F to obtain the explicit form.

Algorithm for $F\left(\begin{matrix}1/12,\,5/12\\1\end{matrix}\middle|\,z\right)$ $[1/4 \pm 1/6]$

The following algorithm has quadratic convergence, W is defined by relation 30.3-23c:

$$F\left(\begin{matrix}1/12,\,5/12\\1\end{matrix}\middle|\,z\right) = 1/M\left(1,\,W\right)^{1/4} \quad \text{where} \tag{30.3-25a}$$

$$M(a,b) := \left[(a+15b)/16,\,\left(\sqrt{b\,(a+3b)/4}+b\right)/2\right] \tag{30.3-25b}$$

The next relation is the special case $a = 1/6$ of relation 35.2-40e on page 669:

$$F\left(\begin{matrix}\frac{1}{6},\,\frac{1}{6}\\1\end{matrix}\middle|\,z\right) = (1-z)^{-1/6}\,F\left(\begin{matrix}\frac{1}{12},\,\frac{5}{12}\\1\end{matrix}\middle|\,\frac{-4\,z}{(1-z)^2}\right) \tag{30.3-26}$$

The following relations are given in [175, p.17]:

$$F\left(\begin{matrix}\frac{1}{4},\,\frac{1}{4}\\1\end{matrix}\middle|\,-\frac{z}{64}\right) = \left[\frac{1}{16^3}\,(z+16)^3\right]^{-1/12} F\left(\begin{matrix}\frac{1}{12},\,\frac{5}{12}\\1\end{matrix}\middle|\,\frac{1728\,z}{(z+16)^3}\right) \tag{30.3-27a}$$

$$F\left(\begin{matrix}\frac{1}{3},\,\frac{1}{3}\\1\end{matrix}\middle|\,-\frac{z}{27}\right) = \left[\frac{1}{3^6}\,(z+3)^3\,(z+27)\right]^{-1/12} F\left(\begin{matrix}\frac{1}{12},\,\frac{5}{12}\\1\end{matrix}\middle|\,\frac{1728\,z}{(z+3)^3\,(z+27)}\right) \tag{30.3-27b}$$

$$F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\1\end{matrix}\middle|\,-\frac{z}{16}\right) = \left[\frac{1}{16^3}\,(z^2+16z+16)^3\right]^{-1/12} F\left(\begin{matrix}\frac{1}{12},\,\frac{5}{12}\\1\end{matrix}\middle|\,\frac{1728\,z\,(z+16)}{(z^2+16z+16)^3}\right) \tag{30.3-27c}$$

30.4 Computation of π

We give various iterations for computing π with super-linear convergence. The number of full precision multiplications (FPM) is an indication of the efficiency of the algorithm. The approximate number of FPMs that were counted with a computation of π to 4 million decimal digits (using radix $10,000$ and 1 million LIMBs) is indicated like this: #FPM=123.4. The number is computed as three times the FFT-work in terms of full precision real valued FFTs.

30.4.1 Super-linear iterations for π

AGM implemented in [hfloat: src/pi/piagm.cc], #FPM=98.4:

$$a_0 = 1, \qquad b_0 = \frac{1}{\sqrt{2}} \tag{30.4-1a}$$

$$a_{k+1} = \frac{a_k + b_k}{2} \tag{30.4-1b}$$

$$b_{k+1} = \sqrt{a_k\,b_k} \tag{30.4-1c}$$

$$p_n = \frac{2\,a_{n+1}^2}{1-\sum_{k=0}^{n} 2^k\,c_k^2} \quad \to \pi \tag{30.4-1d}$$

$$\pi - p_n = \frac{\pi^2\,2^{n+4}\,e^{-\pi\,2^{n+1}}}{\mathrm{AGM}^2(a_0,b_0)} \tag{30.4-1e}$$

Convergence is second order. Computing π based on the fourth order AGM (relations 30.1-9a... 30.1-9e on page 574) is possible by setting the second argument of the routine (#FPM=149.3 for the quartic variant). Schönhage's variant of the AGM computation (relations 30.1-4a... 30.1-4k on page 574) is implemented in [hfloat: src/pi/piagmsch.cc] (#FPM=78.424).

The AGM method goes back to Gauss, a facsimile of the entry in his 1809 handbook 6 is given in [17, p.101]. The entry states that

$$\pi \quad = \quad \frac{\mathrm{AGM}(1,k)\ \mathrm{AGM}(1,k')}{1-\sum_{k=0}^{\infty} 2^{k-1}\,(c_k^2+c'^2_k)} \tag{30.4-2}$$

where $k' = b_0/a_0$ and $k = \sqrt{1-b_0^2/a_0^2} = c_0/a_0$. For $k = k' = 1/\sqrt{2}$ one obtains relation 30.4-1d. The formula appeared also 1924 in [150, p.39]. The algorithm was rediscovered 1976 independently by Brent [60] (reprinted in [34, p.424]) and Salamin [204] (reprinted in [34, p.418]).

AGM variant given in [50], [hfloat: src/pi/piagm3.cc], #FPM=99.5 (#FPM=155.3 for the quartic variant):

$$a_0 \quad = \quad 1, \qquad b_0 \;=\; \frac{\sqrt{6}+\sqrt{2}}{4} \tag{30.4-3a}$$

$$p_n \quad = \quad \frac{2\,a_{n+1}^2}{\sqrt{3}\,(1-\sum_{k=0}^{n} 2^k\,c_k^2)-1} \quad \to \pi \tag{30.4-3b}$$

$$\pi - p_n \quad < \quad \frac{\sqrt{3}\,\pi^2\,2^{n+4}\,e^{-\sqrt{3}\,\pi\,2^{n+1}}}{\mathrm{AGM}^2(a_0,b_0)} \tag{30.4-3c}$$

AGM variant given in [50], [hfloat: src/pi/piagm3.cc], #FPM=108.2 (#FPM=169.5 for the quartic variant):

$$a_0 \quad = \quad 1, \qquad b_0 \;=\; \frac{\sqrt{6}-\sqrt{2}}{4} \tag{30.4-4a}$$

$$p_n \quad = \quad \frac{6\,a_{n+1}^2}{\sqrt{3}\,(1-\sum_{k=0}^{n} 2^k\,c_k^2)+1} \quad \to \pi \tag{30.4-4b}$$

$$\pi - p_n \quad < \quad \frac{\frac{1}{\sqrt{3}}\,\pi^2\,2^{n+4}\,e^{-\frac{1}{\sqrt{3}}\,\pi\,2^{n+1}}}{\mathrm{AGM}(a_0,b_0)^2} \tag{30.4-4c}$$

Second order iteration from [52, p.170], [hfloat: src/pi/pi2nd.cc], #FPM=255.7:

$$y_0 \quad = \quad \frac{1}{\sqrt{2}}, \qquad a_0 \;=\; \frac{1}{2} \tag{30.4-5a}$$

$$y_{k+1} \quad = \quad \frac{1-(1-y_k^2)^{1/2}}{1+(1-y_k^2)^{1/2}} \quad \to 0+ \tag{30.4-5b}$$

$$\quad = \quad \frac{(1-y_k^2)^{-1/2}-1}{(1-y_k^2)^{-1/2}+1} \tag{30.4-5c}$$

$$a_{k+1} \quad = \quad a_k\,(1+y_{k+1})^2 - 2^{k+1}\,y_{k+1} \quad \to \frac{1}{\pi} \tag{30.4-5d}$$

$$a_k - \pi^{-1} \quad \le \quad 16\cdot 2^{k+1}\,e^{-2^{k+1}\,\pi} \tag{30.4-5e}$$

Relation 30.4-5c shows how to save 1 multiplication per step (see section 28.1 on page 541).

Borwein's quartic (fourth order) iteration from [52, p.170], variant $r = 4$, implemented in [hfloat:

src/pi/pi4th.cc], #FPM=170.5:

$$\begin{aligned}
y_0 &= \sqrt{2}-1, \qquad a_0 = 6-4\sqrt{2} && \text{(30.4-6a)}\\
y_{k+1} &= \frac{1-(1-y_k^4)^{1/4}}{1+(1-y_k^4)^{1/4}} \quad \to 0+ && \text{(30.4-6b)}\\
&= \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} && \text{(30.4-6c)}\\
a_{k+1} &= a_k\,(1+y_{k+1})^4 - 2^{2k+3}\,y_{k+1}\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} && \text{(30.4-6d)}\\
&= a_k\,((1+y_{k+1})^2)^2 - 2^{2k+3}\,y_{k+1}\,((1+y_{k+1})^2-y_{k+1}) && \text{(30.4-6e)}\\
0 &< a_k-\pi^{-1} \le 16\cdot 4^n\,2\,e^{-4^n\,2\,\pi} && \text{(30.4-6f)}
\end{aligned}$$

Identities 30.4-6c and 30.4-6e show how to save operations.

Borwein's quartic (fourth order) iteration, variant $r = 16$, implemented in [hfloat: src/pi/pi4th.cc], #FPM=164.4:

$$\begin{aligned}
y_0 &= \frac{1-2^{-1/4}}{1+2^{-1/4}}, \qquad a_0 = \frac{8/\sqrt{2}-2}{(2^{-1/4}+1)^4} && \text{(30.4-7a)}\\
y_{k+1} &= \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} \quad \to 0+ && \text{(30.4-7b)}\\
a_{k+1} &= a_k\,(1+y_{k+1})^4 - 2^{2k+4}\,y_{k+1}\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} && \text{(30.4-7c)}\\
0 &< a_k-\pi^{-1} \le 16\cdot 4^n\,4\,e^{-4^n\,4\,\pi} && \text{(30.4-7d)}
\end{aligned}$$

The operation count is unchanged, but this variant gives approximately twice as much precision after the same number of steps. The general form of the quartic iterations (relations 30.4-6a... and 30.4-7a...) is given in [52, pp.170ff]:

$$\begin{aligned}
y_0 &= \sqrt{\lambda^*(r)}, \qquad a_0 = \alpha(r) && \text{(30.4-8a)}\\
y_{k+1} &= \frac{(1-y_k^4)^{-1/4}-1}{(1-y_k^4)^{-1/4}+1} \quad \to 0+ && \text{(30.4-8b)}\\
a_{k+1} &= a_k\,(1+y_{k+1})^4 - 2^{2k+2}\,\sqrt{r}\,y_k\,(1+y_{k+1}+y_{k+1}^2) \quad \to \frac{1}{\pi} && \text{(30.4-8c)}\\
0 &< a_k-\pi^{-1} \le \quad 16\cdot 4^n\,\sqrt{r}\,e^{-4^n\,\sqrt{r}\,\pi} && \text{(30.4-8d)}
\end{aligned}$$

Derived AGM iteration (second order, from [52, pp.46ff]), implemented in [hfloat: src/pi/pideriv.cc], #FPM=276.2:

$$\begin{aligned}
x_0 &= \sqrt{2}, \qquad p_0 = 2+\sqrt{2}, \qquad y_1 = 2^{1/4} && \text{(30.4-9a)}\\
x_{k+1} &= \frac{1}{2}\left(\sqrt{x_k}+\frac{1}{\sqrt{x_k}}\right) \quad (k\ge 0) \quad \to 1+ && \text{(30.4-9b)}\\
y_{k+1} &= \frac{y_k\,\sqrt{x_k}+\dfrac{1}{\sqrt{x_k}}}{y_k+1} \quad (k\ge 1) \quad \to 1+ && \text{(30.4-9c)}\\
p_{k+1} &= p_k\,\frac{x_k+1}{y_k+1} \quad (k\ge 1) \quad \to \pi+ && \text{(30.4-9d)}\\
p_k-\pi &< 10^{-2^{k+1}} && \text{(30.4-9e)}
\end{aligned}$$

Cubic AGM iteration (third order) from [56], implemented in [hfloat: src/pi/picubagm.cc], #FPM=182.7:

$$\begin{aligned}
a_0 &= 1, \qquad b_0 = \frac{\sqrt{3}-1}{2} && \text{(30.4-10a)}\\
a_{n+1} &= \frac{a_n + 2\,b_n}{3} && \text{(30.4-10b)}\\
b_{n+1} &= \sqrt[3]{\frac{b_n\,(a_n^2 + a_n\,b_n + b_n^2)}{3}} && \text{(30.4-10c)}\\
p_n &= \frac{3\,a_n^2}{1 - \sum_{k=0}^{n} 3^k\,(a_k^2 - a_{k+1}^2)} \quad \to \pi && \text{(30.4-10d)}
\end{aligned}$$

Quintic (5th order) iteration from [52, p.310], [hfloat: src/pi/pi5th.cc], #FPM=353.2:

$$\begin{aligned}
s_0 &= 5(\sqrt{5}-2), \qquad a_0 = \frac{1}{2} && \text{(30.4-11a)}\\
x &= \frac{5}{s_n} - 1 \quad \to 4 && \text{(30.4-11b)}\\
y &= (x-1)^2 + 7 \quad \to 16 && \text{(30.4-11c)}\\
z &= \left(\frac{x}{2}\,\left(y + \sqrt{y^2 - 4x^3}\right)\right)^{1/5} \quad \to 2 && \text{(30.4-11d)}\\
s_{n+1} &= \frac{25}{s_n(z + x/z + 1)^2} \quad \to 1 && \text{(30.4-11e)}\\
a_{n+1} &= s_n^2 a_n - 5^n \left(\frac{s_n^2 - 5}{2} + \sqrt{s_n\,(s_n^2 - 2s_n + 5)}\right) \quad \to \frac{1}{\pi} && \text{(30.4-11f)}\\
a_n - \frac{1}{\pi} &< 16 \cdot 5^n\, e^{-\pi\,5^n} && \text{(30.4-11g)}
\end{aligned}$$

Cubic (third order) iteration from [25], implemented in [hfloat: src/pi/pi3rd.cc], #FPM=200.3:

$$\begin{aligned}
a_0 &= \frac{1}{3}, \qquad s_0 = \frac{\sqrt{3}-1}{2} && \text{(30.4-12a)}\\
r_{k+1} &= \frac{3}{1 + 2\,(1 - s_k^3)^{1/3}} && \text{(30.4-12b)}\\
s_{k+1} &= \frac{r_{k+1} - 1}{2} && \text{(30.4-12c)}\\
a_{k+1} &= r_{k+1}^2\,a_k - 3^k\,(r_{k+1}^2 - 1) \quad \to \frac{1}{\pi} && \text{(30.4-12d)}
\end{aligned}$$

Nonic (9th order) iteration from [25], implemented in [hfloat: src/pi/pi9th.cc], #FPM=273.7:

$$a_0 = \frac{1}{3}, \qquad r_0 = \frac{\sqrt{3}-1}{2}, \qquad s_0 = (1-r_0^3)^{1/3} \tag{30.4-13a}$$

$$t = 1 + 2\,r_k \tag{30.4-13b}$$

$$u = \left(9\,r_k\,(1 + r_k + r_k^2)\right)^{1/3} \tag{30.4-13c}$$

$$v = t^2 + t\,u + u^2 \tag{30.4-13d}$$

$$m = \frac{27\,(1 + s_k + s_k^2)}{v} \tag{30.4-13e}$$

$$a_{k+1} = m\,a_k + 3^{2\,k-1}\,(1-m) \quad \to \frac{1}{\pi} \tag{30.4-13f}$$

$$s_{k+1} = \frac{(1-r_k)^3}{(t+2\,u)\,v} \tag{30.4-13g}$$

$$r_{k+1} = (1-s_k^3)^{1/3} \tag{30.4-13h}$$

30.4.2 Measured timings and operation counts

#FPM	order	routine in hfloat	time
78.424	2	pi_agm_sch()	76 sec
98.424	2	pi_agm()	93 sec
99.510	2	pi_agm3(fast variant)	94 sec
108.241	2	pi_agm3(slow variant)	103 sec
149.324	4	pi_agm(quartic)	139 sec
155.265	4	pi_agm3(quartic, fast variant)	145 sec
164.359	4	pi_4th_order(r=16 variant)	154 sec
169.544	4	pi_agm3(quartic, slow variant)	159 sec
170.519	4	pi_4th_order(r=4 variant)	160 sec
182.710	3	pi_cubic_agm()	173 sec
200.261	3	pi_3rd_order()	189 sec
255.699	2	pi_2nd_order()	240 sec
273.763	9	pi_9th_order()	256 sec
276.221	2	pi_derived_agm()	259 sec
353.202	5	pi_5th_order()	329 sec

Figure 30.4-A: Measured operations counts and timings for various iterations for the computation of π to 4 million decimal digits.

The operation counts and timings for the algorithms given so far when computing π to 4 million decimal digits (using 1 million `LIMB`s and radix 10,000) are shown in figure 30.4-A. In view of these figures it seems surprising that the quartic algorithms `pi_4th_order()` and the quartic AGM `pi_agm(quartic)` are usually considered close competitors to the second order AGM schemes.

Apart from the operation count the number of variables used has to be taken into account. The algorithms using more variables (like `pi_5th_order()`) cannot be used to compute as many digits as those using only a few (notably the AGM-schemes) given a fixed amount of RAM. Higher order algorithms tend to require more variables.

A further disadvantage of the algorithms of higher order is the more discontinuous growth of the work: if just a few more digits are to be computed than are available after step k then an additional step is required. Consider an extreme case where an algorithm T of order $1,000$ would compute 1 million digits after the second step, at a slightly lower cost than the most effective competitor. Then algorithm T would likely be the 'best' one only for small ranges in the number of digits around the values 10^3, 10^6, 10^9, ...

Finally, it is much easier to find special arithmetical optimizations for the 'simple' (low order) algorithms, Schönhage's AGM variant being the prime example.

30.4.3 More iterations for π

The following iterations are not implemented in hfloat.

Third order algorithm from [49]:

$$v_0 = 2^{-1/8}, \qquad v_1 = 2^{-7/8}\left((1-3^{1/2})\,2^{-1/2}+3^{1/4}\right) \tag{30.4-14a}$$

$$w_0 = 1, \qquad \alpha_0 = 1, \qquad \beta_0 = 0 \tag{30.4-14b}$$

$$v_{n+1} = v_n^3 - \left\{v_n^6 + \left[4v_n^2(1-v_n^8)\right]^{1/3}\right\}^{1/2} + v_{n-1} \tag{30.4-14c}$$

$$w_{n+1} = \frac{2v_n^3 + v_{n+1}\left(3v_{n+1}^2v_n^2-1\right)}{2v_{n+1}^3 - v_n\left(3v_{n+1}^2v_n^2-1\right)}\,w_n \tag{30.4-14d}$$

$$\alpha_{n+1} = \left(\frac{2v_{n+1}^3}{v_n}+1\right)\alpha_n \tag{30.4-14e}$$

$$\beta_{n+1} = \left(\frac{2v_{n+1}^3}{v_n}+1\right)\beta_n + (6w_{n+1}v_n - 2v_{n+1}w_n)\,\frac{v_{n+1}^2\alpha_n}{v_n^2} \tag{30.4-14f}$$

$$\pi_n = \frac{8\cdot 2^{1/8}}{\alpha_n\,\beta_n} \quad \to \pi \tag{30.4-14g}$$

Second order algorithm from [57]:

$$\alpha_0 = 1/3, \qquad m_0 = 2 \tag{30.4-15a}$$

$$m_{n+1} = \frac{4}{1+\sqrt{(4-m_n)\,(2+m_n)}} \tag{30.4-15b}$$

$$\alpha_{n+1} = m_n\,\alpha_n + \frac{2^n}{3}(1-m_n) \quad \to \frac{1}{\pi} \tag{30.4-15c}$$

Implicit second order algorithm from [57] (also in [54, p.700]):

$$\alpha_0 = 1/3, \qquad s_1 = 1/3 \tag{30.4-16a}$$

$$(s_n)^2 + (s_n^*)^2 = 1 \tag{30.4-16b}$$

$$(1+3\,s_{n+1})\,(1+3\,s_n^*) = 4 \tag{30.4-16c}$$

$$\alpha_{n+1} = (1+3\,s_{n+1})\,\alpha_n - 2^n\,s_{n+1} \quad \to \frac{1}{\pi} \tag{30.4-16d}$$

It is trivial to turn this algorithm into an explicit form as with the next algorithm. However, there exist iterations that cannot be turned into explicit forms.

Implicit fourth order algorithm from [57] (also in [54, p.700]):

$$\alpha_0 = 1/3 \qquad s_1 = \sqrt{2}-1 \tag{30.4-17a}$$

$$(s_n)^4 + (s_n^*)^4 = 1 \tag{30.4-17b}$$

$$(1+3\,s_{n+1})\,(1+3\,s_n^*) = 2 \tag{30.4-17c}$$

$$\alpha_{n+1} = (1+s_{n+1})^4\alpha_n + \frac{4^{n+1}}{3}\left[1-(1+s_{n+1})^4\right] \quad \to \frac{1}{\pi} \tag{30.4-17d}$$

Combining two steps of the fourth order iteration leads to an algorithm of order 16. The following form

is given in [57, p.111]:

$$\alpha_0 = 1/3, \qquad s_1 = \sqrt{2}-1 \tag{30.4-18a}$$
$$s_n^* = \left(1-s_n^4\right)^{1/4} \tag{30.4-18b}$$
$$x_n = 1/\left(1+s_n^*\right)^4 \tag{30.4-18c}$$
$$y_n = x_n\left(1+s_n\right)^4 \tag{30.4-18d}$$
$$\alpha_n = 16\,y_n\,\alpha_{n-1} + \frac{4^{2n-1}}{3}\left[1-12\,x_n-4\,y_n\right] \quad \to \frac{1}{\pi} \tag{30.4-18e}$$
$$t_n = 1+s_n^* \tag{30.4-18f}$$
$$u_n = \left[8\,s_n^*\left(1+{s_n^*}^2\right)\right]^{1/4} \tag{30.4-18g}$$
$$s_{n+1} = \frac{\left(1-s_n^*\right)^4}{(t+u)^2\left(t^2+u^2\right)} \tag{30.4-18h}$$

Quadratic iteration by Christian Hoffmann, given in [135, p.5]:

$$a_0 = \sqrt{2}, \qquad b_0 = 0, \qquad p_0 = 2+\sqrt{2} \tag{30.4-19a}$$
$$a_{n+1} = \frac{1}{2}\left(\sqrt{a_n}+1/\sqrt{a_n}\right) \quad \to 1+ \tag{30.4-19b}$$
$$b_{n+1} = \sqrt{a_n}\,\frac{b_n+1}{b_n+a_n} \quad \to 1- \tag{30.4-19c}$$
$$p_{n+1} = p_n\,b_{n+1}\,\frac{1+a_{n+1}}{1+b_{n+1}} \quad \to \pi \tag{30.4-19d}$$

Note that relation 30.4-19b deviates from the one given in the cited paper which seems to be incorrect. This is a variant of the iteration given as relations 30.4-9a..30.4-9e on page 584. The values p_k are identical in both iterations.

Cubic iteration given in [51, p.125]:

$$s_0 = \sqrt{3+2\sqrt{3}}, \qquad a_0 = 1/2 \tag{30.4-20a}$$
$$m_n = 3/s_n \tag{30.4-20b}$$
$$a_{n+1} = \left[\left(s_n^2-1\right)^{1/3}+2\right]^2/s_n \tag{30.4-20c}$$
$$a_{n+1} = m_n^2\,a_n - 3^n\left(m_n^2+2m_n-3\right)/2 \quad \to \frac{1}{\pi} \tag{30.4-20d}$$

The cited paper actually gives a more general form, here we take $N=1$ for simplicity.

Cubic iteration given in [75, p.1506, it-1.2]:

$$t_0 = 1/3, \qquad s_0 = \left(\sqrt{3}-1\right)/2 \tag{30.4-21a}$$
$$s_n = \frac{1-\left(1-s_{n-1}^3\right)^{1/3}}{1+2\left(1-s_{n-1}^3\right)^{1/3}} \tag{30.4-21b}$$
$$= \frac{\left(1-s_{n-1}^3\right)^{-1/3}-1}{\left(1-s_{n-1}^3\right)^{-1/3}+2} \tag{30.4-21c}$$
$$t_n = \left(1+2s_n\right)^2\,t_{n-1} - 3^{n-1}\left(\left(1+2s_n\right)^2-1\right) \quad \to \frac{1}{\pi} \tag{30.4-21d}$$

Note the corrected denominator in relation 30.4-21b (exponent of s_{n-1} is wrongly given as 2).

Quadratic iteration given in [75, p.1507, it-1.3]:

$$k_0 = 0, \qquad s_0 = 1/\sqrt{2} \tag{30.4-22a}$$

$$s_n = \frac{1-\sqrt{1-s_{n-1}^2}}{1+\sqrt{1-s_{n-1}^2}} \tag{30.4-22b}$$

$$k_n = (1+s_n)^2\, k_{n-1} + 2^n\,(1-s_n)\,s_n \quad \to \frac{1}{\pi} \tag{30.4-22c}$$

Cubic iteration given in [75, p.1507, it-1.4]:

$$k_0 = 0, \qquad s_0 = 1/\sqrt[3]{2} \tag{30.4-23a}$$

$$s_n = \frac{1-\sqrt[3]{1-s_{n-1}^3}}{1+2\,\sqrt[3]{1-s_{n-1}^3}} \tag{30.4-23b}$$

$$k_n = (1+2\,s_n)^2\, k_{n-1} + 8\cdot 3^{n-2}\,\sqrt{3}\,s_n\,\frac{1-s_n^3}{1+2\,s_n} \quad \to \frac{1}{\pi} \tag{30.4-23c}$$

Quadratic iteration given in [75, p.1508, it-1.5]:

$$k_0 = 0, \qquad y_0 = 8/9 \tag{30.4-24a}$$

$$y_n = 2\,\frac{6\,y_{n-1}^2 - 5\,y_{n-1} + \sqrt{y_{n-1}(4-y_{n-1})}}{9\,y_{n-1}^2 - 6\,y_{n-1} + 1} \tag{30.4-24b}$$

$$k_n = 2^n\,\sqrt{3}\,\frac{y_{n-1}\,(1-y_{n-1})}{\sqrt{4-3\,y_{n-1}}} + (4-3\,y_{n-1})\,k_{n-1} \quad \to \frac{1}{\pi} \tag{30.4-24c}$$

Quadratic iteration given in [75, p.1508, it-1.6]:

$$k_0 = 0, \qquad y_0 = 4/5 \tag{30.4-25a}$$

$$y_n = \frac{2\,y_{n-1}^2 - y_{n-1} + \sqrt{4\,y_{n-1} - 3\,y_{n-1}^2}}{1+y_{n-1}^2} \tag{30.4-25b}$$

$$Q_n = \sqrt{(y_{n-1}+1)\,(4-3\,y_{n-1})\,(y_{n-1}^2 - 3\,y_{n-1} + 4)} \tag{30.4-25c}$$

$$k_n = \frac{2^n}{\sqrt{7}}\,\frac{y_{n-1}\,(1-y_{n-1})}{2-y_{n-1}}\,Q_n + (2-y_{n-1})^2\,k_{n-1} \quad \to \frac{1}{\pi} \tag{30.4-25d}$$

Quadratic iteration (as product, two forms) given in [53, p.324]:

$$x_1 = \frac{2}{9}\left(\sqrt{6}+2\right), \qquad y_1 = \frac{1}{6}\left(\sqrt{6}+4\right) \tag{30.4-26a}$$

$$x_{n+1} = \frac{2\left(\sqrt{x_n}+x_n\right)}{1+3\,x_n} \tag{30.4-26b}$$

$$y_{n+1} = \frac{2y_n + y_n/\sqrt{x_n} + \sqrt{x_n}}{1+3\,y_n} \tag{30.4-26c}$$

$$\pi = \frac{27}{8}\,\prod_{n=1}^{\infty}\frac{(1+3\,x_n)^2}{(1+3\,y_n)/4} \tag{30.4-26d}$$

$$\pi = \frac{5+2\sqrt{6}}{3}\,\prod_{n=1}^{\infty}\frac{\left(1+1/\sqrt{x_n}\right)^2}{1+3\,y_n} \tag{30.4-26e}$$

The definitive source for iterations to compute π and the underlying mathematics is [52].

30.5 Arctangent relations for π *

This section is about relations of the form

$$k\,\frac{\pi}{4} \quad = \quad m_1 \arctan\frac{1}{x_1} + m_2 \arctan\frac{1}{x_2} + \ldots + m_n \arctan\frac{1}{x_n} \tag{30.5-1}$$

where $k,\ m_1,\ldots,m_n,\ x_1,\ldots,x_n \in \mathbb{Z}$ (in fact, $k = 1$ almost always). This is an n-term relation. For example, a 4-term relation, found 1896 by Størmer [222], is

$$\frac{\pi}{4} \quad = \quad +44 \arctan\frac{1}{57} + 7 \arctan\frac{1}{239} - 12 \arctan\frac{1}{682} + 24 \arctan\frac{1}{12943} \tag{30.5-2}$$

We use the following compact notation

```
m1[x1] +m2[x2] + ... +mn[xn]  == k * Pi/4
```

for relation 30.5-1. For example, Størmer's relation 30.5-2 would be written as

```
+44[57] +7[239] -12[682] +24[12943]   == 1 * Pi/4
```

We write the relations so that the arguments x_j are strictly increasing. Further, n-term relations are sorted so that the first arguments x_1 are in decreasing order (if $x_1 \ldots x_j$ coincide with two relations then the arguments x_{j+1} are used for sorting). For example, a few 6-term relations are

```
+322[577]  +76[682] +139[1393] +156[12943] +132[32807] +44[1049433]   == 1 * Pi/4
+122[319]  +61[378]  +115[557]   +29[1068]   +22[3458]   +44[27493]   == 1 * Pi/4
+100[319] +127[378]   +71[557]   -15[1068]   +66[2943]  +44[478707]   == 1 * Pi/4
+337[307] -193[463] +151[4193]  +305[4246] -122[39307]  -83[390112]   == 1 * Pi/4
+183[268]  +32[682]  +95[1568]   +44[4662] -166[12943]   -51[32807]   == 1 * Pi/4
```

Note that the second and third relation are sorted according to their fifth arguments (3458 and 2943). Among all n-term relations we consider a relation *better* than another if it precedes it. The first one is the *best* relation. Our goal is to find the best n-term relation for n small. For example, the relation

```
+322[577]  +76[682] +139[1393] +156[12943] +132[32807] +44[1049433]   == 1 * Pi/4
```

is the best (known!) 6-term relation. The best n-term relations for $2 \le n \le 12$ currently known are shown in figure 30.5-A. Note that $k = -1$ in the 10-term relation, and $k = 2$ in the 12-term relation. The best relations for $13 \le n \le 21$ (shortened to save space) are shown in figure 30.5-B. Figure 30.5-C gives just the first argument (x_1) of the best relations for $2 \le n \le 27$.

```
+4[5] -1[239]   == 1 * Pi/4

+12[18] +8[57] -5[239]   == 1 * Pi/4

+44[57] +7[239] -12[682] +24[12943]   == 1 * Pi/4

+88[192] +39[239] +100[515] -32[1068] -56[173932]   == 1 * Pi/4

+322[577] +76[682] +139[1393] +156[12943] +132[32807] +44[1049433]   == 1 * Pi/4

+1587[2852] +295[4193] +593[4246] +359[39307]
 +481[55603] +625[211050] -708[390112]   == 1 * Pi/4

+2192[5357] +2097[5507] -227[9466] +832[12943]
 +537[34522] -2287[39307] -171[106007] -708[1115618]   == 1 * Pi/4

+3286[34208] +9852[39307] +5280[41688] +7794[44179]
 +7608[60443] +4357[275807] -1484[390112] -1882[619858] +776[976283]   == 1 * Pi/4

+1106[54193] -30569[78629] -28687[88733] -13882[173932]
 +9127[390112] -9852[478707] -24840[1131527] +4357[3014557]
 +21852[5982670] +23407[201229582]   == -1 * Pi/4

+36462[390112] +135908[485298] +274509[683982] -39581[1984933]
 +178477[2478328] -114569[3449051] -146571[18975991] +61914[22709274]
 -69044[24208144] -89431[201229582] -43938[2189376182]   == 1 * Pi/4

+893758[1049433] +655711[1264557] +310971[1706203] +503625[1984933]
 -192064[2478328] -229138[3449051] -875929[18975991] -616556[21638297]
 -187143[22709274] -171857[24208144] -251786[201229582] -432616[2189376182]   == 2 * Pi/4
```

Figure 30.5-A: Best n-term arctan relations currently known for $2 \le n \le 12$.

```
+1126917[3449051] +1337518[4417548] ... -216308[2189376182]   == 1 * Pi/4
+446879[6826318] +5624457[8082212] ... +483341[17249711432]   == 1 * Pi/4
+5034126[20942043] +1546003[22709274] ... +1337518[250645741818]   == 1 * Pi/4
+14215326[53141564] +6973645[54610269] ... +8735690[34840696582]   == 1 * Pi/4
+12872838[201229582] +27205340[203420807] ... +35839320[134520516108]   == 1 * Pi/4
+2859494[299252491] -41068896[321390012] ... -89623108[18004873694818]   == -1 * Pi/4
+270619381[778401733] -138919506[1012047353] ... +146407224[30038155625330]   == 1 * Pi/4
+807092487[2674664693] +479094776[2701984943] ... +214188292[564340076432]   == 1 * Pi/4
+598245178[5513160193] -115804626[7622130953] ... -1521437626[38057255532937]   == 1 * Pi/4
```

Figure 30.5-B: The best n-term arctan relations (shortened) currently known for $13 \leq n \leq 21$.

```
 n-terms       min-arg
    2                5  Machin (1706)    +4[5] -1[239]   == 1 * Pi/4
    3               18  Gauss (YY?)      +12[18] +8[57] -5[239]   == 1 * Pi/4
    4               57  Stormer (1896)
    5              192  JJ (1993), prev: Stormer (1896) 172
    6              577  JJ (1993)
    7            2,852  JJ (1993)
    8            5,357  JJ (2006), prev: JJ (1993) 4,246
    9           34,208  JJ (2006), prev: JJ (1993) 12,943, prev: Gauss (Y?) 5,257
   10           54,193  JJ (2006), prev: JJ (1993) 51,387
   11          390,112  JJ (1993)
   12        1,049,433  JJ (2006), prev: JJ (1993) 683,982
   13        3,449,051  JJ (2006), prev: JJ (1993) 1,984,933
   14        6,826,318  JJ (2006)
   15       20,942,043  HCL (1997), prev: MRW (1997) 18.975,991
   16       53,141,564  JJ (2006)
   17      201,229,582  JJ (2006)
   18      299,252,491  JJ (2006)
   19      778,401,733  JJ (2006)
   20    2,674,664,693  JJ (2006)
   21    5,513,160,193  JJ (2006)
   22   17,249,711,432  JJ (2006), prev: 16,077,395,443 MRW (27-Jan-2003)
   23   58,482,499,557  JJ (2006)
   24  102,416,588,812  JJ (2006)
   25  160,422,360,532  JJ (2006)
   26  392,943,720,343  JJ (2006)
   27  970,522,492,753  JJ (2006)
MRW := Michael Roby Wetherfield
HCL := Hwang Chien-Lih
JJ := Joerg Arndt
```

Figure 30.5-C: First arguments of the best n-term arctan relation known today, for $2 \leq n \leq 27$. The 4-term entry corresponds to relation 30.5-2 on the facing page.

30.5.1 How to find one relation

In the 5-term relation

```
+88[192] +39[239] +100[515] -32[1068] -56[173932]   == 1 * Pi/4
```

factor $x_j^2 + 1$ for all (inverse) arguments x_j:

```
192^2+1 == 36865 == 5 73 101
239^2+1 == 57122 == 2 13 13 13 13
515^2+1 == 265226 == 2 13 101 101
1068^2+1 == 1140625 == 5 5 5 5 5 5 73
173932^2+1 == 30252340625 == 5 5 5 5 5 13 73 101 101
```

Note that all odd prime factors are the four primes $5, 13, 73, 101$. The coefficients m_j can be computed as follows. Write (for all arguments x_j)

$$x_j^2 + 1 \quad = \quad 2^{e(j,0)}\, 5^{e(j,1)}\, 13^{e(j,2)}\, 73^{e(j,3)}\, 101^{e(j,4)} \tag{30.5-3}$$

Now define a matrix M using the exponents $e(j, u)$ (ignoring the prime 2):

$$M^T_{j,i} \quad := \quad \pm e(j, i) \tag{30.5-4}$$

The sign of $M_{j,i}$ is minus if $(x_j \bmod p_i) < p_i/2$. With our example we obtain

```
transpose(M) :=
  [-5, -1, +1, -2]  //  173932^2+1 ==   5^5 *13^1 *73^1 *101^2
  [+6,  0, +1,  0]  //  1068^2+1   ==   5^6       *73^1
  [ 0, +1,  0, -2]  //  515^2+1    ==        13^1       *101^2  (*2)
  [ 0, -4,  0,  0]  //  239^2+1    ==        13^4               (*2)
  [-1,  0, +1, +1]  //  192^2+1    ==   5^1       *73^1 *101^1
//  5, 13, 73, 101  <--= primes
```

For the signs of the upper left 3×2 sub-matrix, note that $(173932 \bmod 5) = 2 < 5/2$, $(173932 \bmod 13) = 5 < 13/2$, $(1068 \bmod 5) = 3 > 5/2$, and $(515 \bmod 13) = 8 > 13/2$. The nullspace of M consists of one vector:

```
[-56,   -32,   100,   39,   88]
```

This tells us that

```
+88[192] +39[239] +100[515] -32[1068] -56[173932]   == k * Pi/4
```

We determine that $k = 1$ by a floating point computation of the left hand side. Quite often one finds a relation where $k = 0$, which we are not interested in. For example, the candidates 12943, 1068, 682, 538, 239 all factor into (2 and) the odd primes 5, 13, 61, 73. The matrix M is

```
transpose(M) =
  [+4, +3, -1,  0] //  12943^2+1  ==   5^4 *13^3 *61^1         (*2)
  [+6,  0,  0, +1] //  1068^2+1   ==   5^6             *73^1
  [-3,  0, -2,  0] //  682^2+1    ==   5^3       *61^2
  [+1, -1, +1, -1] //  538^2+1    ==   5^1 *13^1 *61^1 *73^1
  [ 0, -4,  0,  0] //  239^2+1    ==        13^4               (*2)
//  5, 13, 61, 73  <--= primes
```

The nullspace of M is

```
[1, -1, -1, -1, 1]
```

and the relation is

```
+1[239] -1[538] -1[682] -1[1068] +1[12943]   ==   0
```

30.5.2 Searching for sets of candidate arguments

A set of candidate arguments x_j will give a relation only if $x_j^2 + 1$ factor into a common set of primes. Apart from the factor 2, all prime factors are of the form $4\,i+1$. One can choose a subset of those primes $S := \{p_1, \ldots, p_u\}$ and test which of the products $P = 2^{e_0} \cdot p_1^{e_1} \cdots p_u^{e_u}$ are of the form $P = x^2 + 1$. The test is to determine whether $P - 1$ is a perfect square. The pari/gp function `issquare()` does this in an efficient way (as described in [83]). A recursive implementation of the search is

```
\\ global variables:
ct=0;  \\ count solutions
av=vector(1000);  \\ vector containing solutions
\\ pv = [...]; \\ vector of primes of the form 4*i+1
m=10^20;  \\ search max := sqrt(m)

check(t)=
{
    local(a);
    if ( issquare(t-1, &a),  ct++; av[ct] = a; );
    if ( issquare(t+t-1, &a),  ct++; av[ct] = a; );
}

gen_rec(d, p)=
{
    local(g, gg, t);
    if ( d>length(pv),  return() );
    g = pv[d];
    gg = 1;
    while ( 1,
        t = p * gg;
        if ( t>m,  return() );
        if ( gg!=1, check(t) );
        gen_rec(d+1, t);
        gg *= g;
    );
    return();
}
```

We do the search using the four primes 5, 13, 61, and 73:

```
pv=[5, 13, 61, 73]; \\ vector of primes
gen_rec(1, 1);  \\ do the search
```

The candidates found are

```
12943, 1068, 682, 538, 239, 57, 27, 18, 11, 8, 7, 5, 3, 2
```

The following relations are found:

```
+1[239] -1[538] -1[682] -1[1068] +1[12943]   ==   NULL
+44[57] +7[538] -5[682] +7[1068] +17[12943]   == 1 * Pi/4  (5-term)
+1[27] +42[57] +6[538] -5[682] +7[1068] +16[12943]   == 1 * Pi/4  (6-term)
+1[18] +41[57] +6[538] -5[682] +6[1068] +16[12943]   == 1 * Pi/4  (6-term)
+1[11] +39[57] +6[538] -5[682] +6[1068] +15[12943]   == 1 * Pi/4  (6-term)
  [--snip--]
+1[3] +26[57] +4[538] -3[682] +4[1068] +10[12943]   == 1 * Pi/4  (6-term)
+1[2] +18[57] +3[538] -2[682] +3[1068] +7[12943]   == 1 * Pi/4  (6-term)
```

The search is reasonably fast for up to about 12 primes. The drawback of this method is that one has to guess which prime set may lead to a good arctan relation. The prime set $\{5, 13, 17, 29, 37, 53, 61, 89, 97, 101\}$ lead (April-1993) to

```
+36462[390112] +135908[485298] +274509[683982] -39581[1984933]
 +178477[2478328] -114569[3449051] -146571[18975991] +61914[22709274]
 -69044[24208144] -89431[201229582] -43938[2189376182]   == 1 * Pi/4
```

which is still the best 11-term relation known today.

The April-2006 computations were done with a more exhaustive search described in the next section.

30.5.3 Exhaustive search for sets of candidate arguments

We want to find all x where $x^2 + 1$ factors into (2 and) the first 64 primes of the form $4\,i + 1$ ($S = \{5, 13, 17, 29, \ldots, 761\}$). Call the resulting set of candidates A. We will later try (for small n) all $(n-1)$-subsets of S and test whether the corresponding subset of A leads to an arctan relation.

The most simple approach is to factor (for x up to a practical maximum) all $x^2 + 1$ and add x to the set A if all odd prime factors of $x^2 + 1$ are in S. The method, however, is rather slow: about 11,000 CPU cycles are needed for each test.

A much faster approach is the following sieving method. We can determine exactly when a prime p divides x^2+1 by solving $x^2 \equiv -1 \pmod{p}$ as shown in section 37.9 on page 751. We can further solve $x^2 \equiv -1 \pmod{p^h}$ for all h as shown in the cited section. Initialize an array with the value 1 for even indices, else with 2 (x^2+1 is even exactly if x is odd). For each prime $p \in S$ do, for all powers p^h, as follows: multiply the array entries with indices $s, s+p^h, s+2\,p^h, s+3\,p^h, \ldots$ where $s^2 \equiv -1 \pmod{p^h}$ by p. Finally find the entries with index x that are equal to x^2+1, these are the candidates.

There are three useful improvements. Firstly, we can use the logarithm of a prime and add it instead of multiplying by the prime. The final test is then whether entry x is (approximately) equal to $\log(x^2+1)$. Secondly, the array can be avoided altogether by using priority queues (see section 4.5 on page 148). An event of the priority queue will be to add $\log(p)$ (initially at index $x = s$ where $s^2 \equiv -1 \pmod{p^h}$). The event must then be rescheduled to $x + p^h$. Thirdly, almost all computations of the logarithm can be avoided by observing that both x^2+1 and the logarithm are strictly increasing functions. We call a number x so that x^2+1 has all odd prime factors in S is a candidate. The sum of logarithms (of primes) for candidates x are equal to $\log(x^2+1)$. If a was the last candidate then for the next candidate b the sum of logarithms must be strictly greater than that for a. Thereby we only need to compute $\log(x^2+1)$ if a new sum of logarithms is greater than the one for the candidate found most recently. It turns out that a logarithm is computed *exactly* whenever a new candidate is found.

The search costs about 250 cycles per test, which is a good improvement over the first attempt. Analysis of the machine code shows that most of the time is spent in the reschedule operations.

The final improvement comes from the separation of the frequent event (small prime powers) from the rare events (big prime powers). One has to use an array again (but only a small one that fits into level-1 cache and a segmented search should be used). Now the optimum value has to be found from which on an event is considered rare. Very surprisingly, it turns out that the search is fastest if *all* events are considered frequent! This means we can forget about the priority queues. A better suited algorithm (and implementation) for a priority queue might give different results.

The resulting routine is remarkably fast, it uses just slightly more than 11 cycles per test. It was used to determine all candidates $x \le 10^{14}$. The search took about 8 days. The last entries in the list of candidates are

```
99205431802196^2+1 = [13.29.37.53.89.157.241.257.337.373.401^2.761]
99238108604548^2+1 = [5.29.37.61^2.101.349.397^2.433.557^2.661]
99311314035643^2+1 = [2.5^2.13.29.73.113.233.241.269.281.293.317.349.461]
99395767528881^2+1 = [2.13.29.37.53.149.173.181.193.313.353.373.401.449]
99501239756693^2+1 = [2.5^4.13.29.37^2.61.233.277.313^3.317.401]
99627378461772^2+1 = [5.13^2.37.41.73^2.137.277.281.521.557.617.761]
99759820688082^2+1 = [5^2.17.29.37.109.181.257.269.337.389.409.457.653]
99849755159917^2+1 = [2.5.89.101.181.233.257.293.389.457.521.557.677]
99950583525307^2+1 = [2.5^3.13.173.181.193.241^3.257.457^2.677]
99955223464153^2+1 = [2.5.13.61^2.101^2.109.373.421.433.509.709.757]
```

The search produced 43,936 candidates (including 0 and 1). Exactly that many logarithms were computed. This means that on average one logarithm was computed for one in $10^{14}/43,936 > 2 \cdot 10^9$ values tested.

One can further extend the list by testing (for each element x) whether the right hand side formula

```
[x] == [x+d] + [x+(x^2+1)/d]  where d divides x^2+1
```

leads to new candidate $x+d$ and $x+(x^2+1)/d$. Additionally one can try the arguments on the right hand side of relations like

```
[x] == 2[2*x] - [4*x^3+3*x]
[x] == [2*x-1] + [2*x+1] - [2*x^3+x]
[x] == 3[3*x] - [(9*x^3+7*x)/2] - [(27*x^3+9*x)/2]
```

Michael Roby Wetherfield has developed a more sophisticated approach for extending the list and sent me a big set of candidates beyond 10^{14}. His methods are described in [243] (see also [244], [227], [33], and [165]). We note that a single value, $x = 276,914,859,479,857,813,947$ where

```
x^2+1 = [2.5.13.17.29^3.41.53^2.73^2.101.157.181.229.241.313.397.401.509.577]
```

was discarded because it is bigger than $2^{64} = 18,446,744,073,709,551,616$.

We note the curious relation

$$[k\,a] \;=\; [(k+1)\,a] + [(k+1)\,k\,a] - [(k^4+2\,k^3+k^2)\,a^3 + (k^2+k+1)\,a] \tag{30.5-5}$$

Set $f(a,k) := (k^4+2\,k^3+k^2)\,a^3 + (k^2+k+1)\,a$, then

$$f(a,k)^2+1 \;=\; \Big((k\,a)^2+1\Big)\cdot\Big(((k+1)\,a)^2+1\Big)\cdot\Big(((k+1)\,k\,a)^2+1\Big) \tag{30.5-6}$$

30.5.4 Searching for all n-term relations

To find all n-term relations whose arguments are a subset of our just determined list of candidates, we have to test all subsets of $(n-1)$ (out of 64) odd primes, select the corresponding values x, and compute the nullspace as described. Let A_j be the j-th candidate. An array M of 64-bit auxiliary values is used. Its j-th entry M_j is a bit-mask corresponding to the odd primes in the factorization of A_j^2+1: bit i of M_j is set if the i-th odd prime divides A_j^2+1.

To find n-term relations, we must try all $\binom{64}{n-1}$ subsets of size $n-1$ out of the 64 odd primes in our scope. The bit-combination routine from section 1.25 on page 61 was used for this task. The selection of the entries that factor completely in the subset of $n-1$ primes under consideration can be done with a single bit-AND and a branch. The candidates with more than $n-1$ odd primes in their factorization should be discarded before the search.

While the search is very fast for small n, it does not finish in reasonable time for $n > 8$. A considerable speedup can be achieved by splitting our $N = 64$ odd primes into a group of the 20 smallest and $b = 64-20 = 44$ 'big' primes. Write ($q = n-1$ and)

$$\binom{N}{q} \;=\; \binom{b}{0}\binom{N-b}{q} + \binom{b}{1}\binom{N-b}{q-1} + \binom{b}{2}\binom{N-b}{q-2} + \ldots \tag{30.5-7a}$$

$$\;=\; \sum_{j=0}^{q}\binom{b}{j}\binom{N-b}{q-j} \tag{30.5-7b}$$

This means, we first select the $j = 0,\ 1,\ 2,\ \ldots$-subsets of the big primes. We copy the corresponding candidates whose big prime factors are in the current subset into a new array B. The size of B will be significantly smaller than the size of A. From this array we select the arguments according to subsets of the small primes (leaving the subset of big primes fixed). This results in a much improved memory locality and accelerates the search by a factor of about 25.

Still, the limit for n so that an exhaustive search can be done has only been moved a little. But if we look at the prime sets that lead to the best relations, we observe that small primes are much 'preferred':

```
n     prime set of best relation
2     {13}
3     {5, 13}
4     {5, 13, 61}
5     {5, 13, 73, 101}
6     {5, 13, 61, 89, 197}
7     {5, 13, 17, 29, 97, 433}
8     {5, 13, 29, 37, 61, 97, 337}
9     {5, 13, 17, 29, 41, 53, 97, 269}
10    {5, 13, 17, 41, 53, 73, 97, 101, 157}
11    {5, 13, 17, 29, 37, 53, 61, 89, 97, 101}
12    {5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 197}
13    {5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 181, 281}
14    {5, 13, 17, 29, 37, 53, 61, 89, 97, 101, 181, 269, 457}
15    {5, 13, 17, 29, 37, 41, 53, 61, 89, 97, 101, 181, 337, 389}
```

The data suggests that the best possible relation is found long before the search space is exhausted. Therefore we will stop after the number of big primes in the subset is greater than, say, 4. In practice both parameters, the number b of primes considered big and the maximum number of primes taken from that set should be chosen depending on n.

Another important improvement is to discard small candidates before the search. One thereby avoids the huge amount of uninteresting relations with small first arguments x_1. Obviously, the amount of nullspace computation is also reduced significantly.

The results of the searches can be found in [18]. While the searches for the n-term relations with $n > 11$ did not even exhaust the table of candidates (which in turn is incomplete!) one can be reasonably sure that the best relations within our scope (of the first 64 odd primes $4\,i + 1$) have been found. Indeed I do not expect to see a better relation for any $n \leq 15$.

To improve on the results, one should use the first 128 odd primes $4\,i + 1$, sieve up to 10^{16} (distributed on 100 machines), and use a 3-phase subset selection instead of the described 2-phase selection. The selection (an nullspace computation) stage should also be done in a distributed fashion to reasonably exhaust the table of candidates. Such a computation would likely improve on some of the relations with more than 17 terms and produce up to 35-term relations that are in the vicinity of the best possible.

A method for the simultaneous computation of logarithms of small primes that uses a similar method to the one given here is described in section 31.4 on page 608.

Chapter 31

Logarithm and exponential function

We describe algorithms for the computation of the exponential (and hyperbolic cosine) function, and the logarithm (and inverse tangent). Constructions of superlinear iterations the compute the functions from their inverses are given.

We give argument reduction schemes and methods for the fast computation of the exponential and logarithm of power series.

31.1 Logarithm

31.1.1 AGM-based computation

The (natural) logarithm can be computed using the relation (see [52, p.221])

$$\left|\log(d) - R'(10^{-n}) + R'(10^{-n}\,d)\right| \;\leq\; \frac{n}{10^{2(n-1)}} \tag{31.1-1a}$$

$$\log(d) \;\approx\; R'(10^{-n}) - R'(10^{-n}\,d) \tag{31.1-1b}$$

which holds for $n \geq 3$ and $d \in]\frac{1}{2}, 1[$. Note that the first term on the right hand side is constant and can be saved for subsequent log-computations. One uses the relation

$$\log(M\,R^X) \;=\; \log(M) + X\,\log(R) \tag{31.1-2}$$

where M is the mantissa, R the radix, and X the exponent of the floating point representation. The value $\log(R)$ is computed only once. If M is not in the interval [1/2, 3/2] an argument reduction is made via

$$\log(M) \;=\; \log\left(M\,s^f\right) - f\,\log(s) \tag{31.1-3}$$

Where $0 < M < 1$ for the mantissa M, $s = \sqrt{2}$, and $f \in \mathbb{Z}$ so that $M\,s^f \in [1/2,\, 3/2]$. The quantity $\log(s) = \log(\sqrt{2})$ can be precomputed directly via the AGM. A C++ implementation is given in [hfloat: src/tz/log.cc]:

There is a nice way to compute the value of $\log(R)$ if the value of π has been precomputed. One defines

$$\Theta_2(q) = \sum_{n=-\infty}^{\infty} q^{(n+1/2)^2} = 2\left(q^{1/4} + q^{9/4} + q^{25/4} + q^{81/4} + q^{121/4} + \ldots\right) \tag{31.1-4a}$$

$$= 2\,q^{1/4}\left(1 + q^2 + q^6 + q^{12} + q^{20} + \ldots\right) \tag{31.1-4b}$$

$$\Theta_3(q) = \sum_{n=-\infty}^{\infty} q^{n^2} = 1 + 2\left(q + q^4 + q^9 + q^{16} + q^{25} + \ldots\right) \tag{31.1-4c}$$

$$\Theta_4(q) = \sum_{n=-\infty}^{\infty} (-1)^n\, q^{n^2} = 1 + 2\left(-q + q^4 - q^9 + q^{16} - q^{25} \pm \ldots\right) \tag{31.1-4d}$$

Then (with K the elliptic function, see section 30.2.1 on page 575)

$$\log\frac{1}{q} = -\log q = \pi\,\frac{K'}{K} \tag{31.1-5}$$

thereby $q = \exp(-\pi\,K'/K)$, and

$$\frac{\pi}{\log(1/q)} = -\frac{\pi}{\log(q)} = \mathrm{AGM}\left(\Theta_3^2(q),\,\Theta_2^2(q)\right) = \frac{\mathrm{AGM}(1,\,k)}{\mathrm{AGM}(1,\,k')} \tag{31.1-6}$$

Computing $\Theta_3(q)$ is easy if $q = 1/R$:

$$\Theta_3(q) = 1 + 2\sum_{n=1}^{\infty} q^{n^2} = 2\,(1 + \sum_{n=1}^{\infty} q^{n^2}) - 1 \tag{31.1-7}$$

However, the computation of $\Theta_2(q)$ suggests to choose $q = 1/R^4 =: b^4$:

$$\Theta_2(q) = 0 + 2\sum_{n=0}^{\infty} q^{(n+1/2)^2} = 2\sum_{n=0}^{\infty} b^{4n^2+4n+1} \quad \text{where} \quad q = b^4 \tag{31.1-8a}$$

$$= 2\,b\sum_{n=0}^{\infty} q^{n^2+n} = 2\,b\,(1 + \sum_{n=1}^{\infty} q^{n^2+n}) \tag{31.1-8b}$$

Functions that compute $\Theta_2(b^4)$, $\Theta_3(b^4)$ and $\pi/\log(b)$ where b is the inverse fourth power of the used radix R are given in [hfloat: src/tz/pilogq.cc].

We list a few relations involving the theta functions (see, for example [245]):

$$\Theta_3^4(q) = \Theta_2^4(q) + \Theta_4^4(q) \tag{31.1-9}$$

$$\Theta_3(q) = \Theta_3(q^4) + \Theta_2(q^4) \qquad \Theta_4(q) = \Theta_3(q^4) - \Theta_2(q^4) \tag{31.1-10}$$

$$\Theta_2^2(q) = \frac{2\,k\,K}{\pi} \qquad \Theta_3^2(q) = \frac{2\,K}{\pi} \qquad \Theta_4^2(q) = \frac{2\,k'\,K}{\pi} \tag{31.1-11}$$

$$k = \frac{\Theta_2^2(q)}{\Theta_3^2(q)} \qquad k' = \frac{\Theta_4^2(q)}{\Theta_3^2(q)} \tag{31.1-12}$$

$$1 = \mathrm{AGM}\left(\Theta_3^2(q), \Theta_4^2(q)\right) \tag{31.1-13}$$

$$q\prod_{n=1}^{\infty}(1-q^n)^{24} = \frac{256}{\pi^{12}}\,k^2\,k'^8\,K^{12} \tag{31.1-14}$$

A discussion of the AGM-based computation of logarithms can be found in [39].

31.1.2 Computation by inverting the exponential function

With an efficient algorithm for the exponential function one can compute the logarithm using

$$\begin{aligned} y &:= 1 - d\,e^{-x} && \text{(31.1-15)}\\ \log(d) &= x + \log(1-y) && \text{(31.1-16)}\\ &= x + \log\left(1-(1-d\,e^{-x})\right) \;=\; x + \log\left(e^{-x}d\right) \;=\; x + (-x + \log(d)) && \text{(31.1-17)} \end{aligned}$$

Thereby,

$$\log(d) \;=\; x + \log(1-y) \;=\; x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \frac{y^4}{4} + \dots\right) \qquad \text{(31.1-18)}$$

Truncation of the series before the n-th power of y gives an iteration of order n:

$$\begin{aligned} x_{k+1} &= \Phi_n(x_k) \quad \text{where} && \text{(31.1-19a)}\\ \Phi_n(x) &:= x - \left(y + \frac{y^2}{2} + \frac{y^3}{3} + \dots + \frac{y^{n-1}}{n-1}\right) && \text{(31.1-19b)} \end{aligned}$$

31.1.2.1 Padé approximants for the logarithm

Padé approximants $P_{[i,j]}(z)$ of $\log(1-z)$ at $z=0$ produce iterations of order $i+j+1$. Compared to the power series based iteration one needs one additional long division but saves half of the exponentiations. This can be a substantial saving for high order iterations.

The approximants can be computed via the continued fraction expansion of $\log(1+z)$:

$$\log(1+z) \;=\; 0 + \cfrac{c_1\,z}{1 + \cfrac{c_2\,z}{1 + \cfrac{c_3\,z}{1 + \cfrac{c_4\,z}{1+\cdots}}}} \qquad \text{(31.1-20)}$$

where $c_1 = 1$ and

$$c_k \;=\; \frac{k}{4(k-1)} \quad \text{if } k \text{ even}, \qquad c_k \;=\; \frac{k-1}{4k} \quad \text{else} \qquad \text{(31.1-21)}$$

Using recurrence relations 35.3-7a and 35.3-7b on page 682 with $a_0 = 0$, $a_k = 1$ and $b_k = c_k \cdot z$ we obtain (fractions in lowest terms):

$$\begin{aligned} 1 &\mapsto P_{[1,0]} = \frac{z}{1} && \text{(31.1-22a)}\\ 2 &\mapsto P_{[1,1]} = \frac{2\,z}{2+\,z} && \text{(31.1-22b)}\\ 3 &\mapsto P_{[2,1]} = \frac{6\,z+\,z^2}{6+4\,z} && \text{(31.1-22c)}\\ 4 &\mapsto P_{[2,2]} = \frac{6\,z+3\,z^2}{6+6\,z+\,z^2} && \text{(31.1-22d)}\\ 5 &\mapsto P_{[3,2]} = \frac{30\,z+21\,z^2+\,z^3}{30+36\,z+9\,z^2} && \text{(31.1-22e)}\\ 6 &\mapsto P_{[3,3]} = \frac{60\,z+60\,z^2+11\,z^3}{60+90\,z+36\,z^2+3\,z^3} && \text{(31.1-22f)}\\ 7 &\mapsto P_{[4,3]} = \frac{420\,z+510\,z^2+140\,z^3+3\,z^4}{420+720\,z+360\,z^2+48\,z^3} && \text{(31.1-22g)}\\ 8 &\mapsto P_{[4,4]} = \frac{420\,z+630\,z^2+260\,z^3+25\,z^4}{420+840\,z+540\,z^2+120\,z^3+6\,z^4} && \text{(31.1-22h)} \end{aligned}$$

The expressions are Padé approximants correct up to order k. For even k these are the diagonal approximants $[k/2, k/2]$ which satisfy the functional equation $\log(1/z) = -\log(z)$: $P(1/z-1) = -P(z-1)$. Further information like the error term of the diagonal approximants is given in [173].

The diagonal Padé approximants can be computed by setting $P_0 = 0$, $Q_0 = 1$, $P_2 = z$, $Q_2 = 1 + z/2$, and computing, for $k = 4, 6, \ldots, 2n$,

$$P_k = A_k P_{k-2} + B_k P_{k-4} \tag{31.1-23a}$$
$$Q_k = A_k Q_{k-2} + B_k Q_{k-4} \tag{31.1-23b}$$

(these are relations 35.3-14a and 35.3-14b on page 685). The A_k and B_k are defined as

$$A_k = 1 + z/2 \tag{31.1-23c}$$
$$B_k = \frac{z^2}{16} \frac{(k-2)^2}{1-(k-2)^2} \tag{31.1-23d}$$

Then P_{2n}/Q_{2n} is the Padé approximant $[n, n]$ of $\log(1+z)$ which is correct up to order $2\,n$. The following pari/gp function implements the algorithm:

```
log_pade(n, z='z)=
{ /* Return Pade approximant [n,n] of log(1+z) */
    local(P0,Q0,P2,Q2,tp,tq, t);
    if ( n<1, return(0) );
    P0=0;  Q0=1;
    P2=z;  Q2=1+z/2;
    forstep (k=4, 2*n, 2,
        Ak = 1+z/2;  \\ == +z*C(k-1)+z*C(k)+1;
        t = (k-2)^2;
        Bk = z^2/16*t/(1-t);  \\ == -z^2*C(k-1)*C(k-2);
        tp = Ak*P2 + Bk*P0;
        tq = Ak*Q2 + Bk*Q0;
        P0=P2; P2=tp;
        Q0=Q2; Q2=tq;
    );
    return( P2/Q2 );
}
```

31.1.2.2 Padé approximants for arctan *

A continued fraction for arctan is (given in [172, p.569])

$$\arctan(z) = z \cfrac{1}{1 + \cfrac{z^2/(1\cdot 3)}{1 + \cfrac{2^2 z^2/(3\cdot 5)}{1 + \cfrac{3^2 z^2/(5\cdot 7)}{1 + \cdots}}}} \tag{31.1-24}$$

Padé approximants P_k/Q_k for the inverse tangent can be obtained by setting $P_0 = 0$, $P_1 = z$, $Q_0 = 1$, $Q_1 = 1$, and the recurrences

$$P_{k+1} = P_k + P_{k-1} \frac{k^2 z^2}{4k^2 - 1} \tag{31.1-25a}$$
$$Q_{k+1} = Q_k + P_{k-1} \frac{k^2 z^2}{4k^2 - 1} \tag{31.1-25b}$$

The first few approximants are

$$2 \mapsto P_{[1,2]} = \frac{3\,x}{3+x^2} \tag{31.1-26a}$$

$$3 \mapsto P_{[3,2]} = \frac{15\,x+4\,x^3}{15+9\,x^2} \tag{31.1-26b}$$

$$4 \mapsto P_{[3,4]} = \frac{105\,x+55\,x^3}{105+90\,x^2+9\,x^4} \tag{31.1-26c}$$

$$5 \mapsto P_{[5,4]} = \frac{945\,x+735\,x^3+64\,x^5}{945+1050\,x^2+225\,x^4} \tag{31.1-26d}$$

$$6 \mapsto P_{[5,6]} = \frac{1155\,x+1190\,x^3+231\,x^5}{1155+1575\,x^2+525\,x^4+25\,x^6} \tag{31.1-26e}$$

$$k \mapsto P_{[i,j]} = \arctan(z)+O(z^{2k+1}) \tag{31.1-26f}$$

31.1.3 Argument reduction for the logarithm

When the logarithm shall be computed for high but not extreme precision (up to several hundred decimal digits or so), the following scheme can beat the AGM algorithm. Use the functional equation for the logarithm

$$\log(z^a) = a\,\log(z) \tag{31.1-27}$$

to reduce the argument by setting $a = 1/N$. Now with N big enough $z^{1/N}$ will be close to one: $r := z^{1/N} = 1+e$ where e is small. Then a few terms of the Taylor series of $\log(1+e) = e - e^2/2 + e^3/3 \pm \ldots$ suffice to compute the logarithm. Compute the logarithm of z as follows

1. set $r = z^{1/N}$ and $e = r-1$
2. compute $l := \log(1+e)$ to the desired precision using the Taylor series
3. return $L := N\,l$

One can also use a Padé approximant in step 2. With argument $z = 2.0$, $N = 2^{32}$, and four terms of the Taylor series we obtain:

```
? z=2.0;   \\ argument for log()
? n=32; N=2^n;
? r=z^(1/N)  \\ compute by 32 sqrt extractions
  1.00000000016138590420965976120397663110198503274461201650532657 85 E-10
? e=r-1; \\ small
  1.61385904209659761203976631101985032744612016505326578 5 E-10
? l=e-1/2*e^2+1/3*e^3-1/4*e^4  \\ approx log(1+e)
  1.61385904196637056166593013670802248659405413369314055 0 E-10
? L=N*l     \\ final result
  0.693147180559945309417232121458176568075406093226565036 5
? log(z)  \\ check with builtin log
  0.693147180559945309417232121458176568075500134360255254 1
```

One may also use the following reduction for $L(z) := \log(1+z)$, which avoids loss of precision for small values of z:

$$L(z) = 2\,L\left(\frac{z}{1+\sqrt{1+z}}\right) \tag{31.1-28}$$

31.1.4 Argument reduction for arctan

We use the equation

$$\arctan(z) = 2\,\arctan\left(\frac{z}{1+\sqrt{1+z^2}}\right) \tag{31.1-29}$$

Compute the inverse tangent of z as follows

1. set $r := z$
2. repeat n times: $r = r/(1+\sqrt{1+r^2})$ (for n big enough)
3. compute $a := \arctan(r)$ to the desired precision using the Taylor series
4. return $A := 2^n\, a$

We compute arctan(1.0) using $n = 16$ and four terms of the Taylor series:

```
? z=1.0;
? n=16;
? r=z;for(k=1,n,r=r/(1+sqrt(1+r^2)));r
  0.000011984224905930303478511634794650661319585128744025261 89
? a=r-1/3*r^3+1/5*r^5-1/7*r^7
  0.000011984224905356572107172559292905818497455676569676602 63
? A=2^n*a
  0.785398163397448309615660845819875721049255219670325829 9
? atan(z)  \\ check with builtin atan
  0.785398163397448309615660845819875721049292349843776455 2
```

All divisions in the reduction phase can be saved by using

$$\arctan(1/z) \quad = \quad 2\,\arctan\left(\frac{1}{1+\sqrt{1+z^2}}\right) \tag{31.1-30}$$

The inverse sine and cosine can be computed as

$$\arcsin(z) \quad = \quad \arctan\left(\frac{z}{\sqrt{1-z^2}}\right) \tag{31.1-31a}$$

$$\arccos(z) \quad = \quad \arctan\left(\frac{\sqrt{1-z^2}}{z}\right) \tag{31.1-31b}$$

31.1.5 A curious series for the logarithm *

We finally note two relations resembling the well-known series

$$\frac{1}{2}\,\log\left(\frac{1+x}{1-x}\right) \quad = \quad x + \frac{1}{3}\,x^3 + \frac{1}{5}\,x^5 + \ldots + \frac{1}{2k+1}\,x^{2k+1} + \ldots \tag{31.1-32}$$

The first is

$$\frac{1}{6}\,\log\left(\frac{1+3x+3x^2}{1-3x+3x^2}\right) \quad = \quad x - \frac{3^2}{5}\,x^5 - \frac{3^3}{7}\,x^7 + \frac{3^5}{11}\,x^{11} + \frac{3^6}{13}\,x^{13} \pm \ldots = \tag{31.1-33a}$$

$$= \quad \sum_{k=0}^{\infty}\left(+\frac{3^{6k+0}}{12k+1}\,x^{12k+1} - \frac{3^{6k+2}}{12k+5}\,x^{12k+5} - \frac{3^{6k+3}}{12k+7}\,x^{12k+7} + \frac{3^{6k+5}}{12k+11}\,x^{12k+11}\right) \tag{31.1-33b}$$

The second, given in [26], is

$$\frac{1}{3}\,\log\left(\frac{1+x+x^2}{1-2\,x+x^2}\right) \quad = \quad x + \frac{x^2}{2} + \frac{x^4}{4} + \frac{x^5}{5} + \ldots \;=\; \sum_{k=0}^{\infty}\left(+\frac{x^{3k+1}}{3k+1} + \frac{x^{3k+2}}{3k+2}\right) \tag{31.1-34}$$

Relation 31.1-33a can be brought in a similar form, namely

$$\frac{1}{2\sqrt{3}}\,\log\left(\frac{1+\sqrt{3}\,x+x^2}{1-\sqrt{3}\,x+x^2}\right) \quad = \quad x - \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^{11}}{11} + \frac{x^{13}}{13} \pm \ldots = \tag{31.1-35a}$$

$$= \quad \sum_{k=0}^{\infty}\left(+\frac{x^{12k+1}}{12k+1} - \frac{x^{12k+5}}{12k+5} - \frac{x^{12k+7}}{12k+7} + \frac{x^{12k+11}}{12k+11}\right) \tag{31.1-35b}$$

31.2 Exponential function

31.2.1 AGM-based computation of the exponential function

We use the following relation (see [150, pp.35-36]):

$$q \;=\; \exp\left(-\pi\,\frac{K'}{K}\right) \tag{31.2-1}$$

Now write

$$\frac{K'}{K} \;=\; \frac{\mathrm{AGM}(1,k')}{\mathrm{AGM}(1,k)} \;=\; \frac{\mathrm{AGM}(1,b_0)}{\mathrm{AGM}(1,b'_0)} \tag{31.2-2}$$

where $k' = b_0$ and $k = b'_0 = \sqrt{1-b_0^2}$ and use ([150, p.38], note the missing '4' there)

$$\frac{\pi}{2}\,\frac{\mathrm{AGM}(1,b_0)}{\mathrm{AGM}(1,b'_0)} \;=\; \lim_{n\to\infty}\frac{1}{2^n}\log\frac{4\,a_n}{c_n} \tag{31.2-3}$$

thereby

$$q \;=\; \exp\left(-2\lim_{n\to\infty}\frac{1}{2^n}\,\log\frac{4\,a_n}{c_n}\right) \;=\; \lim_{n\to\infty}\exp\left(-\frac{1}{2^{n-1}}\,\log\frac{4\,a_n}{c_n}\right) \tag{31.2-4a}$$

$$\;=\; \lim_{n\to\infty}\left(\exp\log\frac{4\,a_n}{c_n}\right)^{-1/2^{n-1}} \;=\; \lim_{n\to\infty}\left(\frac{4\,a_n}{c_n}\right)^{-1/(2^{n-1})} \tag{31.2-4b}$$

This gives

$$q \;=\; \lim_{n\to\infty}\left(\frac{c_n}{4\,a_n}\right)^{1/(2^{n-1})} \tag{31.2-5}$$

One obtains an algorithm for $\exp(-x)$ by first solving for k, k' so that $x = \pi\,K'/K$ (precomputed π) and applying the last relation that implies the computation of a 2^{n-1}-th root. Note that the quantity c should be computed via $c_{n+1} = \frac{c_n^2}{4\,a_{n+1}}$ throughout the AGM computation in order to preserve its accuracy.

For $k = 1/\sqrt{2} =: s$ one has $k = k'$ and so $q = \exp(-\pi)$. Thus the calculation of $\exp(-\pi) = 0.0432139182637\ldots$ can directly be done via a single AGM computation as $(c_n/(4a_n))^N$ where $N = 1/2^{(n-1)}$. The quantity $i^i = \exp(-\pi/2) = 0.2078795763507\ldots$ can be obtained using $N = 1/2^n$.

31.2.2 Computation by inverting the logarithm

The exponential function can be computed using the iteration that is obtained as follows:

$$\exp(d) \;=\; x\,\exp\left(d-\log(x)\right) \tag{31.2-6}$$

$$\;=\; x\,\exp(y) \quad \text{where} \quad y := d-\log(x) \tag{31.2-7}$$

$$\;=\; x\left(1+y+\frac{y^2}{2}+\frac{y^3}{3!}+\frac{y^4}{4!}+\ldots\right) \tag{31.2-8}$$

The corresponding n-th oder iteration is

$$x_{k+1} \;=\; \Phi_n(x_k) \quad \text{where} \tag{31.2-9a}$$

$$\Phi_n(x) \;:=\; x_k\left(1+y+\frac{y^2}{2}+\frac{y^3}{3!}+\ldots+\frac{y^{n-1}}{(n-1)!}\right) \tag{31.2-9b}$$

As the computation of logarithms is expensive one should use an iteration of high order. The C++ implementation given in [hfloat: src/tz/itexp.cc] uses the iteration of order 20.

31.2.2.1 Padé approximants for exp

Padé series $P_{[i,j]}(z)$ of $\exp(z)$ around $z=0$ give iterations of order $i+j+1$. For $i=j$ we obtain

$$P_{[1,1]} = \frac{2+z}{2-z} \tag{31.2-10a}$$

$$P_{[2,2]} = \frac{12+6z+z^2}{12-6z+z^2} \tag{31.2-10b}$$

$$P_{[3,3]} = \frac{120+60z+12z^2+z^3}{120-60z+12z^2-z^3} \tag{31.2-10c}$$

$$P_{[4,4]} = \frac{1680+840z+180z^2+20z^3+z^4}{1680-840z+180z^2-20z^3+z^4} \tag{31.2-10d}$$

Note that the functional equation $\exp(-z)=\frac{1}{\exp(z)}$ holds for the (diagonal) Padé approximants.

A closed form for the $[i,j]$-th Padé approximant of $\exp(z)$ is

$$P_{[i,j]}(z) = \left\{\sum_{k=0}^{i} \frac{\binom{i}{k}}{\binom{i+j}{k}} \frac{z^k}{k!}\right\} / \left\{\sum_{k=0}^{j} \frac{\binom{j}{k}}{\binom{i+j}{k}} \frac{(-z)^k}{k!}\right\} \tag{31.2-11}$$

The numerator for $i=j$, multiplied by $(2i)!/i!$ in order to avoid rational coefficients, equals

$$= \frac{(2i)!}{i!} \cdot \sum_{k=0}^{i} \frac{\binom{i}{k}}{\binom{2\,i}{k}} \frac{z^k}{k!} \tag{31.2-12}$$

The coefficients of the numerator and denominator in the diagonal approximant

$$P_{[i,i]} = \frac{\sum_{k=0}^{i} c_k\, z^k}{\sum_{k=0}^{i} c_k\, (-z)^k} \tag{31.2-13}$$

can be computed using $c_i=1$ (the coefficient of the highest power of z) and the recurrence

$$c_k = c_{k+1}\, \frac{(k+1)\,(2i-k)}{(i-k)} \tag{31.2-14}$$

It is usually preferable to generate the coefficients in the other direction. To do so compute the constant c_0

$$c_0 = \prod_{w=1}^{i} 4\,w-2 \;=\; 2,\ 12,\ 120,\ 1680,\ 30240,\ \ldots \tag{31.2-15}$$

and use the recurrence

$$c_k = c_{k-1}\, \frac{i-k}{(2i-k)\,(k+1)} \tag{31.2-16}$$

We generate the coefficients for $1 \le i \le 8$:

```
? c0(i)=prod(w=1,i,4*w-2)
? qq(i,k)=(i-k)/((2*i-k)*(k+1))
? for (i=1, 8, c=c0(i); print1("["i,","i,"]:  "); \
?   for (k=0, i, print1("  ", c); c*=qq(i,k)); print();)
  [1,1]:    2  1
  [2,2]:    12  6  1
  [3,3]:    120  60  12  1
  [4,4]:    1680  840  180  20  1
  [5,5]:    30240  15120  3360  420  30  1
  [6,6]:    665280  332640  75600  10080  840  42  1
  [7,7]:    17297280  8648640  1995840  277200  25200  1512  56  1
  [8,8]:    518918400  259459200  60540480  8648640  831600  55440  2520  72  1
```

Finally, the Padé approximant $P_{[i,j]}$ can be expressed as ratio of hypergeometric functions:

$$P_{[i,j]} \quad = \quad F\left(\begin{matrix}-i\\-i-j\end{matrix}\Big|\, z\right) / F\left(\begin{matrix}-j\\-i-j\end{matrix}\Big|\, -z\right) \tag{31.2-17}$$

This is relation 35.2-51 on page 670 with $a = -i$ and $b = -j$ where i and j are positive integers.

31.2.3 Argument reduction for the exponential function

As for the logarithm an argument reduction technique can be useful with moderate precisions. We do not use the functional equation for the exponential functional ($\exp(2z) = \exp(z)^2$) because the loss of precision when adding up the Taylor series (one plus a tiny quantity). Instead we use the functional equation for $E(z) := \exp(z) - 1$:

$$E(2z) \quad = \quad 2\,E(z) + E^2(z) \tag{31.2-18}$$

Compute the exponential function of z as follows

1. set $r = z/2^n$ (for n big enough)
2. compute $E := \exp(r) - 1$ to the desired precision using the Taylor series
3. repeat n times: $E = 2\,E + E^2$
4. return $E + 1$

We compute exp(1.0) using $n = 16$ and eight terms of the Taylor series:

```
? z=1.0;
? n=16;
? r=z/2^n
  0.000015258789062500000000000000000000000000000000000000000000000
? E=r*(1+r/2*(1+r/3*(1+r/4*(1+r/5*(1+r/6*(1+r/7*(1+r/8)))))))
  0.000015258905478413948140042622480661730187012348455116225 83
? for(k=1,n,E=2*E+E^2);E=E+1
  2.718281828459045235360287471352662497757247071686614582
? exp(1.0)  \\ check with builtin exp()
  2.718281828459045235360287471352662497757247093699959575
```

One can also compute the exponential function via the hyperbolic cosine using

$$\exp(z) \quad = \quad \cosh(z) + \sinh(z) \;=\; \cosh(z) + \sqrt{\cosh^2(z) - 1} \tag{31.2-19a}$$

Alternatively, compute hyperbolic sine and use

$$\exp(z) \quad = \quad \sinh(z) + \sqrt{1 + \sinh(z)^2} \tag{31.2-19b}$$

The advantage is that half of the coefficients of the Taylor series are zero. Again we do not use the function equation for the cosine ($\cosh(2z) = 2\,\cosh^2(z) - 1$) but that for $C(z) := \cosh(z) - 1$:

$$C(2z) \quad = \quad 2\,(C(z) + 1)^2 - 2 \tag{31.2-20}$$

Compute the hyperbolic cosine as follows

1. set $r = z/2^n$ (for n big enough)
2. compute $C := \cosh(r) - 1$ to the desired precision using the Taylor series
3. repeat n times: $C = 2\,[C - 1]^2 - 2$
4. return $C + 1$

We compute cosh(1.5) using $n = 16$ and four terms of the Taylor series:

```
? z=1.50;
? n=16;
? r=z/2^n
? C=1/2*r^2+1/24*r^4+1/720*r^6+1/40320*r^8
  0.000000000261934474122038277307663984994058290343310054242645592 3
? for(k=1,n,C=2*(C+1)^2-2);C
  1.352409615243247325767667965441644170173960682574839216
? C+1
  2.352409615243247325767667965441644170173960682574839216
? cosh(z) \\ check with builtin cosh
  2.352409615243247325767667965441644170173960748865373192
```

When the Taylor series for $\cos(z)-1$ is used then the cosine can be obtained by the identical algorithm:

```
? z=1.50;
? n=16;
? r=z/2^n
  0.000022888183593750000000000000000000000000000000000000000000
? C=-1/2*r^2+1/24*r^4-1/720*r^6+1/40320*r^8
  -0.000000000261934474099168387731797875839209846883179633343528818 1
? for(k=1,n,C=2*(C+1)^2-2);C
  -0.929262798332297089911810148565731290914908941381793162 3
? C+1
  0.070737201667702910088189851434268709085091058618206837 69
? cos(z)  \\ check with builtin cos
  0.070737201667702910088189851434268709085091027563346869 42
```

Compute the sine as $\sin(z)=\sqrt{1-\cos^2(z)}$, and the tangent as $\tan(z)=\sin(z)/\cos(z)$.

31.3 Logarithm and exponential function of power series

The computation of the logarithm, the exponential function, and the inverse trigonometric functions turns out to be surprisingly simple with power series.

31.3.1 Logarithm

Let $f(x)$ be a power series in x and $g(x)=\log(f(x))$. Then we have $\frac{d\,g(x)}{dx}=\frac{f'(x)}{f(x)}$. Thereby, symbolically,

$$g(x) \;=\; \log(f(x)) \;=\; \int \frac{f'(x)}{f(x)}\,dx \tag{31.3-1}$$

A few lines of pari/gp to demonstrate this:

```
? sp=8;default(seriesprecision,sp+1);
? f=taylor((1)/(1-x-x^2),x) /* shifted Fibonacci (with constant term) */
 1 + x + 2*x^2 + 3*x^3 + 5*x^4 + 8*x^5 + 13*x^6 + 21*x^7 + 34*x^8 + O(x^9)

? d=deriv(f,x)
 1 + 4*x + 9*x^2 + 20*x^3 + 40*x^4 + 78*x^5 + 147*x^6 + 272*x^7 + O(x^8)

? q=d/f  /* the only nontrivial computation */
 1 + 3*x + 4*x^2 + 7*x^3 + 11*x^4 + 18*x^5 + 29*x^6 + 47*x^7 + O(x^8)

? lf=intformal(q)
 x + 3/2*x^2 + 4/3*x^3 + 7/4*x^4 + 11/5*x^5 + 3*x^6 + 29/7*x^7 + 47/8*x^8 + O(x^9)

? f-exp(lf)  /* check with builtin exp() */
 O(x^9)
```

31.3.2 Inverse trigonometric functions

Now let $a(x)=\arctan(f(x))$. Then, symbolically,

$$a(x) \;=\; \int \frac{f'(x)}{1+f(x)^2}\,dx \tag{31.3-2}$$

Verification for the trivial case $f(x)=x$:

```
? sp=13;default(seriesprecision,sp+1);
? f=taylor(x,x)
 x + O(x^14)
? d=deriv(f,x)
 1 + O(x^13)
? q=d/(1+f^2)
 1 - x^2 + x^4 - x^6 + x^8 - x^10 + x^12 + O(x^13)
? af=intformal(q,x)
 x - 1/3*x^3 + 1/5*x^5 - 1/7*x^7 + 1/9*x^9 - 1/11*x^11 + 1/13*x^13 + O(x^14)
? f-tan(af)  /* check with builtin tan() */
 O(x^14)
```

For $s(x) = \arcsin(f(x))$ use

$$s(x) \quad = \quad \int \frac{f'(x)}{\sqrt{1-f(x)^2}}\, dx \tag{31.3-3}$$

31.3.3 Exponential function

With $e(x) = \exp(f(x))$ we can use a scheme similar to those shown in section 28.7 on page 558. We express a function $g(y)$ as

$$g(y) \quad = \quad \prod_{k=1}^{\infty} [1 + T(Y_k)] \tag{31.3-4}$$

where $Y_1 = y$, $Y_{k+1} = N(Y_k)$ and $1 + T(y)$ is the truncated Taylor series of g. A second order product is obtained by taking $1 + T(y) = 1 + y$ (the series of $\exp(y)$ truncated before the second term) and

$$N(y) \quad = \quad f^{-1}\left(\frac{f(y)}{1+T(y)}\right) \tag{31.3-5}$$

For $g(y) = \exp(y)$ one obtains $N(y) = y - \log(1+y)$ and

$$\exp(y) \quad = \quad [1+y] \cdot [1+y-\log(1+y)] \cdot [1+y-\log(1+y)-\log(1+y-\log(1+y))] \cdots \tag{31.3-6a}$$

$$= \quad \prod_{k=1}^{\infty} [1+Y_k] \tag{31.3-6b}$$

where $Y_1 = y = f(x)$ and $Y_{k+1} = Y_k - \log(1+Y_k)$. The product $\prod_{k=1}^{N}$ is correct up to order y^{2^N-1}. The computation involves $N-2$ logarithms and $N-1$ multiplications. Implementation in pari/gp:

```
texp(y, N=5)=
{
    local(Y, e, t);
    Y = y;  e = 1 + Y;
    for (k=2, N,
        t = deriv(1+Y,x)/(1+Y);
        t = intformal(t);  \\ here: t = log(1+Y);
        Y -= t;
        e *= (1+Y);
    );
    return( e );
}
```

Check:

```
? f=taylor((x)/(1-x-x^2),x)
  x + x^2 + 2*x^3 + 3*x^4 + 5*x^5 + 8*x^6 + 13*x^7 + 21*x^8 + ...
? e=exp(f)  /* builtin exp() */
  1 + x + 3/2*x^2 + 19/6*x^3 + 145/24*x^4 + 467/40*x^5 + 16051/720*x^6 + ...
? t=texp(f,4);
? t-e
  -1/32768*x^16 - 35/98304*x^17 - ...
```

31.4 Simultaneous computation of logarithms of small primes

We describe a method to compute the logarithms of a given set of (small) primes simultaneously. We use a method similarly to the one for finding arctan-relations for π given in section 30.5 on page 590. We define

$$L(z) \quad := \quad 2\,\mathrm{arccoth}(z) \;=\; 2\sum_{k=0}^{\infty}\frac{1}{(2k+1)\,z^{2k+1}} \tag{31.4-1}$$

and note that (relation 35.2-83d on page 675)

$$\log(z) \quad = \quad 2\,\mathrm{arccoth}\,\frac{z+1}{z-1} \tag{31.4-2}$$

We will determine a set of relations that express the logarithm of a prime as linear combination of terms $L(X_i)$ where the X_i are large integers so that the series for L converges fast.

```
S = { 51744295, 170918749, 265326335, 287080366, 362074049, 587270881,
      617831551, 740512499, 831409151, 1752438401, 2151548801, 2470954914, 3222617399 }

2:  [ -1595639, -17569128, -8662593, -31112926, -13108464, -11209640,
      -12907342, +9745611, -1705229, -12058985, +4580610, +4775383, -12972664 ]

3:  [ -2529028, -27846409, -13729885, -49312821, -20776424, -17766859,
      -20457653, +15446428, -2702724, -19113039, +7260095, +7568803, -20561186 ]

5:  [ -3704959, -40794252, -20113918, -72241977, -30436911, -26027978,
      -29969920, +22628608, -3959419, -28000096, +10635847, +11088096, -30121593 ]

7:  [ -4479525, -49322778, -24318973, -87345026, -36800111, -31469438,
      -36235490, +27359389, -4787183, -33853851, +12859398, +13406195, -36418872 ]

11: [ -5520004, -60779197, -29967648, -107633040, -45347835, -38778983,
      -44652067, +33714275, -5899123, -41717234, +15846307, +16520111, -44878044 ]

13: [ -5904566, -65013499, -32055403, -115131507, -48507081, -41480597,
      -47762841, +36063046, -6310097, -44623547, +16950271, +17671017, -48004561 ]

17: [ -6522115, -71813158, -35408027, -127172929, -53580360, -45818987,
      -52758281, +39834823, -6970060, -49290653, +18723073, +19519201, -53025282 ]

19: [ -6778159, -74632382, -36798067, -132165454, -55683805, -47617738,
      -54829453, +41398649, -7243689, -51225694, +19458099, +20285481, -55106936 ]

23: [ -7217972, -79475039, -39185776, -140741248, -59296949, -50707501,
      -58387161, +44084875, -7713709, -54549566, +20720673, +21601741, -58682649 ]

29: [ -7751584, -85350490, -42082712, -151146003, -63680669, -54456218,
      -62703622, +47343993, -8283970, -58582320, +22252516, +23198720, -63020955 ]

31: [ -7905109, -87040909, -42916186, -154139543, -64941904, -55534757,
      -63945506, +48281670, -8448039, -59742579, +22693241, +23658185, -64269124 ]

37: [ -8312407, -91525553, -45127374, -162081337, -68287932, -58396097,
      -67240196, +50769306, -8883311, -62820720, +23862474, +24877135, -67580488 ]

41: [ -8548719, -94127517, -46410292, -166689119, -70229278, -60056229,
      -69151756, +52212618, -9135853, -64606639, +24540856, +25584363, -69501722 ]
```

Figure 31.4-A: Relations for the fast computation of the logarithms of the primes up to 41.

Compute $\log(p_i)$ for the primes p_i in a predefined set P of n primes as follows:

1. Find a set S of numbers $X \in \mathbb{Z}$ so that X^2-1 factor completely into the primes in P.
2. Select a subset S of n (large) numbers X_k so that all $L(X_k)$ are linearly independent.
3. Try to find, for each prime p_i, a relation $\log(p_i) = \sum_{j=1}^{n} m_j\,L(X_j)$. If this fails return to step 2.

For example, with the first 13 primes ($P = \{2, 3, 5, 7, 11, \ldots, 41\}$) one can find

$$\begin{aligned} S \quad = \quad & \{X_1, X_2, \ldots, X_{13}\} \;= \\ & \{51744295, 170918749, 265326335, 287080366, 362074049, 587270881, \\ & 617831551, 740512499, 831409151, 1752438401, 2151548801, 2470954914, 3222617399\} \end{aligned} \tag{31.4-3}$$

We use the short form `p: [m1, m2, m3, ..., m13]` to denote a relation

$$\log(p) \quad = \quad \sum_{j=1}^{13} m_j \, L(X_j) \tag{31.4-4}$$

Now we have the relations given in figure 31.4-A, the first one is

$$\log(2) \quad = \quad -1595639 \, L(51,744,295) - 17569128 \, L(170,918,749) \pm \ldots - 12972664 \, L(3,222,617,399)$$

Note that the series with slowest convergence already gives more than 15 digits per term ($\log_{10}(51,744,295^2) \approx 15.4$), and the last series gives 19 digits per term.

```
                  2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41

   51744295:  [ -2, +2,  0, +3,  0,  0, +2, -3, -1, +1,  0,  0, -1 ]
  170918749:  [ +1, +4, -5, +1, +1, +1, +1,  0, -1, -1, +1,  0, -1 ]
  265326335:  [ -7, -2,  0, +1, -1, +1,  0, -2,  0, -1, +2, +1, +1 ]

  287080366:  [  0, +1, +1, -5, +2, +1,  0, -1, +3, -1, -1,  0,  0 ]
  362074049:  [ +5, -4, -2, +1,  0, -2,  0,  0, -2, +2, +2,  0,  0 ]
  587270881:  [ +4, +2, +1, +3, -1,  0, -1,  0,  0, +1, -1, -3, +1 ]

  617831551:  [ -6, +4, +2, +1,  0, -6,  0, +1,  0,  0, +1, +1,  0 ]
  740512499:  [ -1, -1, -5, -2, +7, -1,  0, +1,  0,  0, -1,  0,  0 ]
  831409151:  [ -9, -1, +2, -1, +3, +1,  0,  0, -1,  0, +2,  0, -2 ]

 1752438401:  [ +6, -4, +2,  0, -2, -2,  0, +2, -2,  0,  0, +1, +1 ]
 2151548801:  [ +6, -2, +2,  0,  0, -2,  0,  0, +2, -4, +1,  0, +1 ]
 2470954914:  [  0,  0, -1, +1, -3, -5, +2,  0,  0,  0, +3,  0, +1 ]
 3222617399:  [ -2, -6, -2, +4, +1, +2,  0, +2, -1,  0, -2,  0,  0 ]
```

Figure 31.4-B: Values $L(x)$ as linear combinations of logarithms of small primes.

Figure 31.4-B shows the linear combinations of logarithms of small primes that give the values $L(x)$. The first row gives the relation

$$L(51,744,295) \quad = \quad -2 \, \log(2) + 2 \, \log(3) + 3 \, \log(7) \pm \ldots - 1 \, \log(41) \tag{31.4-5}$$

The shown values, as a matrix, are the inverse the values in figure 31.4-A.

Precomputed logarithms of small primes can be used for the computation of the logarithms of integers k if one can determine a smooth number near k. For example, the logarithm of 65537 (a prime) can be computed as

$$\log(65537) \quad = \quad \log\left(65537 \cdot \frac{65536}{65536}\right) \; = \; \log\left(\frac{65537}{65536}\right) + \log(65536) \tag{31.4-6a}$$

$$= \quad \log\left(1 + \frac{1}{65536}\right) + 16 \, \log(2) \tag{31.4-6b}$$

The series of the first logarithm converges fast and $\log(2)$ is precomputed. This idea has been given by Jim White [priv.comm.]. If k is not near a smooth number but $u \cdot k$ is smooth where u factors into the chosen prime set, one can use the relation

$$\log(k) \quad = \quad \log(u\,k) - \log(u) \tag{31.4-7}$$

Here $\log(u)$ is the sum of precomputed logarithms and with $\log(u\,k)$ one can proceed as above.

Chapter 32

Numerical evaluation of power series

We give algorithms for the numerical evaluation of power series. When the series coefficients are rational the binary splitting (binsplit) algorithm can be applied for rational arguments, and the rectangular schemes with real (full-precision) arguments. As a special case of the binary splitting algorithm, a method for fast radix conversion is described. Finally we describe a technique for the summation of series with alternating coefficients.

32.1 The binary splitting algorithm for rational series

The straightforward computation of a series for which each term adds a constant amount of precision (for example, the arc-cotangent series with arguments > 1) to a precision of N digits involves the summation of proportional N terms. To get N bits of precision one has to add proportional N terms of the sum, each term involves one (length-N) short division (and one addition). Therefore the total work is proportional N^2, which makes it impossible to compute billions of digits from linearly convergent series even if they are as 'good' as Chudnovsky's famous series for π (given in [77]):

$$\frac{1}{\pi} = \frac{6541681608}{\sqrt{640320}^3} \sum_{k=0}^{\infty} \left(\frac{13591409}{545140134} + k \right) \left(\frac{(6k)!}{(k!)^3\,(3k)!} \frac{(-1)^k}{640320^{3k}} \right) \tag{32.1-1a}$$

$$= \frac{12}{\sqrt{640320}^3} \sum_{k=0}^{\infty} (-1)^k \frac{(6k)!}{(k!)^3\,(3k)!} \frac{13591409 + 545140134 \cdot k}{(640320)^{3k}} \tag{32.1-1b}$$

Binary splitting scheme for products

We motivate the binsplit algorithm by giving the analogue for the fast computation of the factorial. Define $f_{m,n} := m \cdot (m+1) \cdot (m+2) \cdots (n-1) \cdot n$, then $n! = f_{1,n}$. We compute $n!$ by recursively using the relation $f_{m,n} = f_{m,x} \cdot f_{x+1,n}$ where $x = \lfloor (m+n)/2 \rfloor$:

```
indent(i)=for(k=1,8*i,print1(" "));  \\ aux: print 8*i spaces

F(m, n, i=0)=
{ /* Factorial, self documenting */
    local(x, ret);
    indent(i);  print( "F(", m, ", ", n, ")");
    if ( m==n, /* then: */
        ret = m;  \\ == F(m,m)
    , /* else: */
        x = floor( (m+n)/2 );
        ret = F(m, x, i+1) * F(x+1, n, i+1);
    );
```

```
F(1, 8)
        F(1, 4)
                F(1, 2)
                        F(1, 1)
                        ^== 1
                        F(2, 2)
                        ^== 2
                ^== 2
                F(3, 4)
                        F(3, 3)
                        ^== 3
                        F(4, 4)
                        ^== 4
                ^== 12
        ^== 24
        F(5, 8)
                F(5, 6)
                        F(5, 5)
                        ^== 5
                        F(6, 6)
                        ^== 6
                ^== 30
                F(7, 8)
                        F(7, 7)
                        ^== 7
                        F(8, 8)
                        ^== 8
                ^== 56
        ^== 1680
^== 40320
```

Figure 32.1-A: Quantities with the computation of 8!.

```
    indent(i);  print( "^== ", ret);
    return( ret );
}
```

The function prints the intermediate values occurring in the computation. The additional parameter `i` keeps track of the calling depth, used with the auxiliary function `indent()`. Figure 32.1-A shows the output with the computation of 8! =`F(1,8)`. A fragment like

```
F(5, 6)
        F(5, 5)
        ^== 5
        F(6, 6)
        ^== 6
^== 30
```

says "`F(5,6)` called `F(5,5)` [which returned 5], then called `F(6,6)` [which returned 6]. Then `F(5,6)` returned 30." For the computation of other products modify the line `ret=m;` as indicated in the code.

Binary splitting scheme for sums

For the evaluation of a sum $\sum_{k=0}^{N-1} a_k$ we use the ratios r_k of consecutive terms:

$$r_k \quad := \quad \frac{a_k}{a_{k-1}} \tag{32.1-2}$$

Set $a_{-1} := 1$ to avoid a special case for $k = 0$. One has

$$\sum_{k=0}^{N-1} a_k \quad =: \quad r_0\,(1 + r_1\,(1 + r_2\,(1 + r_3\,(1 + \ldots (1 + r_{N-1}) \ldots)))) \tag{32.1-3}$$

Now define

$$r_{m,n} \quad := \quad r_m\,(1 + r_{m+1}\,(\ldots (1 + r_n) \ldots)) \quad \text{where} \quad m < n \tag{32.1-4a}$$

$$r_{m,m} \quad := \quad r_m \tag{32.1-4b}$$

then

$$r_{m,n} \quad = \quad \frac{1}{a_{m-1}} \sum_{k=m}^{n} a_k \tag{32.1-5}$$

and especially

$$r_{0,n} = \sum_{k=0}^{n} a_k \tag{32.1-6}$$

We have

$$\begin{aligned} r_{m,n} &= r_m + r_m \cdot r_{m+1} + r_m \cdot r_{m+1} \cdot r_{m+2} + \dots \\ &\quad \dots + r_m \cdot \dots \cdot r_x + r_m \cdot \dots \cdot r_x \cdot [r_{x+1} + \dots + r_{x+1} \cdot \dots \cdot r_n] \end{aligned} \tag{32.1-7a}$$

$$= r_{m,x} + \prod_{k=m}^{x} r_k \cdot r_{x+1,n} \tag{32.1-7b}$$

The product telescopes, one gets (for $m \le x < n$)

$$r_{m,n} = r_{m,x} + \frac{a_x}{a_{m-1}} \cdot r_{x+1,n} \tag{32.1-8}$$

32.1.1 Implementation using rationals

```
R(0, 6)
        R(0, 3)
                R(0, 1)
                        R(0, 0)
                        ^== 1/2
                        R(1, 1)
                        ^== 1/2
                ^== 3/4
                R(2, 3)
                        R(2, 2)
                        ^== 1/2
                        R(3, 3)
                        ^== 1/2
                ^== 3/4
        ^== 15/16
        R(4, 6)
                R(4, 5)
                        R(4, 4)
                        ^== 1/2
                        R(5, 5)
                        ^== 1/2
                ^== 3/4
                R(6, 6)
                ^== 1/2
        ^== 7/8
^== 127/128
```

Figure 32.1-B: Quantities with the binsplit computation of $\sum_{k=0}^{6} 2^{-(k+1)} = 127/128$.

Now we can formulate the binary splitting algorithm by giving a binsplit function using pari/gp:

```
R(m, n)=
{ /* Rational binsplit */
    local(x, ret);
    if ( m==n, /* then: */
        ret = A(m)/A(m-1);
    , /* else: */
        x = floor( (m+n)/2 );
        ret = R(m, x) + A(x) / A(m-1) * R(x+1, n);
    );
    return( ret );
}
```

Here `A(k)` must be a function that returns the `k`-th term of the series we wish to compute, in addition one must have `a(-1)`=1. For example, to compute arctan(1/10) one would use

```
A(k)=if(k<0, 1, (-1)^(k)/((2*k+1)*10^(2*k+1)));
```

Figure 32.1-B shows the intermediate values with the computation of $\sum_{k=0}^{6} 2^{-(k+1)}$.

32.1.2 Implementation using integers

In case the programming language used does not provide rational numbers one needs to rewrite formula 32.1-8 in separate parts for denominator and numerator. With $a_i = p_i/q_i$, $p_{-1} = q_{-1} = 1$ and $r_{m,n} =: U_{m,n}/V_{m,n}$ one gets

$$U_{m,n} = p_{m-1}\, q_x\, U_{m,x}\, V_{x+1,n} + p_x\, q_{m-1}\, U_{x+1,n}\, V_{m,x} \tag{32.1-9a}$$

$$V_{m,n} = p_{m-1}\, q_x\, V_{m,x}\, V_{x+1,n} \tag{32.1-9b}$$

```
    Q(0, 6)
            Q(0, 3)
                    Q(0, 1)
                            Q(0, 0)
                            ^== [1, 10]
                            Q(1, 1)
                            ^== [-10, 3000]
                    ^== [29900, 300000]
                    Q(2, 3)
                            Q(2, 2)
                            ^== [3000, -500000]
                            Q(3, 3)
                            ^== [-500000, 70000000]
                    ^== [-104250000000000000, 17500000000000000000]
            ^== [1569781275000000000000000000, 157500000000000000000000000000]
     [--snip--]
```

Figure 32.1-C: Explosive growth of intermediate quantities with computing arctan(1/10).

The following implementation also contains code for reduction to lowest terms:

```
Q(m, n)=
{ /* Integer binsplit */
    local(x, ret, bm, bx, tm, tx);
    if ( m==n, /* then: */
        bm = B(m);  bx = B(m-1);
        ret = [ bm[1]*bx[2] , bx[1]*bm[2] ]; \\ == B(m)/B(m-1);
        x = gcd(ret[1], ret[2]);  /* Reduction */
        ret = [ret[1]/x, ret[2]/x];  /* Reduction */
    , /* else: */
        x = floor( (m+n)/2 );
        tm = Q(m, x);    \\ [U_{m,x}, V_{m,x}]
        tx = Q(x+1, n);  \\ [U_{x+1,n}, V_{m,n}]
        bm = B(m-1);     \\ [p_{m-1}, q_{m-1}]
        bx = B(x);       \\ [p_{x}, q_{x}]
        \\ ret == Q(m, x) + B(x) / B(m-1) * Q(x+1, n);
        ret = [ (bm[1]*bx[2]*tm[1]*tx[2] + bx[1]*bm[2]*tx[1]*tm[2])/10,
                (bm[1]*bx[2]*tm[2]*tx[2])/10 ];
        x = gcd(ret[1], ret[2]);  /* Reduction */
        ret = [ret[1]/x, ret[2]/x];  /* Reduction */
    );
    return( ret );
}
```

The reduction step can do good or bad, depending on the terms of the sum. When computing arctan(1/10) *without* the reduction, the intermediate quantities grow exponentially, as shown in figure 32.1-C. The square brackets are the quantities $[U_{m,n}, V_{m,n}]$. Such explosive growth will occur with all Taylor series unless the function argument equals one.

32.1.3 Performance

We compute the sum for arctan(1/10) up to the 5,000th term with the direct method, the rational binsplit and the integer binsplit with and without reduction. The timings for the computation are:

```
A(k)=if(k<0,1, (-1)^(k)/((2*k+1)*10^(2*k+1)));  \\ for rational binsplit
B(k)=if(k<0, [1,1], [(-1)^(k), ((2*k+1)*10^(2*k+1))] ); \\ for integer binsplit
N=5000;
```

```
  sum(k=0,N,A(k)); \\ direct method:  69,385 ms.
  R(0,N); \\ rational binsplit: 2,532 ms.
  Q(0,N); \\ integer binsplit with gcd reduction: 4,152 ms.
  Q(0,N); \\ integer binsplit without gcd reduction: >8min, "forever"
```

Things look quite different when computing the sum $\sum_{k=0}^{50,000} (-1)^k/(2k+1)^2$. The intermediate quantities U and V have only small common factors, so it is better to omit the reduction step:

```
B(k)=if(k<0, [1,1], [(-1)^k, (2*k+1)^2] );
A(k)=if(k<0,1, (-1)^(k)/(2*k+1)^2);
  N=50000;
  sum(k=0,N,A(k)); \\ direct method:  32,396 ms.
  R(0,N); \\ rational binsplit:  6,826 ms.
  Q(0,N); \\ integer binsplit with gcd reduction:  27,485 ms.
  Q(0,N); \\ integer binsplit without gcd reduction:  6,251 ms.
```

With builtin routines for binsplit summation the advantage will be much more in favor than these figures suggest.

The reason why summation via binary splitting is better than the straightforward way is that its complexity is only $O(\log N\, M(N))$, where $M(N)$ is the complexity of one N-bit multiplication (see [129]). If an FFT based multiplication algorithm is used ($M(N) = N \log N$) the work is $O((\log N)^2\, N)$ This means that sums of linear but sufficient convergence are again candidates for high precision computations. The algorithm should be implemented in the 'depth first' manner presented, and *not* via the naive pairs, pairs of pairs, etc. (breadth first) way. The reasons are better locality and less memory consumption. The naive way needs most memory after the first pass, when pairs have been multiplied.

32.1.4 Extending prior computations

The ratio $r_{0,N-1}$ (that is, the sum of the first N terms) can be reused if one wants to evaluate the sum to a higher precision than before. To get twice the precision use

$$r_{0,2\,N-1} \quad = \quad r_{0,N-1} + a_{N-1} \cdot r_{N,2\,N-1} \tag{32.1-10}$$

(this is formula 32.1-8 with $m = 0, x = N-1, n = 2N-1$). With explicit rational arithmetic:

$$U_{0,2N-1} \quad = \quad q_{N-1}\, U_{0,N-1}\, V_{N,2N-1} + p_{N-1}\, U_{N,2N-1}\, V_{0,N-1} \tag{32.1-11a}$$
$$V_{0,2N-1} \quad = \quad q_{N-1}\, V_{0,N-1}\, V_{N,2N-1} \tag{32.1-11b}$$

Thereby with the appearance of some new computer that can multiply two length $2 \cdot N$ numbers[1] one only needs to combine the two ratios $r_{0,N-1}$ and $r_{N,2N-1}$ that had been precomputed by the last generation of computers. This costs only a few full-size multiplications on your new and expensive supercomputer (instead of several *hundreds* for the iterative schemes), which means that one can improve on prior computations at low cost.

If one wants to stare at zillions of decimal digits of the floating point expansion then one division is also needed which costs no more than 4 multiplications as described in section 28.1 on page 541.

32.1.5 Computation of π: binary splitting versus AGM-type iterations

Using formula 32.1-1a on page 611 and the binary splitting scheme for the computation of π can outperform the AGM-style iterations given in section 30.4 on page 582. The reason is that the memory access pattern is more favorable than with the iterations. When computing N digits of π the iterations compute proportional $\log_2(N)$ roots (and or inverses) to full precision. At the last phase of each root computation full-length multiplications have to be computed. These access all memory storing a few full-precision words. In contrast, the binary splitting involves full-precision multiplications only at the very last phase.

[1] assuming the old model could multiply length-N numbers.

The drawback of the binary splitting scheme is that it may need significantly more memory than two full words. This may happen if the numerator and denominator grow fast which is more likely if no series so favorable as 32.1-1a can be used for the quantity to be computed.

32.1.6 Fast radix conversion

A binary splitting scheme for radix conversion of a radix-z integer $[a_N a_{N-1} \dots a_2 a_1 a_0]_z$ can be obtained via recursive application of the scheme

$$\sum_{k=M}^{N} a_k\, z^k \quad = \quad \sum_{k=M}^{M+X-1} a_k\, z^k + z^X \sum_{k=M+X}^{N} a_k\, z^k \tag{32.1-12}$$

where X is chosen to be the largest power of two that is smaller than $d := N - M$.

```
        R(0, 15)
                R(0, 7)
                        R(0, 3)
                                R(0, 1)
                                ^== 67
                                R(2, 3)
                                ^== 101
                        ^== 25923
                        R(4, 7)
                                R(4, 5)
                                ^== 7
                                R(6, 7)
                                ^== 33
                        ^== 8455
                ^== 554132803
                R(8, 15)
                        R(8, 11)
                                R(8, 9)
                                ^== 67
                                R(10, 11)
                                ^== 101
                        ^== 25923
                        R(12, 15)
                                R(12, 13)
                                ^== 7
                                R(14, 15)
                                ^== 33
                        ^== 8455
                ^== 554132803
        ^== 2379982267079943491
```

Figure 32.1-D: Intermediate results when converting the number 2107654321076543_{16} to decimal.

We define an auxiliary function that computes (for $d > 1$) the largest exponent s so that $2^s < d$:

```
ex2le(d)=
{ /* return largest s so that 2^s < d */
    local(s, t);
    t=1;  s=0;
    while ( d>t,  t<<=1; s+=1; );
    t >>= 1;  s--;
    return(s);
}
```

We precompute z^2, z^4, z^8, …, z^{2^w} where $2^w < N$:

```
N=15;
z=16; \\ radix
vz=vector(ceil(log(N)/log(2)));
vz[1]=z;
for (k=2, length(vz), vz[k]=vz[k-1]^2);  \\ N space
```

Now the conversion function can be defined as

```
Ri(m, n, i=0)=
{ /* Radix conversion, self documenting */
```

```
    local(x, d, ret, t);
    indent(i);  print( "R(", m, ", ", n, ")");
    d = n-m;
    if ( d <= 1, /* then: */
        if ( d==0, ret = A(m); , ret = A(m) + z*A(n); );
    , /* else: */
        t = ex2le(d);
        x = 1<<t;
        ret = Ri(m, m+x-1, i+1) +  vz[t+1] * Ri(m+x, n, i+1);
    );
    indent(i);  print( "^== ", ret);
    return( ret );
}
```

We define `A(k)=(k+3)%8;` and convert the 16-digit, radix-16 number $[a_{15}a_{14}\ldots a_2a_1a_0]_{16} := 2107654321076543_{16}$ to decimal. The intermediate results are shown in figure 32.1-D.

32.2 Rectangular schemes for evaluation of power series

The *rectangular scheme* for the evaluation of polynomials was given in [190], and later in [215]. We use it for the evaluation of truncated power series up to a given power $N-1$ of the series variable. We give two variants, one for series whose coefficients are small rationals (as for the logarithm), and another for series where the ratios of successive coefficients are a small rationals (as for the exponential function). When the numbers of rows and columns in the schemes are identical, a method involving proportional $\sqrt{N}$ full-precision multiplications is obtained. The schemes are very competitive up to very high precision in practice, even compared with AGM-based methods.

32.2.1 Rectangular scheme for arctan and logarithm

Computing the sum of the first N terms of a power series as

$$S_N := \sum_{k=0}^{N-1} A_k\, z^k = A_0 + z\,(A_1 + z\,(A_2 + z\,(A_3 + \ldots z\,(A_{N-1})\ldots))) \tag{32.2-1}$$

costs N long (full-precision) multiplications if z is a full-precision number. If the A_k are small rational values, and $N = R \cdot C$ then one can rewrite S_N as

$$\begin{aligned} S_N = {} & A_{0C} + A_{0C+1}\,z + A_{0C+2}\,z^2 + \ldots + A_{1C-1}\,z^{C-1} + \\ & +z^C\,[A_{1C} + A_{1C+1}\,z + A_{1C+2}\,z^2 + \ldots + A_{2C-1}\,z^{C-1} + \\ & +z^C\,[A_{2C} + A_{2C+1}\,z + A_{2C+2}\,z^2 + \ldots + A_{3C-1}\,z^{C-1} + \\ & +z^C\,[A_{3C} + A_{3C+1}\,z + A_{3C+2}\,z^2 + \ldots + A_{4C-1}\,z^{C-1} + \\ & + \quad \ldots \quad + \\ & +z^C\,[A_{(R-1)C} + A_{(R-1)C+1}\,z + A_{(R-1)C+2}\,z^2 + \ldots + A_{RC-1}\,z^{C-1}\ ]\ldots]]] \end{aligned} \tag{32.2-2}$$

We compute S_N as

$$\left[\left[\left[\ldots[U_{R-1}]\ z^C + \ldots + U_3\right]\ z^C + U_2\right]\ z^C + U_1\right]\ z^C + U_0 \tag{32.2-3}$$

where $U_r := \sum_{k=0}^{C-1} A_{rC+k}\, z^k$ is the sum in one row of relation 32.2-2.

Precomputing the quantities z^2, z^3, z^4,$\ldots$ z^C involves $C-1$ long multiplications. The sums in each row of expression 32.2-2 involve only short multiplications with series coefficients A_i. The multiplication by z^C for each but the first row involves further $R-1$ long multiplications. The computation uses C temporaries (z, z^2, $\ldots$, z^C) and proportional $R+C$ long multiplications. Choosing $R = C = \sqrt{N}$ leads to a complexity of $2\sqrt{N}$ long multiplications, and also involves $\sqrt{N}$ temporaries.

32.2.1.1 Implementation for arctan

We implement the scheme for the arctan in pari/gp:

```
fa(n) =  \\ inverse of series coefficient
{  /* fa(n) := (-1)^n/(2n+1) */
    local(an);
    an = (2*n+1);
    if ( bitand(n,1), an=-an);
    return( an );
}

atan_rect(z, R, C)=
{  /* compute atan(z) as z*(1-z^2/3+z^4/5-z^6/7+-... +-z^(2*(R*C-1))/(2*R*C-1) */
    local(S, vz, s, ur, k);
    vz = vector(C);  \\ vz == [z^2,z^4,z^6,...,z^(2*C)]
    vz[1] = z*z;  \\ 1 long multiplication (special for arctan)
    for (k=2, C, vz[k]=vz[1]*vz[k-1]);  \\ C-1 long multiplications
    k = R*C;  \\ index of current coefficient
    s = 0;  \\ sum
    forstep (r=R-1, 0, -1,
        ur = 0;  \\ sum of this row
        forstep (c=C-1, 1, -1,  k-=1;  ur+=vz[c]/fa(k); );
        k -= 1;  ur += 1/fa(k);
        if ( r!=R-1, s*=vz[C]; );  \\ R-1 long multiplications
        s += ur;
    );
    s *= z;  \\ 1 long multiplication (special for arctan)
    return( s );
}
```

We compute $\pi/16$ as $\arctan(z)$ where $z = \sqrt{2\sqrt{2}+4} - \sqrt{2} - 1 \approx 0.19891236$ (using relation 31.1-30 on page 602 twice on $z = 1$), using a precision of 30,000 decimal digits. We use $R = C = \sqrt{N} =: S$:

```
? ? z=1;z=z+sqrt(z^2+1);z=z+sqrt(z^2+1);z=1/z; \\ ==> Pi/16
? a=atan(z);  \\ builtin arctan:  computed in 1,123 ms.
? r=atan_rect(z,S,S);  \\ computed in 2,377 ms.
  \\ using S=147, and N=S^2=21609
? a-r
0.E-30017  \\ result OK
```

The given implementation involves about two times of the cost of the builtin routine. Argument reduction make the method much more competitive:

```
? a=atan(z); \\ computed in 1,123 ms.
? z=1/z;
? for(k=1,32,z=z+sqrt(z^2+1)) \\ computed in 204 ms.
? z=1/z
4.57161899770987400328861548736 E-11
? S=ceil(sqrt(-1/2*rp*log(10)/log(z)))
39 \\ N=S^2=1521
? r=atan_rect(z,S,S); \\ computed in 284 ms.
? r*=2^32
? a-r
-1.3690050398194919519 E-30016  \\ OK
```

With 100,000 decimal digits the performance ratio is roughly the same. Note that one will have to limit the number C of temporaries according to the available memory.

Compute the inverse sine and cosine as $\arcsin(z) = \arctan \frac{z}{\sqrt{1-z^2}}$ and $\arccos(z) = \frac{\pi}{2} - \arcsin(z)$.

32.2.1.2 Implementation for the logarithm

A routine for $log(1-z)$ is

```
log_rect(z, R, C)=
{ /* compute log(1-z) as 1+x/2+x^2/3+...+x^(R*C-1)/(R*C) */
    local(S, vz, s, ur, k);
    vz = vector(C);  \\ vz == [z^2,z^4,z^6,...,z^(2*C)]
    vz[1] = z;
    for (k=2, C, vz[k]=z*vz[k-1]);  \\ C-1 long multiplications
    k = R*C;  \\ index of current coefficient
```

```
    s = 0;  \\ sum
    forstep (r=R-1, 0, -1,
        ur = 0;  \\ sum of this row
        forstep (c=C-1, 1, -1,  k-=1;  ur+=vz[c]/(k+1); );
        k -= 1;  ur += 1/(k+1);
        if ( r!=R-1, s*=vz[C]; );  \\ R-1 long multiplications
        s += ur;
    );
    s *= z;  \\ 1 long multiplication (special for arctan)
    return( -s );
}
```

However, using a precision of 30,000 decimal digits and argument $z = 1/5$ the routine is slower than the builtin one (using the AGM) by a factor of about 1/7. With argument reduction (relation 31.1-27 on page 601), and $R = C = \sqrt{N} =: S$ we obtain a more competitive performance:

```
? e=log(1-z) \\ computed in 621 ms.
-0.22314355131420975576629509031O
? z=1-z;
? for(k=1,32,z=sqrt(z));  \\ computed in 132 ms.
? z=1-z;  \\ == 5.19546566783481003872552738341 E-11
? S=ceil(sqrt(-rp*log(10)/log(z)))
55  \\ N=S^2=3025
? r=log_rect(z,N);  \\ computed in 461 ms.
? r*=2^32
-0.22314355131420975576629509031O
? e-r
-6.071613129762050924 E-30008  \\ OK
```

We note that with both the logarithm and the arctan, subsequent computations with the builtin routine are faster as some constants that are computed with the first call are reused.

Compute the inverse hyperbolic sine and cosine as $\operatorname{arcsinh}(z) = \log(z + \sqrt{z^2+1})$ and $\operatorname{arccosh}(z) = \log(z + \sqrt{z^2-1})$ (for $z \geq 0$).

32.2.2 Rectangular scheme for exp, sine, and cosine

We rewrite the sum of the first N terms of a power series

$$S_N \quad := \quad \sum_{k=0}^{N-1} A_k \, z^k \tag{32.2-4}$$

as

$$\begin{aligned} S_N \quad = \quad & 1 \left[A_{0C+0} + A_{1C+0}\, z^{1C} + A_{2C+0}\, z^{2C} + \ldots + A_{(R-1)C+0} z^{(R-1)C} \right] + \\ & z^1 \left[A_{0C+1} + A_{1C+1}\, z^{1C} + A_{2C+1}\, z^{2C} + \ldots + A_{(R-1)C+1} z^{(R-1)C} \right] + \\ & z^2 \left[A_{0C+2} + A_{1C+2}\, z^{1C} + A_{2C+2}\, z^{2C} + \ldots + A_{(R-1)C+2} z^{(R-1)C} \right] + \\ & z^3 \left[A_{0C+3} + A_{1C+3}\, z^{1C} + A_{2C+3}\, z^{2C} + \ldots + A_{(R-1)C+3} z^{(R-1)C} \right] + \\ & + \quad \ldots \quad + \\ & z^{C-2} \left[A_{1C-2} + A_{2C-2}\, z^{1C} + A_{3C-2}\, z^{2C} + \ldots + A_{RC-2} z^{(R-1)C} \right] + \\ & z^{C-1} \left[A_{1C-1} + A_{2C-1}\, z^{1C} + A_{3C-1}\, z^{2C} + \ldots + A_{RC-1} z^{(R-1)C} \right] \end{aligned} \tag{32.2-5}$$

Compute the sum as (the transposed version of relation 32.2-2 on page 617)

$$S_N \quad = \quad [[[\ldots[[U_{C-1}]\, z + U_{C-2}]\, z + \ldots + U_3]\, z + U_2]\, z + U_1]\, z + U_0 \tag{32.2-6}$$

where $U_c = \sum_{k=0}^{R-1} A_{kC+c}\, z^{kC}$ (C temporary sums are computed). When proceeding colum-wise the update $A_i \to A_{i+1}$ involves only a short multiplication by the ratio A_{i+1}/A_i. Only when going to the next column a long multiplication by z^C is required ($R-1$ long multiplications). Finally, there are $C-1$ long multiplications by z.

32.2.2.1 Implementation for the exponential function

A routine for the computation of $\exp(z) - 1$ can be given as follows:

```
exp_rect(z, R, C)=
{ /* compute exp(z)-1 as z*[ 1+z/2!+z^2/3! +...+z^(R*C-1)/((R*C)!) ] */
    local(ur, zc, k, t);
    zc = z^C; \\ proportional log(C) long multiplications
    ur = vector(C);
    k = 1;  \\ ratio of series coefficients  /* set to zero for plain exp */
    t = 1.0;
    for (r=1, R,  \\ number of columns (!)
        for (c=1, C,  ur[c] += t;  k++;  t /= (k); );
        if ( r!=R,  t *= zc; );  \\ R-1 long multiplications
    );
    t = ur[C];
    forstep (c=C-1, 1, -1,  t*=z; t+=ur[c]);  \\ C-1 long multiplications
    t *= z;  /* omit for plain exp */
    return( t );
}
```

We use the argument reduction given as relation 31.2-18 on page 605 and compute $exp(1/5)$ to a precision of 30,000 decimal digits. We use $R = C = \sqrt{N} =: S$:

```
? z=0.2;
? e=exp(z)  \\ computed in 855 ms.
1.221402758160169833921071994 64
? nred=32;
? z/=2^nred
4.656612873077392578125000000000 E-11
? S=48; \\ N=S^2=2304
? r=exp_rect(z,S,S)  \\ computed in 395 ms.
4.656612873185812795375233346 67 E-11
? for(k=1,nred,r=r+r+r^2);  \\ computed in 68 ms.
? r+=1
1.221402758160169833921071994 64
? e-r
7.965120231677044083 E-30016  \\ OK
```

Using 100,000 digits, `nred=112`, and `S=52` we obtain the timings

```
? e=exp(z);                \\ computed in 8,601 ms.
? r=exp_rect(z,S,S);       \\ computed in 2,345 ms.
? for(k=1,nred,r=r+r+r^2); \\ computed in 1,640 ms.
```

32.2.2.2 Implementation for the cosine

A routine for computing $\cos(z) - 1$ can be given as

```
cos_rect(z, R, C)=
{ /* compute cos(z)-1 as z^2*[ -1/2!+z^2/4! - z^4/6! +- ... ] */
    local(ur, zc, k, t);
    z *= z;
    zc = z^C; \\ proportional log(C) long multiplications
    ur = vector(C);
    k = 2;  \\ ratio of series coefficients
    t = -0.5;
    for (r=1, R,  \\ number of columns (!)
        for (c=1, C,  ur[c] += t;  k++;  t /= (k); k++;  t /= -(k); );
        if ( r!=R,  t *= zc; );  \\ R-1 long multiplications
    );
    t = ur[C];
    forstep (c=C-1, 1, -1,  t*=z; t+=ur[c]);  \\ C-1 long multiplications
    t *= z;  /* omit for plain exp */
    return( t );
}
```

We use the argument reduction as in relation 31.2-20 on page 605, and compute $\cos(1/5)$ to 30,000 decimal digits:

```
? z=0.2;
? e=cos(z)  \\ computed in 788 ms.
0.980066577841241631124196516748
? nred=32;
```

```
? z/=2^nred;
? S=34;  \\ N=S^2=1156
? r=cos_rect(z,S,S);  \\ computed in 318 ms.
? for(k=1,nred,r=2*(r+1)^2-2);  \\ computed in 70 ms.
? r+=1
0.980066577841241631124196516748
? e-r
-3.646143951667310362 E-30017  \\ OK
```

The sine and tangent can be computed as $\sin(z) = \sqrt{1-\cos(z)^2}$, and $\tan(z) = \sin(z)/\cos(z)$.

The routine is easily converted to compute the hyperbolic cosine. The relation $\exp(z) = \cosh(z) - \sqrt{\cosh(z)^2 - 1}$ gives an alternative way to compute the exponential function.

32.3 The magic sumalt algorithm for alternating series

The following convergence acceleration algorithm for alternating series is due to Cohen, Villegas and Zagier, see [85]. As remarked in the cited paper, the algorithm often gives meaningful results also for non-alternating and even divergent series.

The algorithm computes an estimate of the sum $s = \sum_{k=0}^{\infty} x_k$ as

$$s_n \quad = \quad \sum_{k=0}^{n-1} c_{n,k}\, x_k \tag{32.3-1}$$

The weights $c_{n,k}$ do not depend on the values x_j. With the following pseudo code the summands x_k have to be supplied in the array `x[0,1,...,n-1]`:

```
function sumalt(x[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        c := c - b
        s := s + c * x[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }
    return s/d
}
```

With alternating sums the accuracy of the estimate will be $(3+\sqrt{8})^{-n} \approx 5.82^{-n}$. For example, the estimate for $4 \cdot \arctan(1)$ using the first 8 terms is

$$\pi \quad \approx \quad 4 \cdot \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \right) \; = \; 3.017\ldots \tag{32.3-2}$$

The sumalt-massaged estimate is

$$\begin{aligned} \pi \quad &\approx \quad 4 \cdot \left(\frac{665856}{1} - \frac{665728}{3} + \frac{663040}{5} - \frac{641536}{7} + \right. \\ &\qquad \left. + \frac{557056}{9} - \frac{376832}{11} + \frac{163840}{13} - \frac{32768}{15} \right) /665857 \\ &= \quad 4 \cdot 3365266048/4284789795 = 3.141592665\ldots \end{aligned} \tag{32.3-3}$$

and already gives 7 correct digits of π. The linear but impressive growth of the accuracy of successive sumalt estimates with n, the number of terms used, is illustrated in figure 32.3-A.

n:	sumalt(n)	sumalt(n)$-\pi$
1:	2.66666666666666666666	0.474925986923126571795
2:	3.13725490196078431372	0.004337751629008924737
3:	3.14074074074074074074	0.000851912849052497721
4:	3.14163571841214822150 7	−0.000043064822354983044
5:	3.14158654640367396834 8	0.000006107186119270114
6:	3.14159344421566940366 0	−0.000000690625866165197
7:	3.14159256493754012201 5	0.000000088652253116447
8:	3.14159266522431586401 7	−0.000000011634522625555
9:	3.14159265200881195161 9	0.000000001580981286843
10:	3.14159265380973156931 8	−0.000000000219938330856
11:	3.14159265355857875551 3	0.000000000031214482948
12:	3.14159265359429633847 0	−0.000000000004503100007
13:	3.14159265358913458051 7	0.000000000000658657944
14:	3.14159265358989071862 5	−0.000000000000097480163
15:	3.14159265358977866437 5	0.000000000000014574087
16:	3.14159265358979543677 5	−0.000000000000002198312
17:	3.14159265358979290428 5	0.000000000000000334177
18:	3.14159265358979328961 4	−0.000000000000000051151
19:	3.14159265358979323058 4	0.000000000000000007877
20:	3.14159265358979323968 2	−0.000000000000000001220

Figure 32.3-A: Sumalt-estimates of $\pi = 4 \cdot \arctan(1)$ using $n = 1, 2, \ldots, 20$ terms.

Therefore even slowly converging series like

$$\pi = 4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2\,k+1} = 4 \cdot \arctan(1) \tag{32.3-4a}$$

$$C = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2\,k+1)^2} = 0.9159655941772190\ldots \tag{32.3-4b}$$

$$\log(2) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k+1} = 0.6931471805599453\ldots \tag{32.3-4c}$$

$$\zeta(s) = \frac{1}{1-2^{1-s}} \sum_{k=1}^{\infty} \frac{(-1)^k}{k^s} \tag{32.3-4d}$$

can be used to compute estimates that are correct up to thousands of digits. The algorithm scales like n^2 if the series terms in the array `x[]` are small rational values and like $n^3 \cdot \log(n)$ if they are full precision (rational or float) values.

In fact, pari/gp has a built-in sumalt routine, we use it to compute the *Catalan constant*:

```
? default(realprecision,1000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes    60 ms.
? default(realprecision,2000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes   376 ms.
? default(realprecision,4000);
? sumalt(k=0,(-1)^k/(2*k+1)^2);  \\ takes 2,730 ms.
```

The time scales roughly with the third power of the precision used.

All values c_k and b_k occurring in the computation are integers. In fact, the b_k in the computation with n terms are the coefficients of the expanded n-th Chebyshev polynomial (of the first kind, see section 34.2 on page 645) with argument $1 + 2x$:

k :	b_k	c_k
0:	1	665857
1:	128	665856
2:	2688	665728
3:	21504	663040
4:	84480	641536
5:	180224	557056
6:	212992	376832
7:	131072	163840
8:	32768	32768

$$\begin{aligned} T_8(1+2x) &= 1+128x+2688x^2+21504x^3+84480x^4+ \qquad (32.3\text{-}5a)\\ &\quad +180224x^5+212992x^6+131072x^7+32768x^8 = T_{16}(\sqrt{1+x}) \\ T_{16}(x) &= 1-128x^2+2688x^4-21504x^6+84480x^8- \qquad (32.3\text{-}5b)\\ &\quad -180224x^{10}+212992x^{12}-131072x^{14}+32768x^{16} \end{aligned}$$

Now observe that one has always $c_n = b_n = 2^{2n-1}$ in a length-n sumalt computation. Obviously, 'going backwards' avoids the computation of $(3+\sqrt{8})^n$:

```
function sumalt(x[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
    {
        s := s + c * x[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

The b_k and c_k occurring in a length-n sumalt computation can be given explicitly as

$$b_k = \frac{n}{n+k}\binom{n+k}{2k}2^{2k} \qquad (32.3\text{-}6a)$$

$$c_k = \sum_{i=k}^{n}\frac{n}{n+i}\binom{n+i}{2i}2^{2i} \qquad (32.3\text{-}6b)$$

To compute an estimate of $\sum_{k=0}^{\infty} x_k$ using the first n partial sums use the following pseudo code (the partial sums $p_k = \sum_{j=0}^{k} x_j$ are expected in `p[0,1,...,n-1]`):

```
function sumalt_partial(p[], n)
{
    d := (3+sqrt(8))^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1))
    }
    return s/d
}
```

The backward variant is:

```
function sumalt_partial(p[], n)
{
    b := 2**(2*n-1)
    c := b
    s := 0
    for k:=n-1 to 0 step -1
```

```
    {
        s := s + b * p[k]
        b := b * ((2*k+1)*(k+1)) / (2*(n+k)*(n-k))
        c := c + b
    }
    return s/c
}
```

For series of already geometrical rate of convergence (where $|a_k/a_{k+1}| \approx e$) it is better to use

```
function sumalt_partial(p[], n, e)
{
    d := ( 2*e + 1 + 2*sqrt(e*(e+1)) )^n
    d := (d+1/d)/2
    b := 1
    c := d
    s := 0
    for k:=0 to n-1
    {
        s := s + b * p[k]
        b := b * (2*(n+k)*(n-k)) / ((2*k+1)*(k+1)) * e
    }
    return s/d
}
```

Convergence is improved from $\sim e^{-n}$ to $\sim (2e + 1 + 2\sqrt{e(e+1)})^{-n} \approx (4e + 2)^{-n}$. This algorithm specializes to the original one for $e = 1$.

C++ routines implementing the sumalt algorithm and the variant for partial sums are given in [hfloat: src/hf/sumalt.cc].

Chapter 33

Computing the elementary functions with limited resources

This chapter presents two types of algorithms for computations with limited resources, the shift-and-add and the CORDIC algorithms. The algorithms allow the computations of the elementary functions as the logarithm, exponential function, sine, cosine and their inverses with only shifts, adds, comparisons and table lookups. Algorithms of this type are usually used for pocket calculators.

33.1 Shift-and-add algorithms for $\log_b(x)$ and b^x

In this section so-called *shift-and-add algorithms* for the computation of $\log_b(x)$ and b^x are presented. These algorithms use only additions, multiplications by a power of two ('shifts') and comparisons. Precomputed lookup table with as many entries as the desired accuracy in bits is required. The algorithms are especially useful with limited hardware capabilities.

The implementations given in this section use floating point numbers. They can be rewritten to scaled integer arithmetic without difficultly.

33.1.1 Computing the base-b logarithm

We use a table that contains the values $A_n = \log_b\left(1 + \frac{1}{2^n}\right)$ where $n > 0$, it is created as follows:

```
double *shiftadd_ltab;
ulong ltab_n;

void
make_shiftadd_ltab(double b)
{
    double l1b = 1.0 / log(b);
    double s = 1.0;
    for (ulong k=0; k<ltab_n; ++k)
    {
        shiftadd_ltab[k] = log(1.0+s) * l1b;  // == log_b(1+1/2^k)
        s *= 0.5;
    }
}
```

The algorithm takes as input the argument $x \geq 1$ and the number of iterations N and computes $\log_b(x)$. It proceeds as follows:

1. Initialize: set $t_0 = 0$, $e_0 = 1$.
2. Compute $u_n = e_n \cdot (1 + 2^{-n})$. If $u_n \leq x$ the set $d_n = 1$, else set $d_n = 0$.

n :	u_n	t_n	e_n	A_n
init	-	0.00000000	+1.00000000	+1.00000000
1:	1.50000000	0.00000000	+1.00000000	+0.58496250
2:	1.25000000	0.00000000	+1.00000000	+0.32192809
2:	1.56250000	0.32192809	+1.25000000	+0.32192809
3:	1.40625000	0.32192809	+1.25000000	+0.16992500
3:	1.58203125	0.49185309	+1.40625000	+0.16992500
4:	1.49414062	0.49185309	+1.40625000	+0.08746284
5:	1.45019531	0.49185309	+1.40625000	+0.04439411
6:	1.42822265	0.49185309	+1.40625000	+0.02236781
7:	1.41723632	0.49185309	+1.40625000	+0.01122725
8:	1.41174316	0.49185309	+1.40625000	+0.00562454
8:	1.41725778	0.49747764	+1.41174316	+0.00562454
9:	1.41450047	0.49747764	+1.41174316	+0.00281501
10:	1.41312181	0.49747764	+1.41174316	+0.00140819
10:	1.41450182	0.49888583	+1.41312181	+0.00140819
11:	1.41381182	0.49888583	+1.41312181	+0.00070426
11:	1.41450215	0.49959010	+1.41381182	+0.00070426
12:	1.41415698	0.49959010	+1.41381182	+0.00035217
12:	1.41450224	0.49994228	+1.41415698	+0.00035217
13:	1.41432961	0.49994228	+1.41415698	+0.00017609
14:	1.41424330	0.49994228	+1.41415698	+0.00008805
15:	1.41420014	0.49994228	+1.41415698	+0.00004402
15:	1.41424330	0.49998631	+1.41420014	+0.00004402
∞:	1.41421356	0.50000000	+1.41421356	+0.00000000
	$= x$	$= \log_2(\sqrt{2})$	$= x$	$= 0$

Figure 33.1-A: Numerical values occurring in the shift-and-add computation of $\log_2(\sqrt{2}) = 1/2$. The computation of $\log_{1/2}(\sqrt{2}) = -1/2$ corresponds to the same values but opposite signs for all entries A_n and y_n.

3. If $d_n \neq 0$ then set $t_{n+1} = t_n + A_n$ and $e_{n+1} = u_n$ and repeat the last step. Else set $t_{n+1} = t_n$ and $e_{n+1} = e_n$.

4. Increment n. If $n = N$ return t_n, else goto step 2.

A C++ implementation is given in [FXT: arith/shiftadd-log-demo.cc], note that the variable `n` equals N, and `k` equals n:

```
double
shiftadd_log(double x, ulong n)
{
    if ( n>=ltab_n )  n = ltab_n;
    double t = 0.0;
    double e = 1.0;
    double v = 1.0;
    // [PRINT]
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;  // v == (1>>k)
        double u;
        bool d;
        while ( 1 )
        {
            u = e + e * v;  // u=e;  u+=(e>>k);
            d = ( u<=x );
            // [PRINT]
            if ( d==false )  break;
            t += shiftadd_ltab[k];
            e = u;
        }
    }
    return  t;
}
```

The variable `v` is a power of 1/2 so all multiplies by it can with scaled integer arithmetic be replaced by shifts as indicated by the comments. The values for first steps of the computation for the argument $x_0 = \sqrt{2}$ are given in figure 33.1-A. The columns of the figure correspond to the variables `k`($= n$), `u`($= u_n$), `t`($= t_n$), `e`($= e_n$), and `shiftadd_ltab[k]`($= A_n$).

n :	u_n	t_n	e_n	A_n
init	-	0.00000000	+1.00000000	+1.00000000
1:	1.50000000	0.00000000	+1.00000000	+0.58496250
1:	2.25000000	0.58496250	+1.50000000	+0.58496250
1:	3.37500000	1.16992500	+2.25000000	+0.58496250
1:	5.06250000	1.75488750	+3.37500000	+0.58496250
1:	7.59375000	2.33985000	+5.06250000	+0.58496250
1:	11.3906250	2.92481250	+7.59375000	+0.58496250
2:	9.49218750	2.92481250	+7.59375000	+0.32192809
3:	8.54296875	2.92481250	+7.59375000	+0.16992500
4:	8.06835937	2.92481250	+7.59375000	+0.08746284
5:	7.83105468	2.92481250	+7.59375000	+0.04439411
5:	8.07577514	2.96920662	+7.83105468	+0.04439411
6:	7.95341491	2.96920662	+7.83105468	+0.02236781
6:	8.07768702	2.99157443	+7.95341491	+0.02236781
∞:	8.00000000	2.99999999	+8.00000000	+0.00000000
	$= x$	$= \log_2(8)$	$= x$	$= 0$

Figure 33.1-B: Values occurring in the first few steps of a shift-and-add computation of $\log_2(8) = 3$.

The algorithm has been adapted from [184] (chapter 5) where the correction is made only once for each value A_n limiting the range of convergence to $x < X$ where

$$\begin{aligned} X &= \prod_{n=0}^{\infty} \left(1 + \frac{1}{2^n}\right) \\ &= 4.76846205806274344829979857735679447754 3\ldots \end{aligned} \tag{33.1-1}$$

As given, the algorithm converges for any $x > 0$, $x \neq 1$. A numerical example for the argument $x = 8$ is given in figure 33.1-B. The basis b must satisfy $b > 0$ and $b \neq 1$.

33.1.2 Computing b^x

We can use the same precomputed table as with the computation of $\log_b(x)$.

The algorithm takes as input the argument x and the number of iterations N and computes b^x for $b > 1$, $x \in \mathbb{R}$. It proceeds as follows:

1. Initialize: set $t_0 = 0$, $e_0 = 1$.
2. Compute $u_n = t_n + A_n$. If $u_n \leq x$ the set $d_n = 1$, else set $d_n = 0$.
3. If $d_n \neq 0$ then set $t_{n+1} = u_n$ and $e_{n+1} = e_n \cdot (1 + 2^{-n})$ and repeat the last step. Else set $t_{n+1} = t_n$ and $e_{n+1} = e_n$.
4. Increment n. If $n = N$ return e_n, else goto step 2.

A C++ implementation is given in [FXT: arith/shiftadd-exp-demo.cc]:

```
double
shiftadd_exp(double x, ulong n)
{
    if ( n>=ltab_n )  n = ltab_n;
    double t = 0.0;
    double e = 1.0;
    double v = 1.0;
    // [PRINT]
```

n :	u_n	t_n	e_n	A_n
init	0.00000000	0.00000000	+1.00000000	+0.00000000
1:	0.58496250	0.00000000	+1.00000000	+0.58496250
2:	0.32192809	0.00000000	+1.00000000	+0.32192809
2:	0.64385618	0.32192809	+1.25000000	+0.32192809
3:	0.49185309	0.32192809	+1.25000000	+0.16992500
3:	0.66177809	0.49185309	+1.40625000	+0.16992500
4:	0.57931593	0.49185309	+1.40625000	+0.08746284
5:	0.53624721	0.49185309	+1.40625000	+0.04439411
6:	0.51422090	0.49185309	+1.40625000	+0.02236781
7:	0.50308035	0.49185309	+1.40625000	+0.01122725
8:	0.49747764	0.49185309	+1.40625000	+0.00562454
8:	0.50310219	0.49747764	+1.41174316	+0.00562454
9:	0.50029266	0.49747764	+1.41174316	+0.00281501
10:	0.49888583	0.49747764	+1.41174316	+0.00140819
10:	0.50029403	0.49888583	+1.41312181	+0.00140819
11:	0.49959010	0.49888583	+1.41312181	+0.00070426
11:	0.50029437	0.49959010	+1.41381182	+0.00070426
12:	0.49994228	0.49959010	+1.41381182	+0.00035217
12:	0.50029446	0.49994228	+1.41415698	+0.00035217
13:	0.50011838	0.49994228	+1.41415698	+0.00017609
14:	0.50003033	0.49994228	+1.41415698	+0.00008805
15:	0.49998631	0.49994228	+1.41415698	+0.00004402
15:	0.50003034	0.49998631	+1.41420014	+0.00004402
∞:	0.50000000	0.50000000	+1.41421356	+0.00000000
	$= x$	$= x$	$= 2^{1/2}$	$= 0$

Figure 33.1-C: Numerical values occurring in the shift-and-add computation of $b^x = 2^{1/2} = \sqrt{2}$. The values are printed at points where a comment `[PRINT]` appears in the code.

```
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;  // v == (1>>k)
        double u;
        bool d;
        while ( 1 )
        {
            u = t + shiftadd_ltab[k];
            d = ( u<=x );
            // [PRINT]
            if ( d==false )  break;
            t = u;
            e += e * v;  // e+=(e>>k);
        }
    }
    return  e;
}
```

33.1.3 An alternative algorithm for the logarithm

A slightly different method for the computation of the base-b logarithm ($b > 0$, $b \neq 1$) is given in [154]. Here the table used has to contain the values $A_n = \log_b\left(\frac{2^n}{2^n-1}\right)$ where $n > 0$:

```
double *briggs_ltab;
ulong  ltab_len;

void
make_briggs_ltab(ulong na, double b)
{
    double l1b = 1.0 / log(b);
    double s = 2.0; // == 2^k
    briggs_ltab[0] = -1.0; // unsused
    for (ulong k=1; k<na; ++k)
```

n :	x_n	y_n	z_n	A_n
init	1.41421356	0.00000000	+0.70710678	+0.00000000
2:	1.41421356	0.00000000	+0.35355339	+0.41503749
2:	1.06066017	0.41503749	+0.26516504	+0.41503749
3:	1.06066017	0.41503749	+0.13258252	+0.19264507
4:	1.06066017	0.41503749	+0.06629126	+0.09310940
5:	1.06066017	0.41503749	+0.03314563	+0.04580368
5:	1.02751454	0.46084118	+0.03210982	+0.04580368
6:	1.02751454	0.46084118	+0.01605491	+0.02272007
6:	1.01145962	0.48356126	+0.01580405	+0.02272007
7:	1.01145962	0.48356126	+0.00790202	+0.01131531
7:	1.00355759	0.49487657	+0.00784029	+0.01131531
8:	1.00355759	0.49487657	+0.00392014	+0.00564656
9:	1.00355759	0.49487657	+0.00196007	+0.00282051
9:	1.00159752	0.49769709	+0.00195624	+0.00282051
10:	1.00159752	0.49769709	+0.00097812	+0.00140957
10:	1.00061940	0.49910666	+0.00097716	+0.00140957
11:	1.00061940	0.49910666	+0.00048858	+0.00070461
11:	1.00013081	0.49981128	+0.00048834	+0.00070461
12:	1.00013081	0.49981128	+0.00024417	+0.00035226
13:	1.00013081	0.49981128	+0.00012208	+0.00017612
13:	1.00000873	0.49998740	+0.00012207	+0.00017612
∞:	1.00000000	0.50000000	+0.00000000	+0.00000000
	$= 1$	$= \log_2(\sqrt{2})$	$= 0$	$= 0$

Figure 33.1-D: Numerical values occurring in the computation of $\log_2(\sqrt{2}) = 1/2$. The value of n is incremented in the inner loop (comment `[PRINT1]` in the code, the value of z changes). The values of x and y change just before the location of the comment `[PRINT2]`, corresponding to consecutive rows with same value of n. The computation of $\log_{1/2}(\sqrt{2}) = -1/2$ corresponds to the same values but opposite signs for all entries A_n and y_n.

```
    {
        briggs_ltab[k] = log(s/(s-1.0)) * l1b;
        s *= 2.0;
    }
}
```

The algorithm terminates when a given precision (`eps`) is reached:

```
double
briggs_log(double x, double eps)
{
    double y = 0;
    double z = x * 0.5;
    // [PRINT]
    ulong k = 1;
    double v = 0.5;  // v == 2^(-k)
    while ( fabs(x-1.0)>=eps )
    {
        while ( fabs(x-z)<1.0 )
        {
            z *= 0.5;
            ++k;  v *= 0.5;
            if ( k >= ltab_len )  goto done;  // no more table entries
            // [PRINT1]
        }
        x -= z;
        y += briggs_ltab[k];
        z = x * v;  // z=(x>>k)
        // invariant:  y_k + log_b(x_k) == log_b(x_0)
        // [PRINT2]
    }
 done:
    return  y;
```

```
}
```

The code is given in [FXT: arith/briggs-log-demo.cc]. The values for first steps of the computation for the argument $x_0 = \sqrt{2}$ are given in figure 33.1-D. The argument x must be greater than or equal to 1. Knuth [154] gives $1 \le x < 2$ but the restriction to values smaller than 2 does not seem to be necessary.

33.2 CORDIC algorithms

The so-called CORDIC algorithms can be used for the computation of functions like sine, cosine, exp and log. The acronym CORDIC stands for **Co**ordinate **R**otation **Di**gital **C**omputer.

Similar to the shift-and-add algorithms (section 33.1) only multiplications by powers of two (shifts), additions, subtractions and comparisons are used. Again, a precomputed lookup table with as many entries as the desired accuracy in bits is required.

Some early floating point units (FPUs) used CORDIC algorithms and your pocket calculator surely does.

33.2.1 The circular case: sine and cosine

n :	x_n	y_n	z_n	$-d \cdot A_n$
init	0.60725293	0.00000000	+1.04719755	+0.00000000
0:	0.60725293	0.60725293	+0.26179938	-0.78539816
1:	0.30362646	0.91087940	-0.20184822	-0.46364760
2:	0.53134631	0.83497278	+0.04313044	+0.24497866
3:	0.42697471	0.90139107	-0.08122455	-0.12435499
4:	0.48331166	0.87470515	-0.01880574	+0.06241880
5:	0.51064619	0.85960166	+0.01243409	+0.03123983
6:	0.49721492	0.86758051	-0.00318963	-0.01562372
7:	0.50399289	0.86369602	+0.00462270	+0.00781234
8:	0.50061908	0.86566474	+0.00071647	-0.00390623
9:	0.49892833	0.86664251	-0.00123664	-0.00195312
10:	0.49977466	0.86615528	-0.00026008	+0.00097656
11:	0.50019758	0.86591124	+0.00022819	+0.00048828
12:	0.49998618	0.86603336	-0.00001594	-0.00024414
13:	0.50009190	0.86597233	+0.00010612	+0.00012207
14:	0.50003904	0.86600285	+0.00004508	-0.00006103
15:	0.50001261	0.86601811	+0.00001457	-0.00003051
∞:	0.50000000	0.86602540	+0.00000000	+0.00000000
	$= \cos(\pi/3)$	$= \sin(\pi/3)$	$= 0$	$= 0$

Figure 33.2-A: Numerical values occurring in the CORDIC computation of $\cos(\pi/3)$ and $\sin(\pi/3)$.

We start with a CORDIC routine for the computation of the sine and cosine. The lookup table has to contain the values $\arctan(2^{-n})$ for $n = 0, 1, 2, 3, \ldots$, these shall be stored in the array `cordic_ctab[]`. An implementation of the function is given in [FXT: arith/cordic-circ-demo.cc]:

```
void
cordic_circ(double theta, double &s, double &c, ulong n)
{
    double x = cordic_1K;
    double y = 0;
    double z = theta;
    double v = 1.0;
    // [PRINT]
    for (ulong k=0; k<n; ++k)
    {
        double d = ( z>=0 ? +1 : -1 );
        double tx = x - d * v * y;
        double ty = y + d * v * x;
        double tz = z - d * cordic_ctab[k];
```

```
        x = tx;  y = ty;  z = tz;
        v *= 0.5;
        // [PRINT]
    }
    c = x;
    s = y;
}
```

For the sake of clarity floating point types are used. All operations can easily be converted to integer arithmetic. The multiplications by `d` are sign changes and should be replaced by an `if`-construct. The multiplications by `v` are shifts.

The values for first 16 steps of the computation for the argument $z_0 = \theta = \pi/3 = 1.04719755\ldots$ are given in figure 33.2-A. While z gets closer to zero (however, the magnitude of z does not necessarily decrease with every step) the values of x and y approach $\sin(\pi/3) = 1/2$ and $\cos(\pi/3) = \sqrt{3}/2 = 0.86602540\ldots$, respectively.

More formally, one initializes

$$x_0 = 1/K = 0.607252935008881\ldots \tag{33.2-1a}$$
$$y_0 = 0 \tag{33.2-1b}$$
$$z_0 = \theta \tag{33.2-1c}$$

and iterates (starting with $n = 0$)

$$A_n = \arctan\left(2^{-n}\right) \quad \text{(precomputed)} \tag{33.2-1d}$$
$$v_n = 2^{-n} \tag{33.2-1e}$$
$$d_n = \operatorname{sign}(z_n) \tag{33.2-1f}$$
$$x_{n+1} = x_n - d_n\, v_n\, y_n \quad \to \cos(\theta) \tag{33.2-1g}$$
$$y_{n+1} = y_n + d_n\, v_n\, x_n \quad \to \sin(\theta) \tag{33.2-1h}$$
$$z_{n+1} = z_n - d_n\, A_n \quad \to 0 \tag{33.2-1i}$$

The scaling constant K is

$$K = \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2k}} \tag{33.2-2a}$$
$$K = 1.64676025812106564836605122228229843565237672570102740 9\ldots \tag{33.2-2b}$$
$$\frac{1}{K} = 0.60725293500888125616944675250492826311239085215008977 24\ldots \tag{33.2-2c}$$

The algorithm converges if $-r \le z_0 \le r$ where

$$r = \sum_{k=0}^{\infty} \arctan(2^{-k}) \tag{33.2-3a}$$
$$r = 1.74328662047234000350433765613641628581383118542820652 3\ldots \tag{33.2-3b}$$
$$r > \frac{\pi}{2} = 1.57079632\ldots \tag{33.2-3c}$$

With arguments x_0, y_0, z_0 one has

$$x \to K\left(x_0\, \cos(z_0) - y_0\, \sin(z_0)\right) \tag{33.2-4a}$$
$$y \to K\left(y_0\, \cos(z_0) + x_0\, \sin(z_0)\right) \tag{33.2-4b}$$
$$z \to 0 \tag{33.2-4c}$$

which, for $x_0 = 1/K, y_0 = 0, z_0 = \theta$ specializes to the computation as above.

A nice feature of the algorithm is that it also works backwards: initialize as above and use the same iteration with the slight modification that $d_n := -\operatorname{sign}(y_n)$, then

$$x \rightarrow K\sqrt{x_0^2 + y_0^2} \tag{33.2-5a}$$
$$y \rightarrow 0 \tag{33.2-5b}$$
$$z \rightarrow z_0 - \arctan\left(\frac{y_0}{x_0}\right) \tag{33.2-5c}$$

The algorithm can be derived by writing

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} +\cos(d_n\, A_n) & -\sin(d_n\, A_n) \\ +\sin(d_n\, A_n) & +\cos(d_n\, A_n) \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{33.2-6}$$

and noting that (using $d_n = \pm 1$, so $\cos(d_n\, A_n) = \cos(A_n)$ and $\sin(d_n\, A_n) = d_n\, \sin(A_n)$)

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \cos(A_n) \begin{bmatrix} +1 & -d_n\, v_n \\ +d_n\, v_n & +1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{33.2-7}$$

where $v_n = 2^{-n}$. The CORDIC algorithm postpones the multiplications by $\cos(A_n)$. One has

$$\cos(A_n) = \cos\left(\arctan(2^{-n})\right) = \frac{1}{\sqrt{1 + 2^{-2n}}} \tag{33.2-8}$$

Thereby

$$K = 1/\prod_{k=0}^{\infty} \cos(A_n) = \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2n}} \tag{33.2-9}$$

33.2.2 The linear case: multiplication and division

A slight variation gives a base-2 multiply-add algorithm:

$$A_n = 2^{-n} \tag{33.2-10a}$$
$$v_n = 2^{-n} \tag{33.2-10b}$$
$$d_n = \operatorname{sign}(z_n) \tag{33.2-10c}$$
$$x_{n+1} = x_n \tag{33.2-10d}$$
$$y_{n+1} = y_n + d_n\, v_n\, x_n \tag{33.2-10e}$$
$$z_{n+1} = z_n - d_n\, A_n \tag{33.2-10f}$$

then

$$x \rightarrow x_0 \tag{33.2-11a}$$
$$y \rightarrow y_0 + x_0\, z_0 \tag{33.2-11b}$$
$$z \rightarrow 0 \tag{33.2-11c}$$

Going backwards (replace relation 33.2-10c by $d_n := -\operatorname{sign}(y_n)$) gives an algorithm for division:

$$x \rightarrow x_0 \tag{33.2-12a}$$
$$y \rightarrow 0 \tag{33.2-12b}$$
$$z \rightarrow z_0 - \frac{y_0}{x_0} \tag{33.2-12c}$$

n :	x_n	y_n	z_n	A_n
init	1.20749706	0.00000000	+1.00000000	+0.00000000
1:	1.20749706	0.60374853	+0.45069385	-0.54930614
2:	1.35843420	0.90562280	+0.19528104	-0.25541281
3:	1.47163705	1.07542707	+0.06962382	-0.12565721
4:	1.53885124	1.16740439	+0.00704225	-0.06258157
+4:	1.61181401	1.26358259	-0.05553931	-0.06258157
5:	1.57232706	1.21321340	-0.02427913	+0.03126017
6:	1.55337060	1.18864579	-0.00865286	+0.01562627
7:	1.54408430	1.17651008	-0.00084020	+0.00781265
8:	1.53948856	1.17047850	+0.00306606	+0.00390626
9:	1.54177465	1.17348532	+0.00111293	-0.00195312
10:	1.54292063	1.17499096	+0.00013637	-0.00097656
11:	1.54349436	1.17574434	-0.00035190	-0.00048828
12:	1.54320731	1.17536751	-0.00010776	+0.00024414
13:	1.54306383	1.17517913	+0.00001430	+0.00012207
+13:	1.54320729	1.17536749	-0.00010776	-0.00012207
14:	1.54313555	1.17527330	-0.00004673	+0.00006103
15:	1.54309968	1.17522621	-0.00001621	+0.00003051
∞:	1.54308063	1.17520119	+0.00000000	+0.00000000
	$= \cosh(1)$	$= \sinh(1)$	$= 0$	$= 0$

Figure 33.2-B: Numerical values occurring in the CORDIC computation of cosh(1) and sinh(1). Note that steps 4 and 13 are executed twice.

33.2.3 The hyperbolic case: *sinh* and *cosh*

The versions presented so far can be unified as

$$v_n = 2^{-n} \tag{33.2-13a}$$
$$x_{n+1} = x_n - m\, d_n\, v_n\, y_n \tag{33.2-13b}$$
$$y_{n+1} = y_n + d_n\, v_n\, x_n \tag{33.2-13c}$$
$$z_{n+1} = z_n - d_n\, A_n \tag{33.2-13d}$$

where the linear case corresponds to $m = 0$ and $A_n = 2^{-n}$, the circular case to $m = 1$ and $A_n = \arctan(2^{-n})$. The forward direction ('rotation mode') is obtained by setting $d_n = \operatorname{sign}(z_n)$, the backward direction ('vectoring mode') by setting $d_n = -\operatorname{sign}(y_n)$.

Setting $m = -1$ gives a CORDIC algorithm that computes the hyperbolic sine and cosine or their inverses. The lookup table has to contain the values $\operatorname{arctanh}(2^{-n})$ for $n = 1, 2, 3, \ldots$, stored in the array `cordic_htab[]`. The algorithm needs a modification in order to converge: the iteration starts with index one and some steps have to be executed twice. The sequence of the indices that need to be processed twice is $4, 13, 40, 121, \ldots$ ($i_0 = 4$, $i_{n+1} = 3\, i_n + 1$).

A sample implementation is given in [FXT: arith/cordic-hyp-demo.cc]:

```
void
cordic_hyp(double theta, double &s, double &c, ulong n)
{
    double x = cordic_1Kp;
    double y = 0;
    double z = theta;
    double v = 1.0;
    // [PRINT]
    ulong i = 4;
    for (ulong k=1; k<n; ++k)
    {
        v *= 0.5;
    again:
        double d = ( z>=0 ? +1 : -1 );
        double tx = x + d * v * y;
```

```
        double ty = y + d * v * x;
        double tz = z - d * cordic_htab[k];
        x = tx;  y = ty;  z = tz;
        // [PRINT]
        if ( k==i )  { i=3*i+1; goto again; }
    }
    c = x;
    s = y;
}
```

The values for first steps of the computation for the argument $\theta = z_1 = 1.0$ are given in figure 33.2-B.

The scaling constant corresponding to K is K', one has

$$K' = \prod_{k=1}^{\infty} \sqrt{1-2^{-2k}} \cdot \prod_{k=0}^{\infty} \sqrt{1-2^{-2i_k}} \tag{33.2-14a}$$

$$K' = 0.828159360960215627076198327759175146869453837690842529 1\ldots \tag{33.2-14b}$$

$$\frac{1}{K'} = 1.207497067763072128877721011310915836812783221769813422\ldots \tag{33.2-14c}$$

The duplicated indices appear twice in the product. The algorithm can be used for the computation of the exponential function using $\exp(x) = \sinh(x) + \cosh(x)$. The algorithm converges if $-r' \leq z_1 \leq r'$ where

$$r' = \sum_{k=1}^{\infty} \operatorname{arctanh}(2^{-k}) + \sum_{k=0}^{\infty} \operatorname{arctanh}(2^{-i_k}) \tag{33.2-15a}$$

$$r' = 1.118173015526503803610627556783092451806572942929536106\ldots \tag{33.2-15b}$$

With arguments x_1, y_1, z_1 one has

$$x \rightarrow K'\left(x_1 \cosh(z_1) + y_1 \sinh(z_1)\right) \tag{33.2-16a}$$

$$y \rightarrow K'\left(y_1 \cosh(z_1) + x_1 \sinh(z_1)\right) \tag{33.2-16b}$$

$$z \rightarrow 0 \tag{33.2-16c}$$

which, for $x_1 = 1/K', y_1 = 0, z_1 = \theta$ specializes to the computation as above.

The backward version ($d_n := -\operatorname{sign}(y_n)$) computes

$$x \rightarrow K'\sqrt{x_1^2 - y_1^2} \tag{33.2-17a}$$

$$y \rightarrow 0 \tag{33.2-17b}$$

$$z \rightarrow z_1 - \operatorname{arctanh}\left(\frac{y_1}{x_1}\right) \tag{33.2-17c}$$

For the computation of the natural logarithm use $\log(w) = 2 \operatorname{arctanh} \frac{w-1}{w+1}$. That is, start with $x_1 = w+1$ and $y_1 = w-1$, then $z \rightarrow \frac{1}{2}\log(w)$.

The computation of the square root $\sqrt{w}$ can be obtained by starting with $x_1 = w+1/4$ and $y_1 = w-1/4$ then $z \rightarrow K'\sqrt{w}$.

The reader is referred to [14], [132] and chapter 6 of [184] for further studies.

Chapter 34

Recurrences and Chebyshev polynomials

This chapter presents algorithms and material concerning recurrences.

Firstly, several algorithms for recurrences, mostly for the case of constant coefficients, are given. Secondly, the Chebyshev polynomials are described. These are an important special case of a recurrence.

34.1 Recurrences

A sequence $[a_0, a_1, a_2, \dots]$ so that a *recurrence relation*

$$a_n \quad = \quad \sum_{j=1}^{k} m_j \, a_{n-j} \tag{34.1-1}$$

with given m_j holds for all a_j is called a k-th order *recurrence*. The recurrence is linear, homogeneous, with constant coefficients. The sequence is defined by both the recurrence relation and the first k elements.

An example, the second order recurrence relation $a_n = 1\,a_{n-1} + 1\,a_{n-2}$ together with $a_0 = 0$ and $a_1 = 1$ gives the Fibonacci numbers F_n, starting with $a_0 = 2$ and $a_1 = 1$ gives the Lucas numbers L_n:

```
  n : 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
F(n): 0  1  1  2  3  5  8 13 21 34 55 89 144 233 377
L(n): 2  1  3  4  7 11 18 29 47 76 123 199 322 521 843
```

The *characteristic polynomial* of the recurrence relation 34.1-1 is given by

$$p(x) \quad = \quad x^k - \sum_{j=1}^{k} m_j \, x^{k-j} \tag{34.1-2}$$

The definition can be motivated by writing down the recurrence relation for the element with index $n = k$:

$$0 \quad = \quad a_k - \left(\sum_{j=1}^{k} m_j \, a_{k-j} \right) \tag{34.1-3}$$

34.1.1 Fast computation using matrix powers

For the recurrence defined by the recurrence relation

$$a_n \quad := \quad m_1 \, a_{n-1} + m_2 \, a_{n-2} \tag{34.1-4}$$

and the start a_0, a_1 use the relation

$$[a_0,\ a_1]\begin{bmatrix}0 & m_2\\ 1 & m_1\end{bmatrix}^k \quad = \quad [a_k,\ a_{k+1}] \tag{34.1-5}$$

for the fast computation of an individual element a_k. The algorithm is fast when powering algorithms (see section 27.6) are used.

Note that with two consecutive terms of the recurrence in the resulting vector it is easy two compute the following terms a_{k+1}, a_{k+2}, ... using the original recurrence relation.

The generalization is straightforward. For example, a recurrence $a_n = m_1\,a_{n-1} + m_2\,a_{n-2} + m_3\,a_{n-3}$ corresponds to

$$[a_0,\ a_1,\ a_2]\begin{bmatrix}0 & 0 & m_3\\ 1 & 0 & m_2\\ 0 & 1 & m_1\end{bmatrix}^k \quad = \quad [a_k,\ a_{k+1},\ a_{k+2}] \tag{34.1-6}$$

The matrix is the companion matrix of the characteristic polynomial $x^3 - (m_1\,x^2 + m_2\,x^1 + m_3\,x^0)$, see relation 40.5-1 on page 864. Note that the indexing of the m_k is different here.

Performance

The computations are fast. As an example we give the timing of the computation of a few sequence terms with large indices. The following calculations were carried out with exact arithmetic, the post-multiply with the float 1.0 renders the output readable:

```
? M=[0,1;1,1] \\ Fibonacci sequence
? #
   timer = 1 (on)
? ([0,1]*M^10000)[1]*1.0
  time = 1 ms.
  3.364476487643 E2089
? ([0,1]*M^100000)[1]*1.0
  time = 10 ms.
  2.597406934722 E20898
? ([0,1]*M^1000000)[1]*1.0
  time = 458 ms.
  1.953282128707 E208987
```

The powering algorithm can obviously be used also for polynomial recurrences such as for the Chebyshev polynomials $T_n(x)$:

```
? M=[0,-1;1,2*x]
  [0  -1]
  [1 2*x]
? for(n=0,5,print(n,": ",([1,x]*M^n)[1]))
  0: 1
  1: x
  2: 2*x^2 - 1
  3: 4*x^3 - 3*x
  4: 8*x^4 - 8*x^2 + 1
  5: 16*x^5 - 20*x^3 + 5*x
? p=([1,x]*M^1000)[1];
  time = 1,027 ms.
? poldegree(p)
  1000
? log(polcoeff(p,poldegree(p)))/log(10)
  300.728965668317  \\ The coefficient of x^1000 is a 301-digit number
```

With modular arithmetic the quantities remain bounded and the computations can be carried out for extreme large values of n. We use the modulus $m = 2^{1279} - 1$ and compute the $n = (m+1)/4$ element of the sequence $2, 4, 14, 52, \ldots$ where $a_n = 4\,a_{n-1} - a_{n-2}$:

```
? m=2^1279-1;  \\ a 1279-bit number
? log(m)/log(10)
  385.0173 \\ 306 decimal digits
? M=Mod([0,-1;1,4],m);  \\ all entries modulo m
?  component( ([2,4]*M^((m+1)/4))[1], 2)
  time = 118 ms.
  0
```

The result is zero which proves that m is prime, see section 37.11.4. Here is a one-liner that prints all exponents $e < 1000$ of Mersenne primes:

```
? forprime(e=3,1000,m=2^e-1;M=Mod([0,-1;1,4],m);if(0==(([2,4]*M^((m+1)/4))[1]),print1(" ", e)))
  3 5 7 13 17 19 31 61 89 107 127 521 607
```

The computation takes a few seconds only.

The connection of recurrences and matrix powers is investigated in [32].

34.1.2 Faster computation using polynomial arithmetic

The matrix power algorithm for computing the k-th element of a n-th order recursion involves proportional $\log_k$ multiplications of $n \times n$ matrices. As matrix multiplication (with the straightforward algorithm) is proportional n^3 the algorithm is not optimal for recursions of high order. Note that the matrix entries grow exponentially, so the asymptotics as given is valid only for computations with bounded values such as with modular arithmetic. We will see that the involved work can be brought down from $\log k \cdot n^3$ to $\log k \cdot n^2$ and even to $\log k \cdot n \cdot \log n$.

The characteristic polynomial for the recursion $a_n := 3\,a_{n-1} + 1\,a_{n-2} + 2\,a_{n-3}$ is

$$p(x) \quad = \quad x^3 - 3\,x^2 - 1\,x - 2 \tag{34.1-7}$$

We list the first few powers of the companion matrix M of $p(x)$:

$$M^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad M^1 = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 1 \\ 0 & 1 & 3 \end{bmatrix} \qquad M^2 = \begin{bmatrix} 0 & 2 & 6 \\ 0 & 1 & 5 \\ 1 & 3 & 10 \end{bmatrix} \qquad M^3 = \begin{bmatrix} 2 & 6 & 20 \\ 1 & 5 & 16 \\ 3 & 10 & 35 \end{bmatrix} \tag{34.1-8}$$

Note that each power is a left shifted version of its predecessor, only the rightmost colum is 'new'. Now compare the columns of the matrix powers to the first few values x^k modulo $p(x)$:

$$x^0 \bmod p(x) \quad = \quad 0\,x^2 + 0\,x + 1 \tag{34.1-9a}$$
$$x^1 \bmod p(x) \quad = \quad 0\,x^2 + 1\,x + 0 \tag{34.1-9b}$$
$$x^2 \bmod p(x) \quad = \quad 1\,x^2 + 0\,x + 0 \tag{34.1-9c}$$
$$x^3 \bmod p(x) \quad = \quad 3\,x^2 + 1\,x + 2 \tag{34.1-9d}$$
$$x^4 \bmod p(x) \quad = \quad 10\,x^2 + 5\,x + 6 \tag{34.1-9e}$$
$$x^5 \bmod p(x) \quad = \quad 35\,x^2 + 16\,x + 20 \tag{34.1-9f}$$

Observe that $x^k \bmod p(x)$ corresponds to the leftmost column of M^k.

We now turn the observation into an efficient algorithm. The main routines in this section take as arguments a vector v of initial values, a vector m of recursion coefficients and an index k. The vector $r = [a_k, a_{k+1}, \ldots, a_{k+n}]$ is returned. We compute the leftmost column of M^k as $z := x^k \bmod p(x)$ and compute a_k as the scalar product of z (as a vector) and v. Our main routine is:

```
frec(v, m, k)=
{
    local(n, pc, pv, pp, px, r, t);
    n = length(m);
    if ( k<=n, return( recstep(v, m, k) ) );  \\ small indices by definition
    pc = vec2charpol(m);
    pp = Mod( x, pc );
    px = pp^(k);
    r = vector(n);
    for (i=1, n,
        t = component(px,2);
        r[i] = sum(j=1,n, v[j]*polcoeff(t,j-1,x));
        px *= pp;
    );
    return( r );
}
```

If only the value a_k is of interest, skip the computations in the final `for` loop for the values `i`> 1.

For small indices k the result is computed directly by definition, using the following auxiliary routine:

```
recstep(v, m, k)=
{ /* update v by k steps according to the recursion coefficients in m */
    local(n,r);
    if ( k<=0, return(v) );  \\ negative k is forbidden
    n = length(m);
    r = vector(n);
    for (i=1, k,
        for (j=1, n-1, r[j]=v[j+1] ); \\ shift left
        r[n] = sum(j=1,n, m[n+1-j]*v[j]);  \\ new element (convolution)
        v = r;
    );
    return( r );
}
```

The auxiliary routine used to compute the characteristic polynomial corresponding to the vector m is:

```
vec2charpol(m)=
{ /* return characteristic polynomial for the recursion coefficients in m */
    local(d,p);
    d = length(m);
    p = x^d - Pol(m,x);
    return( p );
}
```

The computation of the k-th element of a n-term recurrence involves proportional $\log k$ modular polynomial multiplications. Thereby the total cost is $\log k \cdot M(n)$ where $M(n)$ is the cost of the multiplication of two polynomials of degree n. That is, the cost is $\log k \cdot n^2$ when usual polynomial multiplication is used, and $\log k \cdot n \cdot \log n$ if an FFT scheme is applied.

The matrix power algorithm, restated for the argument structure defined above, can be implemented as:

```
mrec(v, m, k)=
{
    local(p,M);
    p = vec2charpol(m);
    M = matcompanion(p);
    M = M^k;
    return ( v * M );
}
```

All main routines can be used with symbolic values:

```
? frec([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? mrec([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
? recstep([a0,a1],[m1,m2],3)
  [m2*m1*a0 + (m1^2 + m2)*a1, (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1]
```

Performance

We check the performance (suppressing output):

```
? k=10^5;
? recstep([0,1],[1,1],k);
  time = 2,811 ms.  \\ time linear in k
? mrec([0,1],[1,1],k);
  time = 10 ms. \\ time linear in log(k)
? frec([0,1],[1,1],k);
  time = 4 ms. \\ time linear in log(k)
```

The relative performance of the routine `frec()` and `mrec()` differs more with higher orders n of the recurrence, we use $n = 10$:

```
? n=10; v=vector(n); v[n]=1; m=vector(n,j,1); k=10^5;  \\ tenth order recurrence
? mrec(v,m,k);
  time = 2,813 ms.
? f=frec(v,m,k);
  time =  159 ms.
```

```
? log(f)/log(10.0)
  [30078.67, 30078.97, 30079.27, 30079.58, 30079.88, 30080.18, \
  30080.48, 30080.78, 30081.08, 30081.38] \\ about 30k decimal digits each
```

Somewhat surprisingly, we see a performance gain greater than n even though the computations were done using integers. Finally, we repeat the computations modulo $p = 2^{521} - 1$ for $k = 10^{30}$:

```
?  n=10; v=vector(n); v[n]=1; m=vector(n,j,1); k=10^30;
?  p=2^521-1; v=Mod(v,p); m=Mod(m,p);
?  mrec(v,m,k);
   time = 312 ms.
?  frec(v,m,k);
   time = 14 ms.
```

That the performance gain with integers is not smaller than with modular arithmetic can be motivated by the fact that the quantities in both algorithms grow with the same rate. Now at each step the performance ratio should approximately equal n. Thereby the algorithms perform with the same ratio.

The computational advantage of powering modulo the characteristic polynomial versus matrix powering has been pointed out 1994 by Brent [62, p.392] (page 4 of the preprint).

34.1.3 Inhomogeneous recurrences

The fast algorithms for the computation of recurrences do only work with *homogeneous* recurrences as defined by relation 34.1-1 on page 635. A *inhomogeneous* recurrence is defined by a relation

$$a_n \quad = \quad \sum_{j=1}^{k} m_j \, a_{n-j} + P(n) \tag{34.1-10}$$

where $P(n)$ is a nonzero polynomial in n. We will show how to transform a inhomogeneous recurrence into a homogeneous recurrence of greater order.

34.1.3.1 Recurrence relations with a constant

In case a constant is to be added in an k-th order relation, one can use a recurrence of order $k+1$. From the recurrence relation $a_n = m_1 \, a_{n-1} + m_2 \, a_{n-2} + \ldots + m_k \, a_{n-k} + C$ subtract a shifted version $a_{n-1} = m_1 \, a_{n-2} + m_2 \, a_{n-3} + \ldots + m_k \, a_{n-k-1} + C$ to obtain $a_n = (m_1 + 1) \, a_{n-1} + (m_2 - m_1) \, a_{n-2} + \ldots + (m_k - m_{k-1}) \, a_{n-k}$.

An example should make the idea clear: with $a_n = 34 \, a_{n-1} - a_{n-2} + 2$ subtract a shifted version $a_{n-1} = 34 \, a_{n-2} - a_{n-3} + 2$ to obtain $a_n = 35 \, a_{n-1} - 35 \, a_{n-2} + a_{n-3}$. Setting $a_0 = 1$, $a_1 = 36$ we get, using the original relation

```
? n=7;
? ts=vector(n); ts[1]=1; ts[2]=36;
? for(k=3,n,ts[k]=34*ts[k-1]-ts[k-2]+2);
? ts
  [1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

and, using the relation without constant,

```
? ts=vector(n); ts[1]=1; ts[2]=36; ts[3]=34*ts[2]-ts[1]+2;
? for(k=4,n,ts[k]=35*ts[k-1]-35*ts[k-2]+ts[k-3]);
? ts
  [1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

34.1.3.2 The general case

If the recurrence is of the form

$$a_n \quad = \quad m_1 \, a_{n-1} + m_2 \, a_{n-2} + \ldots + m_k \, a_{n-k} + P(n) \tag{34.1-11}$$

where $P(n)$ is a polynomial of degree d in n then a homogeneous recurrence of order $k+d+1$

$$a_n \quad = \quad M_1\, a_{n-1} + M_2\, a_{n-2} + \ldots + M_{k+d+1}\, a_{n-k-d-1} \tag{34.1-12}$$

can be obtained by repeatedly subtracting a shifted relation.

The following pari/gp routine takes as input a vector of the multipliers m_i ($i = 1, \ldots, k$) and a polynomial of degree d in n. It returns a homogeneous recurrence relation as a vector $[M_1, \ldots, M_{k+d+1}]$:

```
ihom2hom(m, p)=
{
    local(d, M, k);
    if ( p==0, return(m) );
    d = poldegree(p, 'n);
    k = length(m);
    M = vector(k+d+1);
    for (j=1, k, M[j]=m[j]);
    for (s=1, d+1,
        M[1] += 1;  \\ left hand side
        for (j=2, k+s, M[j] -= m[j-1]; );
        m = M;
    );
    return(M);
}
```

In order to verify the output we use a (slow) routine that directly computes the values of an inhomogeneous recurrence:

```
ihom(v, m, k, p)=
{
    local(n, r);
    if ( k<=0, return(v[1]) );
    n = length(m);
    r = vector(n);
    for (i=1, k,
        for (j=1, n-1, r[j]=v[j+1] ); \\ shift left
        r[n] = sum(j=1,n, m[n+1-j]*v[j]);  \\ new element (convolution)
        r[n] += subst(p, 'n, i+n-1);  \\ add inhomogeneous term
        v = r;
    );
    return( r[1] );
}
```

We use the recurrence relation $a_n = 3\,a_{n-1} + 2a_{n-2} + (n^3 - n^2 - 7)$. We compute the homogeneous equivalent (intermediate values of M added):

```
? m=[3,+2];p=n^3-n^2-7;
? M=ihom2hom(m,p)
    [3, 2, 0, 0, 0, 0]
    [4, -1, -2, 0, 0, 0]
    [5, -5, -1, 2, 0, 0]
    [6, -10, 4, 3, -2, 0]
    [7, -16, 14, -1, -5, 2]
  [7, -16, 14, -1, -5, 2]  \\ a_n = 7*a_{n-1} - 16*a_{n-2} + 14*a_{n-3} +- ...
```

We can compute the first few values for the sequence starting with $a_0 = 2$, $a_1 = 5$ by the direct method:

```
? v=[2,5];
? for(k=0,9,print(k,":  ",ihom(v,m,k,p)));
  0:  2
  1:  5
  2:  16
  3:  69
  4:  280
  5:  1071
  6:  3946
  7:  14267
  8:  51134
  9:  182577
```

A vector of start values and the homogeneous equivalent allow the fast computation using the powering algorithms:

```
? V=vector(length(M),j,ihom(v,m,j-1,p))
  [2, 5, 16, 69, 280, 1071]
? for(k=0,9,print(k,":  ",frec(V,M,k)[1]));
 [- same output as with direct computation -]
```

The computation of $a_{10,000}$ now takes less than a second:

```
? z=frec(V,M,10^5)[1];  \\ result computed in 156 ms.
? 1.0*z
  1.72279531330182 E55164
? z=ihom(v,m,10^5,p);  \\ result computed in 6,768 ms.
```

34.1.4 Recurrence relations for subsequences

34.1.4.1 Two term recurrences

The recurrence for the subsequence of every k-th element of a two term recurrence $a_n = \alpha\, a_{n-1} + \beta\, a_{n-2}$ can be obtained as follows. Write

$$\begin{aligned}
a_{n+0} &= A_0\, a_n + B_0\, a_{n-0} &&= 2\, a_n \;-\; 1\, a_{n-0} && \text{(34.1-13a)}\\
a_{n+1} &= A_1\, a_n + B_1\, a_{n-1} &&= \alpha\, a_n \;+\; \beta\, a_{n-1} && \text{(34.1-13b)}\\
a_{n+2} &= A_2\, a_n + B_2\, a_{n-2} &&= (\alpha^2 + 2\beta)\, a_n \;-\; \beta^2\, a_{n-2} && \text{(34.1-13c)}\\
a_{n+3} &= A_3\, a_n + B_3\, a_{n-3} &&= (\alpha^3 + 3\alpha\beta)\, a_n \;+\; \beta^3\, a_{n-3} && \text{(34.1-13d)}\\
a_{n+4} &= A_4\, a_n + B_4\, a_{n-4} &&= (\alpha^4 + 4\alpha^2\beta + 2\beta^2)\, a_n \;-\; \beta^4\, a_{n-4} && \text{(34.1-13e)}\\
a_{n+k} &= A_k\, a_n + B_k\, a_{n-k} && && \text{(34.1-13f)}
\end{aligned}$$

We have $a_n = A_k\, a_{n-k} + B_k\, a_{n-2k}$ where $A_0 = 2$, $A_1 = \alpha$ and $A_{k+1} = \alpha\, A_k + \beta\, A_{k-1}$ (and $B_k = -(-\beta)^k$). That is, the first coefficient A_k of the recursion relations for the subsequences can be computed by the original recurrence relation. For efficient computation use

$$[A_k,\ A_{k+1}] \;=\; [2,\ \alpha] \begin{bmatrix} 0 & \beta \\ 1 & \alpha \end{bmatrix}^k \qquad \text{(34.1-14)}$$

A closed form for A_k in terms of Chebyshev polynomials is given in [30, item 14]:

$$A_k \;=\; 2\,(-\beta)^{k/2}\, T_k\left(\alpha/\sqrt{-4\,\beta}\right) \qquad \text{(34.1-15)}$$

A simple example, let F_n and L_n denote the n-th Fibonacci and Lucas number, respectively. Then $\alpha = \beta = 1$ and

$$[A_k,\ A_{k+1}] \;=\; [2,\ 1] \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^k \;=\; [L_k,\ L_{k+1}] \qquad \text{(34.1-16)}$$

That is

$$F_{k\,n+e} \;=\; L_k\, F_{k\,(n-1)+e} - (-1)^k\, F_{k\,(n-2)+e} \qquad \text{(34.1-17)}$$

where $k \in \mathbb{Z}$ and $e \in \mathbb{Z}$. The variable e expresses the shift invariance of the relation.

34.1.4.2 Recurrences of order n

For the stride-s recurrence relations of order n the following may be the most straightforward algorithm. Let $p(x)$ be the characteristic polynomial of the recurrence and M its companion matrix. Then the characteristic polynomial of M^s corresponds to the recurrence relation of the stride-s subsequence.

```
recsubseq(n, s, m=0)=
{ /* Return vector coefficients of the stride-s subsequence
   * of the n-th order linear recurrence.
   */
    local(p, M, z, r);
    if ( 0==m,
```

```
        m = vector(n,j,eval(Str("m" j)));  \\ use symbols m_j
        , /* else */
        n = length(m);  \\ m given
     );
    p = vec2charpol(m);
    M = matcompanion(p);
    z = x^n-charpoly(M^s);
    r = vector(n,j,polcoeff(z,n-j,x));
    return( r );
}
```

For the second order recurrence we get what we have already seen for $s = 0, \ldots, 4$:

```
? m=[a,b];
? for(s=-2,5,print(s,":  ",recsubseq(0,s,m)););
  -2:  [1/b^2*a^2 + 2/b,   -1/b^2]
  -1:  [-1/b*a,   1/b]
   0:  [2,   -1]
   1:  [a,   b]
   2:  [a^2 + 2*b,   -b^2]
   3:  [a^3 + 3*b*a,   b^3]
   4:  [a^4 + 4*b*a^2 + 2*b^2,   -b^4]
   5:  [a^5 + 5*b*a^3 + 5*b^2*a,   b^5]
```

For the third order recurrence we get:

```
? m=[a,b,c];
? for(s=-2,5,print(s,":  ",recsubseq(0,s,m)););
  -2:  [2/-c*a - 1/-c^2*b^2,   1/-c^2*a^2 + 2/-c^2*b,   1/c^2]
  -1:  [-1/c*b,   1/-c*a,   1/c]
   0:  [3,   -3,   1]
   1:  [a,   b,   c]
   2:  [a^2 + 2*b,   2*c*a - b^2,   c^2]
   3:  [a^3 + 3*b*a + 3*c,   -3*c*b*a + (b^3 - 3*c^2),   c^3]
   4:  [a^4 + 4*b*a^2 + 4*c*a + 2*b^2, \
        -2*c^2*a^2 + 4*c*b^2*a + (-b^4 + 4*c^2*b), \
        c^4]
   5:  [a^5 + 5*b*a^3 + 5*c*a^2 + 5*b^2*a + 5*c*b, \
        5*c^2*b*a^2 + (-5*c*b^3 + 5*c^3)*a + (b^5 - 5*c^2*b^2), \
        c^5]
```

34.1.5 Generating functions for recurrences

A generating function for a recurrence has a power series where the k-th coefficient equals the k-th term of the recurrence. For example, for the Fibonacci- and Lucas numbers:

$$\frac{x}{1-x-x^2} = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + 13x^7 + 21x^8 + 34x^9 + \ldots \quad (34.1\text{-}18a)$$

$$= \sum_{k=0}^{\infty} F_k\, x^k \quad (34.1\text{-}18b)$$

$$\frac{2-x}{1-x-x^2} = 2 + x + 3x^2 + 4x^3 + 7x^4 + 11x^5 + 18x^6 + 29x^7 + 47x^8 + \ldots \quad (34.1\text{-}18c)$$

$$= \sum_{k=0}^{\infty} L_k\, x^k \quad (34.1\text{-}18d)$$

In general, for a recurrence $a_n = \sum_{k=1}^{K} m_k\, a_{n-k}$ with given a_0, a_1, …, a_K one has

$$\frac{\sum_{j=0}^{K-1} b_j\, x^j}{1 - \sum_{j=1}^{K} m_j\, x^j} = \sum_{j=0}^{\infty} a_j\, x^j \quad (34.1\text{-}19a)$$

where the denominator is the reciprocal polynomial of the characteristic polynomial and

$$b_0 = a_0 \tag{34.1-20a}$$
$$b_1 = a_1 - (a_0\, m_1) \tag{34.1-20b}$$
$$b_2 = a_2 - (a_0\, m_2 + a_1\, m_1) \tag{34.1-20c}$$
$$b_3 = a_3 - (a_0\, m_3 + a_1\, m_2 + a_2\, m_1) \tag{34.1-20d}$$
$$b_k = a_k - \sum_{j=0}^{k-1} a_j\, m_{k-j} \tag{34.1-20e}$$

As an example we choose the sequence

$$[0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, \ldots] \tag{34.1-21}$$

with the recurrence relation $a_n = a_{n-1} + a_{n-2} + a_{n-2}$:

```
? a=[0,0,1]~;
? m=[1,1,1]~;
? K=length(m);
? b=vector(K, k, a[k]-sum(j=0,k-2, a[j+1]*m[k-j-1]))
  [0, 0, 1]
? pb=sum(j=0,K-1,b[j+1]*x^j)
  x^2
? pr=1-sum(k=1,K,m[k]*x^k) \\ reciprocal of charpoly
  -x^3 - x^2 - x + 1
? gen=pb/pr  \\ the generating function
  x^2/(-x^3 - x^2 - x + 1)
? t=taylor(gen, x)
  x^2 + x^3 + 2*x^4 + 4*x^5 + 7*x^6 + 13*x^7 + 24*x^8 + 44*x^9 + 81*x^10
  + 149*x^11 + 274*x^12 + 504*x^13 + 927*x^14 + 1705*x^15 + 3136*x^16 + O(x^17)
? t=truncate(t);
? for(j=0,poldegree(t),print1("  ",polcoeff(t,j)))
  0  0  1  1  2  4  7  13  24  44  81  149  274  504  927  1705  3136
```

Note that the denominator is the reciprocal of the characteristic polynomial. The general form of the expressions for a two term linear recurrence can be obtained using symbols:

```
? a=[a0,a1]~;
? m=[m1,m2]~;
? K=length(m);
? b=vector(K,k,a[k]-sum(j=0,k-2,a[j+1]*m[k-j-1]))
  [a0, -m1*a0 + a1]
? pb=sum(j=0,K-1,b[j+1]*x^j)
  (-m1*a0 + a1)*x + a0
? pr=1-sum(k=1,K,m[k]*x^k)
  -m2*x^2 - m1*x + 1
? gen=pb/pr  \\ the generating function
  ((-m1*a0 + a1)*x + a0)/(-m2*x^2 - m1*x + 1)
? t=taylor(gen,x);
? t=truncate(t);
? for(j=0,poldegree(t),print(j,":  ",polcoeff(t,j)))
  0:  a0
  1:  a1
  2:  m2*a0 + m1*a1
  3:  m2*m1*a0 + (m1^2 + m2)*a1
  4:  (m2*m1^2 + m2^2)*a0 + (m1^3 + 2*m2*m1)*a1
  5:  (m2*m1^3 + 2*m2^2*m1)*a0 + (m1^4 + 3*m2*m1^2 + m2^2)*a1
  6:  (m2*m1^4 + 3*m2^2*m1^2 + m2^3)*a0 + (m1^5 + 4*m2*m1^3 + 3*m2^2*m1)*a1
  7:  (m2*m1^5 + 4*m2^2*m1^3 + 3*m2^3*m1)*a0 + (m1^6 + 5*m2*m1^4 + 6*m2^2*m1^2 + m2^3)*a1
```

34.1.6 Binet forms for recurrences

A closed form expression for the Fibonacci numbers is

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{34.1-22}$$

For a two-term recurrence $a_n = m_1\, a_{n-1} + m_2\, a_{n-2}$ a closed form solution is given by

$$a_n \quad = \quad \frac{1}{w}\,[(a_1 - a_0\, M)\, P^n - (a_1 - a_0\, P)\, M^n] \tag{34.1-23a}$$

where $w = \sqrt{m_1^2 + 4\, m_2}$, $P = (m_1 + w)/2$ and $M = (m_1 - w)/2$.

In general such formulas can be obtained as exemplified using a three-term recurrence: let $a_n = m_1\, a_{n-1} + m_2\, a_{n-2} + m_3\, a_{n-3}$, its characteristic polynomial is $p(x) = x^3 - (m_1\, x^2 + m_2\, x + m_3)$. Let r_0, r_1, r_2 be the roots of $p(x)$, then $a_n = c_0\, r_0^n + c_1\, r_1^n + c_2\, r_2^n$ if c_0, c_1, c_2 satisfy

$$a_0 \quad = \quad c_0 + \quad c_1 + \quad c_2 \tag{34.1-24a}$$

$$a_1 \quad = \quad r_0\, c_0 + r_1\, c_1 + r_2\, c_2 \tag{34.1-24b}$$

$$a_2 \quad = \quad r_0^2\, c_0 + r_1^2\, c_1 + r_2^2\, c_2 \tag{34.1-24c}$$

That is, we have to solve the matrix equation $Z \cdot c = a$ for the vector c where a is the vector of starting values and

$$Z \quad = \quad \begin{bmatrix} 1 & 1 & 1 \\ r_0 & r_1 & r_2 \\ r_0^2 & r_1^2 & r_2^2 \end{bmatrix} \tag{34.1-25}$$

Verification with the three term recurrence $a_n = a_{n-1} + a_{n-2} + a_{n-2}$ starting with $a_0 = a_1 = 0$ and $a_2 = 1$:

```
? a=[0,0,1]~;
? m=[1,1,1]~;
? K=length(m);
? p=x^K-sum(k=1,K,m[k]*x^(K-k)) \\ characteristic polynomial
  x^3 - x^2 - x - 1
? r=(polroots(p))
  [1.8392867,   -0.419643 - 0.606290*I,  -0.419643 + 0.6062907*I]~
? Z=matrix(K,K,ri,ci,r[ci]^(ri-1))
  [1 1 1]
  [1.839286   -0.4196433 - 0.6062907*I   -0.4196433 + 0.6062907*I]
  [3.382975   -0.1914878 + 0.5088517*I   -0.1914878 - 0.5088517*I]

? c=matsolve(Z,a)
  [0.1828035 + 1.8947 E-20*I,  -0.09140176 - 0.3405465*I,  -0.0914017 + 0.3405465*I]~
? norm(Z*c-a) \\ check solution
  [1.147 E-39,  6.795 E-39,  3.673 E-39]~
? seq(n)=sum(k=0,K-1,c[k+1]*r[k+1]^n)
? for(n=0,20,print1(" ",round(seq(n))))
  0 0 1 1 2 4 7 13 24 44 81 149 274 504 927 1705 3136 5768 10609 19513 35890
```

The method fails if the characteristic polynomial has multiple roots because then the matrix Z is singular.

34.1.6.1 The special case $c_k = 1$

Let $p(x)$ be the characteristic polynomial of a recurrence, with roots r_i: $p(x) = \prod_k (x - r_k)$. We want to determine the generating function for the recurrence such that $a_j = \sum_k r_k^j$ (that is, all constants c_k are one). For the reciprocal polynomial h of p we have $h(x) = \prod_k (1 - r_k\, x_k)$, and (using the product rule for differentiation)

$$h'(x) \quad = \quad h(x) \sum_k \frac{-r_k}{1 - r_k\, x} \tag{34.1-26}$$

With $r/(1 - r\, x) = \sum_{j \geq 0} r^{j+1}\, x^j$ we find that

$$-\frac{h'(x)}{h(x)} \quad = \quad \sum_{j \geq 0} \left(\sum_k r_k^{j+1} \right) x^j \tag{34.1-27}$$

That is $a_j = \sum_k r_k^{j+1}$, and $c_k = 1$ for all k. The relation is the key to the fast computation of the trace vector in finite fields, see relation 40.3-6 on page 861.

34.1.6.2 Binet form with multiple roots of the characteristic polynomial

When the characteristic polynomial has multiple roots the Binet form has coefficients that are polynomials in n. For example, for the characteristic polynomials $p(x) = (x - r_0)^3 (x - r_1)$ the Binet from would be $a_n = (c_0 + n\,d_0 + n^2\,e_0)\,r_0^n + c_1\,r_1^n$. With $n = 0$, 1, and 2 we obtain the system of equations

$$a_0 = (c_0 + 0\,d_0 + 0^2\,e_0) + c_1 \tag{34.1-28a}$$
$$a_1 = r_0\,(c_0 + 1\,d_0 + 1^2\,e_0) + r_1\,c_1 \tag{34.1-28b}$$
$$a_2 = r_0^2\,(c_0 + 2\,d_0 + 2^2\,e_0) + r_1^2\,c_1 \tag{34.1-28c}$$

In general, the coefficient of the power of the k-th root r_k in the Binet form must be a polynomial of degree $m_k - 1$ where m_k is the multiplicity of r_k.

34.1.7 Logarithms of generating functions *

A seemingly mysterious relation for the generating function of the Fibonacci numbers

$$f(x) := \frac{1}{1 - x - x^2} = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + \ldots = \sum_{k=0}^{\infty} F_{k+1}\,x^k \tag{34.1-29}$$

is

$$\log(f(x)) = x + \frac{1}{2}\,3\,x^2 + \frac{1}{3}\,4\,x^3 + \frac{1}{4}\,7\,x^4 + \frac{1}{5}\,11\,x^5 + \ldots = \sum_{k=1}^{\infty} \frac{1}{k}\,L_k\,x^k \tag{34.1-30}$$

where L_k are the Lucas numbers. Similarly,

$$g(x) := \frac{1}{1 - 2\,x - x^2} = 1 + 2x + 5x^2 + 12x^3 + 29x^4 + 70x^5 + 169x^6 + \ldots \tag{34.1-31a}$$
$$\log(g(x)) = 2\left[x + \frac{1}{2}\,3\,x^2 + \frac{1}{3}\,7\,x^3 + \frac{1}{4}\,17\,x^4 + \frac{1}{5}\,41\,x^5 + \frac{1}{6}\,99\,x^6 + \ldots\right] \tag{34.1-31b}$$

Now set $f(x) =: \frac{1}{h(x)}$, then

$$\frac{d}{d\,x}\log(f(x)) = \frac{d}{d\,x}\log\left(\frac{1}{h(x)}\right) = -\frac{h'(x)}{h(x)} \tag{34.1-32}$$

The expression $\frac{h'(x)}{h(x)}$ is again the generating function of a recurrence and formal integration of the Taylor series terms gives the factors $\frac{1}{k}$. The observation is a special case of the algorithm for the computation of the logarithm for powers series given in section 31.3 on page 606.

34.2 Chebyshev polynomials

The *Chebyshev polynomials* of the first (T) and second (U) kind can be defined by the functions

$$T_n(x) = \cos\left[n\,\arccos(x)\right] \tag{34.2-1a}$$
$$U_n(x) = \frac{\sin\left[(n+1)\,\arccos(x)\right]}{\sqrt{1 - x^2}} \tag{34.2-1b}$$

For integral n both of them are polynomials. The first few polynomials are given in figure 34.2-A (first kind) and figure 34.2-B (second kind).

$$
\begin{aligned}
T_{-n}(x) &= T_n(x) \\
T_{-1}(x) &= x \\
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2\,x^2 - 1 \\
T_3(x) &= 4\,x^3 - 3\,x \\
T_4(x) &= 8\,x^4 - 8\,x^2 + 1 \\
T_5(x) &= 16\,x^5 - 20\,x^3 + 5\,x \\
T_6(x) &= 32\,x^6 - 48\,x^4 + 18\,x^2 - 1 \\
T_7(x) &= 64\,x^7 - 112\,x^5 + 56\,x^3 - 7\,x \\
T_8(x) &= 128\,x^8 - 256\,x^6 + 160\,x^4 - 32\,x^2 + 1 \\
T_9(x) &= 256\,x^9 - 576\,x^7 + 432\,x^5 - 120\,x^3 + 9\,x \\
T_{10}(x) &= 512\,x^{10} - 1280\,x^8 + 1120\,x^6 - 400\,x^4 + 50\,x^2 - 1 \\
T_{11}(x) &= 1024\,x^{11} - 2816\,x^9 + 2816\,x^7 - 1232\,x^5 + 220\,x^3 - 11\,x
\end{aligned}
$$

Figure 34.2-A: The first few Chebyshev polynomials of the first kind.

$$
\begin{aligned}
U_{-n}(x) &= -U_{n-2}(x) \\
U_{-2}(x) &= -1 \\
U_{-1}(x) &= 0 \\
U_0(x) &= 1 \\
U_1(x) &= 2\,x \\
U_2(x) &= 4\,x^2 - 1 \\
U_3(x) &= 8\,x^3 - 4\,x \\
U_4(x) &= 16\,x^4 - 12\,x^2 + 1 \\
U_5(x) &= 32\,x^5 - 32\,x^3 + 6\,x \\
U_6(x) &= 64\,x^6 - 80\,x^4 + 24\,x^2 - 1 \\
U_7(x) &= 128\,x^7 - 192\,x^5 + 80\,x^3 - 8\,x \\
U_8(x) &= 256\,x^8 - 448\,x^6 + 240\,x^4 - 40\,x^2 + 1 \\
U_9(x) &= 512\,x^9 - 1024\,x^7 + 672\,x^5 - 160\,x^3 + 10\,x \\
U_{10}(x) &= 1024\,x^{10} - 2304\,x^8 + 1792\,x^6 - 560\,x^4 + 60\,x^2 - 1 \\
U_{11}(x) &= 2048\,x^{11} - 5120\,x^9 + 4608\,x^7 - 1792\,x^5 + 280\,x^3 - 12\,x
\end{aligned}
$$

Figure 34.2-B: The first few Chebyshev polynomials of the second kind.

One has

$$T_n(x) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k-1)!}{k!\,(n-2k)!} (2x)^{n-2k} \tag{34.2-4a}$$

$$= \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{1}{n-k} \binom{n-k}{k} (2x)^{n-2k} \tag{34.2-4b}$$

$$= \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2\,k} x^{n-2k} (x^2-1)^k \tag{34.2-4c}$$

and

$$U_n(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k)!}{k!\,(n-2k)!} (2x)^{n-2k} \tag{34.2-5a}$$

$$= \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \binom{n-k}{k} (2x)^{n-2k} \tag{34.2-5b}$$

$$= \sum_{k=0}^{\lfloor n/2+1 \rfloor} \binom{n+1}{2\,k+1} x^{n-2k} (x^2-1)^k \tag{34.2-5c}$$

The indexing of U seems to be slightly unfortunate, having $U_0 = 0$ would render many of the relations for the Chebyshev polynomials more symmetric.

The $n+1$ extrema of $T_n(x)$ are located at the points $x_k = \cos \frac{k\,\pi}{n}$ where $k = 0, 1, 2, \ldots, n$ and $-1 \leq x_k \leq +1$, which can be seen from the definition. The values at those points are ± 1. The n zeros lie at $x_k = \cos \frac{(k-1/2)\,\pi}{n}$ where $k = 1, 2, 3, \ldots, n$.

The expansion of x^n in terms of Chebyshev polynomials of the first kind is, for n even,

$$x^n = \frac{1}{2^n} \binom{n}{n/2} + \frac{1}{2^{n-1}} \sum_{k=0}^{n/2-1} \binom{n}{k} T_{n-2k}(x) \tag{34.2-6a}$$

and, for odd n,

$$x^n = \frac{1}{2^{n-1}} \sum_{k=0}^{(n-1)/2} \binom{n}{k} T_{n-2k}(x) \tag{34.2-6b}$$

For the Chebyshev polynomials of the first kind one has

$$T_n\left(\frac{x+1/x}{2}\right) = \frac{x^n + 1/x^n}{2} \tag{34.2-7}$$

This relation can be used to find a solution of $T_n(x) = z$ directly. Indeed

$$x = \frac{R_n + 1/R_n}{2} \quad \text{where} \quad R_n := \left(z + \sqrt{z^2-1}\right)^{1/n} \tag{34.2-8}$$

is a solution which can be chosen to be real if $z \in \mathbb{R}$ and $z > 1$. Thereby we have the closed form expression

$$T_n(z) = \frac{r^n + r^{-n}}{2} \quad \text{where} \quad r := \left(z + \sqrt{z^2-1}\right) \tag{34.2-9}$$

34.2.1 Recurrence relation, generating functions, and the composition law

Both types of Chebyshev polynomials obey the same recurrence (omitting the argument x)

$$N_n = 2\,x\,N_{n-1} - N_{n-2} \tag{34.2-10}$$

where N can be either symbol, T or U. Recurrence relations for subsequences are:

$$\begin{aligned}
N_{n+1} &= [2\,x]\cdot N_n - N_{n-1} && \text{(34.2-11a)}\\
N_{n+2} &= \left[2\,(2x^2-1)\right]\cdot N_n - N_{n-2} && \text{(34.2-11b)}\\
N_{n+3} &= \left[2\,(4x^3-3x)\right]\cdot N_n - N_{n-3} && \text{(34.2-11c)}\\
N_{n+4} &= \left[2\,(8x^4-8x^2+1\right]\cdot N_n - N_{n-4} && \text{(34.2-11d)}\\
N_{n+5} &= \left[2\,(16x^5-20x^3+5x)\right]\cdot N_n - N_{n-5} && \text{(34.2-11e)}\\
N_{n+s} &= [2\,T_s(x)]\cdot N_n - N_{n-s} && \text{(34.2-11f)}
\end{aligned}$$

The generating functions are

$$\frac{1-x\,t}{1-2\,x\,t+t^2} = \sum_{n=0}^{\infty} t^n\,T_n(x) \tag{34.2-12a}$$

$$\frac{1}{1-2\,x\,t+t^2} = \sum_{n=0}^{\infty} t^n\,U_n(x) \tag{34.2-12b}$$

Quick check of relation 34.2-12a using pari/gp:

```
? gen=truncate(taylor((1-t*x)/(1-2*x*t+t^2),t));
? for(k=0,5,print(k,":  ",polcoeff(gen,k,t)));
 0:  1
 1:  x
 2:  2*x^2 - 1
 3:  4*x^3 - 3*x
 4:  8*x^4 - 8*x^2 + 1
 5:  16*x^5 - 20*x^3 + 5*x
```

Binet forms for T (compare with relation 34.2-9 on the previous page) and U are

$$T_n(z) = \frac{1}{2}\left[\left(z+\sqrt{z^2-1}\right)^n + \left(z-\sqrt{z^2-1}\right)^n\right] \tag{34.2-13a}$$

$$U_n(z) = \frac{1}{2\sqrt{z^2-1}}\left[\left(z+\sqrt{z^2-1}\right)^{n+1} - \left(z-\sqrt{z^2-1}\right)^{n+1}\right] \tag{34.2-13b}$$

Composition is multiplication of indices as can be seen by the definition (relation 34.2-1a on page 645):

$$T_n(T_m(x)) = T_{n\,m}(x) \tag{34.2-14}$$

For example,

$$\begin{aligned}
T_{2n}(x) &= T_2(T_n(x)) = 2\,T_n^2(x) - 1 && \text{(34.2-15a)}\\
&= T_n(T_2(x)) = T_n(2\,x^2-1) && \text{(34.2-15b)}
\end{aligned}$$

34.2.2 Index-doubling and relations between T and U

Index-doubling relations for the polynomials of the first kind are

$$\begin{aligned}
T_{2n} &= 2\,T_n^2 - 1 && \text{(34.2-16a)}\\
T_{2n+1} &= 2\,T_{n+1}\,T_n - x && \text{(34.2-16b)}\\
T_{2n-1} &= 2\,T_n\,T_{n-1} - x && \text{(34.2-16c)}
\end{aligned}$$

Similar relations for the polynomials of the second kind are

$$\begin{aligned}
U_{2n} &= U_n^2 - U_{n-1}^2 \;=\; (U_n + U_{n-1})\,(U_n - U_{n-1}) && \text{(34.2-17a)}\\
&= U_n\,(U_n - U_{n-2}) - 1 \;=\; U_{n-1}\,(U_{n+1} - U_{n-1}) + 1 && \text{(34.2-17b)}\\
U_{2n+1} &= U_n\,(U_{n+1} - U_{n-1}) && \text{(34.2-17c)}\\
&= 2\,U_n\,(U_{n+1} - x\,U_n) \;=\; 2\,U_n\,(x\,U_n - U_{n-1}) && \text{(34.2-17d)}\\
U_{2n-1} &= U_{n-1}\,(U_n - U_{n-2}) && \text{(34.2-17e)}\\
&= 2\,U_{n-1}\,(U_n - x\,U_{n-1}) \;=\; 2\,U_{n-1}\,(x\,U_{n-1} - U_{n-2}) && \text{(34.2-17f)}
\end{aligned}$$

Some relations between T and U are

$$\begin{aligned}
T_n &= U_n - x\,U_{n-1} \;=\; x\,U_{n-1} - U_{n-2} \;=\; \frac{1}{2}\,(U_n - U_{n-2}) && \text{(34.2-18a)}\\
T_{n+1} &= x\,T_n - (1-x^2)\,U_{n-1} && \text{(34.2-18b)}\\
U_{2n} &= 2\,T_n\,U_n - 1 && \text{(34.2-18c)}\\
U_{2n-1} &= 2\,T_n\,U_{n-1} \;=\; 2\,(T_{n+1}\,U_n + x) && \text{(34.2-18d)}\\
U_{2n+1} &= 2\,T_{n+1}\,U_n \;=\; 2\,(T_{n+2}\,U_{n-1} + x) && \text{(34.2-18e)}\\
U_{2^n-1} &= 2^n \prod_{k=0}^{n-1} T_{2^k} && \text{(34.2-18f)}
\end{aligned}$$

Relation 34.2-18b, written as

$$U_n \;=\; \frac{x\,T_{n+1} - T_{n+2}}{1-x^2} \;=\; \frac{T_n - x\,T_{n+1}}{1-x^2} \qquad \text{(34.2-19)}$$

can be used to compute the polynomials of the second kind from those of the first kind. One further has:

$$\begin{aligned}
T_{n+m} + T_{n-m} &= 2\,T_n\,T_m && \text{(34.2-20a)}\\
T_{n+m} - T_{n-m} &= 2\,(x^2+1)\,U_{n-1}\,U_{m-1} && \text{(34.2-20b)}\\
U_{n+m-1} + U_{n-m-1} &= 2\,U_{n-1}\,T_m && \text{(34.2-20c)}\\
U_{n+m-1} - U_{n-m-1} &= 2\,T_n\,U_{m-1} && \text{(34.2-20d)}
\end{aligned}$$

Expressions for certain sums:

$$\begin{aligned}
\sum_{k=0}^{n} T_{2k} &= \frac{1}{2}\,(1 + U_{2n}) && \text{(34.2-21a)}\\
\sum_{k=0}^{n-1} T_{2k+1} &= \frac{1}{2}\,U_{2n-1} && \text{(34.2-21b)}\\
\sum_{k=0}^{n} U_{2k} &= \frac{1 - T_{2n+2}}{2\,(1-x^2)} && \text{(34.2-21c)}\\
\sum_{k=0}^{n-1} U_{2k+1} &= \frac{x - T_{2n+1}}{2\,(1-x^2)} && \text{(34.2-21d)}
\end{aligned}$$

Using $\partial_x \cos(n\,\arccos(x)) = n\,\sin(n\,\arccos(x))/\sqrt{1-x^2}$ we obtain

$$\partial_x T_n(x) \;=\; n\,U_{n-1}(x) \qquad \text{(34.2-22)}$$

34.2.3 Fast computation of the Chebyshev polynomials

We give algorithms that improve on both the matrix power, and the polynomial based algorithms.

34.2.3.1 Chebyshev polynomials of the first kind

For even index use relation 34.2-16a ($T_{2n} = 2\,T_n^2 - 1$). For odd index we use relations 34.2-16c and 34.2-16b. We compute the pair $[T_{n-1},\, T_n]$ recursively via

$$[T_{n-1},\, T_n] \;=\; \left[2\,T_{q-1}\,T_q - x,\; 2\,T_q^2 - 1\right] \quad \text{where} \quad q = n/2, \quad \text{if } n \text{ even} \tag{34.2-23a}$$

$$[T_{n-1},\, T_n] \;=\; \left[2\,T_{q-1}^2 - 1,\; 2\,T_{q-1}\,T_q - x\right] \quad \text{where} \quad q = (n+1)/2, \quad \text{if } n \text{ odd} \tag{34.2-23b}$$

Note that no multiplication with x occurs thereby the computation is efficient also for floating point arguments. With integer x the cost of the computation of $T_n(x)$ is $\sim M(n)$ where $M(n)$ is the cost of a multiplication of numbers with the precision of the result. When x is a floating point number the cost is $\sim \log_2(n)\, M(n)$ where $M(n)$ is the cost of a multiplication with the precision used.

The code for the pair computations is

```
fvT(n, x)=
{ /* return  [ T(n-1,x), T(n,x) ]  */
    local(nr, t, t1, t2);
    if ( n<=1,
        if ( 1==n, return( [1, x] ) );
        if ( 0==n, return( [x, 1] ) );
        if ( -1==n, return( [2*x^2-1, x] ) );
        return( 0 );  \\ disallow negative index < -1
    );
    nr = (n+1) >> 1; \\   if ( "n even",  nr = n/2 , nr = (n+1)/2; );
    vr = fvT(nr, x);  \\ recursion
    t1 = vr[1];  t2 = vr[2];
    if ( !bitand(n,1), \\ n is even
        t = [2*t1*t2-x, 2*t2^2-1];
        ,
        t = [2*t1^2-1, 2*t1*t2-x];
     );
    return( t );
}
```

The function to be called by the user is

```
fT(n, x)=
{
    local(q, t, v, T);
    n = abs(n);
    if ( n<=1,
        if (n>=0,  return(if(0==n,1,x)));
        return( fT(-n, x) );
    );
    t = 0;  q = n;
    while ( 0==bitand(q, 1), q>>=1; t+=1; );
    \\ here: n==q*2^t
    T = fvT(q, x)[2];
    while ( t,  T=2*T*T-1; t-=1; );
    return( T );
}
```

We check the speedup by comparing with the matrix-power computation that gives identical results. We compute $T_{4{,}545{,}967}(2)$, a number with more than 2,600,000 decimal digits:

```
vT(n,x)= return( ([1, x]*[0,-1; 1,2*x]^n) );
x=2;  \\ want integer calculations
n=4545967;
vT(n,x);   \\ computed in 9,800 ms.
fvT(n,x);  \\ computed in 2,241 ms.
```

C++ implementations for the computation of $T_n(2)$ and $T_n(x)$ modulo m are given in [FXT: mod/chebyshev1.cc].

34.2.3.2 Chebyshev polynomials of the second kind

One can use the fast algorithm for the polynomials of the first kind and relation 34.2-19 ($U_n = (T_n - x\,T_{n+1})/(1-x^2)$, involving a division):

```
fU(n, x)=
{
    local(v);
    if ( 1==x, return(n+1) );  \\ avoid division by zero
    v = fvT(n+1, x);
    return( (v[1]-x*v[2])/(1-x^2) );
}
```

We give an additional algorithm that uses 3 multiplication for each reduction of the index n. One multiplication is by the variable x. We compute the pair $[U_{n-1},\, U_n]$ recursively via

$$M_q := (U_q + U_{q-1})\,(U_q - U_{q-1}) \tag{34.2-24a}$$

$$[U_{n-1},\, U_n] = [2\,U_{q-1}\,(U_q - x\,U_{q-1}),\; M_q] \quad \text{where} \quad q = n/2, \quad \text{if } n \text{ even} \tag{34.2-24b}$$

$$[U_{n-1},\, U_n] = [M_q,\; 2\,U_q\,(x\,U_q - U_{q-1})] \quad \text{where} \quad q = (n-1)/2, \quad \text{if } n \text{ odd} \tag{34.2-24c}$$

The code for the pair computations is

```
fvU(n, x)=
{ /* return  [ U(n-1,x), U(n,x) ] */
    local(nr, u1, u0, ue, t, u);
    if ( n<=1,
        if ( 1==n, return( [1, (2*x)] ) );
        if ( 0==n, return( [0, 1] ) );
        if ( -1==n, return( [-1, 0] ) );
        if ( -2==n, return( [-(2*x), -1] ) );
        return( 0 );  \\ disallow negative index < -2
    );
    nr = n >> 1;  \\ if ( "n even",  nr = n/2 , nr = (n-1)/2; );
    vr = fvU(nr, x);  \\ recursion
    u1 = vr[1];  u0 = vr[2];
    ue = (u0+u1) * (u0-u1);
    if ( !bitand(n,1),  \\ n is even
        t = u1*(u0-x*u1);  t+=t;
        u = [t,  ue];
        ,
        t = u0*(x*u0-u1);  t+=t;
        u = [ue, t];
    );
    return( u );
}
```

The function to be called by the user is

```
fU(n, x)=  return( fvU(n,x)[2] );
```

The comparison with the matrix-power computation shows almost the same speedup as for the polynomials of the first kind:

```
vU(n,x)= return( [0, 1]*[0,-1; 1,2*x]^n );
x=2;  \\ want integer calculations
n=4545967;
vU(n,x);   \\ computed in 9,783 ms.
fvU(n,x);  \\ computed in 2,704 ms.
```

C++ implementations for the computation of $U_n(2)$ and $U_n(x)$ modulo m are given in [FXT: mod/chebyshev2.cc].

34.2.3.3 Symbolic computation

For symbolic computations the explicit power series as in 34.2-4a or 34.2-5a on page 647 should be preferred. The following routine computes T_n as a polynomial in x:

```
chebyTrec(n)=
{
    local(b, s);
    b = 2^(n-1);
    if ( 0==n%2, if ( 0==n%4, s=+1, s=-1 ), s=0 );
    forstep (k=n, 1, -2,
        s += b*'x^(k);
        b *= -(k*(k-1))/((n+k-2)*(n-k+2));
    );
    return( Pol(s) );
}
```

For U_n one can use

```
chebyUrec(n)=
{
    local(b, s);
    n += 1;
    b = 2^(n-1)/n;
    s = 0;
    forstep (k=n, 1, -2,
        s += (k)*b*'x^(k-1);
        b *= -(k*(k-1))/((n+k-2)*(n-k+2));
    );
    return( Pol(s) );
}
```

34.2.4 Relations to approximations of the square root *

34.2.4.1 Padé approximants for $\sqrt{x^2 \pm 1}$

We start with the relation (from the definitions 34.2-1a and 34.2-1b on page 645, and $sin^2 + cos^2 = 1$)

$$T_n^2 - (x^2 - 1)\, U_{n-1}^2 \quad = \quad 1 \tag{34.2-25}$$

which we write as

$$\sqrt{x^2 - 1} \quad = \quad \sqrt{\frac{T_n^2 - 1}{U_{n-1}^2}} \tag{34.2-26}$$

If we define $R_n = T_n / U_{n-1}$, then

$$R_n(x) \quad = \quad \frac{T_n}{U_{n-1}} \;\approx\; \sqrt{x^2 - 1} \tag{34.2-27}$$

A composition law holds for R:

$$R_{m\,n}(x) \quad = \quad R_m(R_n(x)) \tag{34.2-28}$$

We list the first few values of $R_k(2)$ and $R_k(x)$:

k:	$R_k(2)$	$R_k(x)$
1:	$2/1$	$\frac{x}{1}$
2:	$7/4$	$\frac{2x^2 - 1}{2x}$
3:	$26/15$	$\frac{4x^3 - 3x}{4x^2 - 1}$
4:	$97/56$	$\frac{8x^4 - 8x^2 + 1}{8x^3 - 4x}$
5:	$362/209$	$\frac{16x^5 - 20x^3 + 5x}{16x^4 - 12x^2 + 1}$
∞:	$\sqrt{3}$	$\sqrt{x^2 - 1}$

(34.2-29)

If we define $T_n^+(x) := T(i\,x)/i^n$ and $U_n^+(x) := U(i\,x)/i^n$ then

$$T_n^{+\,2} - (x^2+1)\,U_{n-1}^{+\,2} \quad = \quad 1 \tag{34.2-30}$$

Defining $R_n^+ := T_n^+/U_{n-1}^+$ we have

$$R_{m\,n}^+(x) \quad = \quad R_m^+(R_n^+(x)) \tag{34.2-31}$$

and

$$\sqrt{x^2+1} \quad = \quad \sqrt{\frac{T_n^{+\,2}-1}{U_{n-1}^{+\,2}}} \quad \approx \quad \frac{T_n^+}{U_{n-1}^+} \;=\; R_n^+(x) \tag{34.2-32}$$

The first few values of $R_k^+(1)$ and $R_k^+(x)$ are

$$\begin{array}{cccc} k\text{:} & R_k^+(1) & R_k^+(x) \\ 1\text{:} & 1/1 & \dfrac{x}{1} \\ 2\text{:} & 3/2 & \dfrac{2x^2+1}{2x} \\ 3\text{:} & 7/5 & \dfrac{4x^3+3x}{4x^2+1} \\ 4\text{:} & 17/12 & \dfrac{8x^4+8x^2+1}{8x^3+4x} \\ 5\text{:} & 41/29 & \dfrac{16x^5+20x^3+5x}{16x^4+12x^2+1} \\ \infty\text{:} & \sqrt{2} & \sqrt{x^2+1} \end{array} \tag{34.2-33}$$

Relations 34.2-30 and 34.2-25 can be used to power solutions of Pell's Diophantine equation, see section 37.13.2 on page 780.

34.2.4.2 Two products for the square root

For those fond of products: for $d > 0$, $d \neq 1$

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty}\left(1+\frac{1}{q_k}\right) \quad \text{where} \quad q_0 = \frac{d+1}{d-1}, \quad q_{k+1} = 2\,q_k^2 - 1 \tag{34.2-34}$$

(convergence is quadratic) and

$$\sqrt{d} \quad = \quad \prod_{k=0}^{\infty}\left(1+\frac{2}{h_k}\right) \quad \text{where} \quad h_0 = \frac{d+3}{d-1}, \quad h_{k+1} = (h_k+2)^2\,(h_k-1)+1 \tag{34.2-35}$$

(convergence is cubic). These are given in [29] (also in [110], more expressions can be found in [103]). The paper gives $h_{k+1} = \frac{4\,d}{d-1}\prod_{i=0}^{k}\left(h_i^2\right) - 3$. Note that for relation 34.2-34 we have

$$q_k \quad = \quad T_{2^k}(q_0) \tag{34.2-36}$$

$$\frac{1}{q_k} \quad = \quad \frac{(d-1)^N}{\sum_{i=0}^{N}\binom{2\,N}{2\,i}\,d^i} \;=\; \frac{2\,(1-d)^N}{(1+\sqrt{d})^{2N}+(1-\sqrt{d})^{2N}} \quad \text{where} \quad N = 2^k \tag{34.2-37}$$

where T_n is the n-th Chebyshev polynomial of the first kind. One finds

$$q_k \quad = \quad T_{2^k}(1/c) \quad \text{where} \quad c = \frac{1-d}{1+d}, \quad c < 1 \tag{34.2-38}$$

and

$$\sqrt{\frac{1-c}{1+c}} \quad \approx \quad \frac{1-c}{c}\,\frac{U_{2^k-1}(1/c)}{T_{2^k}(1/c)} \tag{34.2-39}$$

which can be expressed in $d = \frac{1-c}{1+c}$ as

$$\sqrt{d} \quad \approx \quad \frac{2\,d}{1-d}\,\frac{U_{2^k-1}(\frac{1+d}{1-d})}{T_{2^k}(\frac{1+d}{1-d})} \quad \text{where} \quad d > 1 \tag{34.2-40}$$

where U_n is the n-th Chebyshev polynomial of the second kind. We have $U_{2^k-1}(x) = 2^k \prod_{i=0}^{k-1} T_{2^i}(x)$. Successively compute $T_{2^i} = 2\,T_{2^{i-1}}^2 - 1$ and accumulate the product $U_{2^i-1} = 2\,U_{2^{i-1}-1}\,T_{2^{i-1}}$ until U_{2^k-1} and T_{2^k} are obtained. Alternatively use the relation $U_k(x) = \frac{1}{k+1}\,\partial_x T_{k+1}(x)$ and use the recursion for the coefficients of T as shown on page 651.

A systematic approach to find product expressions for roots is given in section 28.7 on page 558.

Chapter 35

Cyclotomic polynomials, Hypergeometric functions, and continued fractions

We describe the cyclotomic polynomials and some of their properties, together with the Möbius inversion principle. We also give algorithms to convert a power series into Lambert series and infinite products.

We describe the hypergeometric functions which contain most of the 'useful' functions such as the logarithm and the sine as special cases. The transformation formulas for hypergeometric functions can be used to obtain series transformations that are non-obvious. The computation of certain hypergeometric functions by AGM-type algorithms is described in section 30.3 on page 578.

Further continued fractions are described together with algorithms for their computation.

35.1 Cylotomic polynomials, Möbius inversion, Lambert series

35.1.1 Cyclotomic polynomials

The roots (over $\mathbb{C}$) of the polynomial $x^n - 1$ are the n-th roots of unity:

$$x^n - 1 \quad = \quad \prod_{k=0}^{n-1} \left(x - \exp\left(\frac{2\,i\,\pi\,k}{n} \right) \right) \tag{35.1-1}$$

The n-th *cyclotomic polynomial* Y_n can be defined as the polynomial whose roots are the primitive n-th roots of unity:

$$Y_n(x) \quad := \quad \prod_{\substack{k=0..n-1 \\ \gcd(k,n)=1}} \left(x - \exp\left(\frac{2\,i\,\pi\,k}{n} \right) \right) \tag{35.1-2}$$

The degree of Y_n equals the number of primitive n-th roots, that is

$$\deg(Y_n) \quad = \quad \varphi(n) \tag{35.1-3}$$

The coefficients are integers, for example,

$$Y_{63}(x) \quad = \quad x^{36} - x^{33} + x^{27} - x^{24} + x^{18} - x^{12} + x^9 - x^3 + 1 \tag{35.1-4}$$

```
  n:   Yn(x)
  1:   x - 1
  2:   x + 1
  3:   x^2 + x + 1
  4:   x^2 + 1
  5:   x^4 + x^3 + x^2 + x + 1
  6:   x^2 - x + 1
  7:   x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
  8:   x^4 + 1
  9:   x^6 + x^3 + 1
 10:   x^4 - x^3 + x^2 - x + 1
 11:   x^10 + ... + 1  <--= all coefficients are one for prime n
 12:   x^4 - x^2 + 1
 13:   x^12 + ... + 1
 14:   x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
 15:   x^8 - x^7 + x^5 - x^4 + x^3 - x + 1
 16:   x^8 + 1
 17:   x^16 + ... + 1
 18:   x^6 - x^3 + 1
 19:   x^18 + ... + 1
 20:   x^8 - x^6 + x^4 - x^2 + 1
 21:   x^12 - x^11 + x^9 - x^8 + x^6 - x^4 + x^3 - x + 1
 22:   x^10 - x^9 + x^8 - x^7 + x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
 23:   x^22 + ... + 1
 24:   x^8 - x^4 + 1
 25:   x^20 + x^15 + x^10 + x^5 + 1
 26:   x^12 - x^11 + x^10 - x^9 + x^8 - x^7 + x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
 27:   x^18 + x^9 + 1
 28:   x^12 - x^10 + x^8 - x^6 + x^4 - x^2 + 1
 29:   x^28 + ... + 1
 30:   x^8 + x^7 - x^5 - x^4 - x^3 + x + 1
```

Figure 35.1-A: The first 30 cyclotomic polynomials.

The first 30 cyclotomic polynomials are shown in figure 35.1-A. The first cyclotomic polynomial with a coefficient not in the set $\{0, \pm 1\}$ is Y_{105}:

$$Y_{105}(x) \quad = \quad x^{48} + x^{47} + x^{46} - x^{43} - x^{42} - 2 \cdot x^{41} - x^{40} - x^{39} + \ldots \tag{35.1-5}$$

The cyclotomic polynomials are irreducible over $\mathbb{Z}$. All except Y_1 are self-reciprocal.

For n prime the cyclotomic polynomial $Y_n(x)$ equals $(x^n - 1)/(x - 1) = x^{n-1} + x^{n-2} + \ldots + x + 1$. For $n = 2\,k$ and odd $k \geq 3$ we have $Y_n(x) = Y_k(-x)$. For $n = p\,k$ where p is a prime that does not divide k we have $Y_n(x) = Y_k(x^p)/Y_p(x)$. The following algorithm for the computation of $Y_n(x)$ is given in [119, p.403]:

1. Let $[p_1, p_2, \ldots, p_r]$ the distinct prime divisors of n. Set $y_0(x) = x - 1$.

2. For $j = 1, 2, \ldots, r$ set $y_j(x) = y_j(x^{p_j})/y_j(x)$ (the division is exact).

3. Return $y_r(x^{n/(p_1\,p_2\,\cdots\,p_r)})$

The last statement uses the fact that for $n = k\,t$ where all prime factors of k divide t we have $Y_n(x) = Y_t(x^k)$. An implementation is

```
polcyclo2(n, z='x)=
{
    local(fc, y);
    fc = factor(n)[,1];  \\ prime divisors
    y = z - 1;
    for (j=1, #fc,  y=subst(y,z,z^fc[j])\y;  n\=fc[j]; );
    y = subst(y, z, z^n);
    return( y );
}
```

Note that the routine will only work when the argument z is a symbol.

35.1.2 The Möbius inversion principle

The *Möbius function* $\mu(n)$ is defined for positive integer arguments n as

$$\mu(n) \quad := \quad \begin{cases} 0 & \text{if} \quad n \quad \text{has a square factor} \\ (-1)^k & \text{if} \quad n \quad \text{is a product of } k \text{ distinct primes} \\ +1 & \text{if} \quad n = 1 \end{cases} \tag{35.1-6}$$

The function satisfies

$$\sum_{d \backslash n} \mu(d) \quad = \quad \begin{cases} 1 & \text{if} \quad n = 0 \\ 0 & \text{else} \end{cases} \tag{35.1-7}$$

A function $f(n)$ defined that satisfies

$$f(n \cdot m) \quad = \quad f(n) \cdot f(n) \quad \text{if} \quad \gcd(n, m) = 1 \tag{35.1-8}$$

is said to be *multiplicative*. For a multiplicative function one always has $f(1) = 1$ and $f(n) = f(p_1^{e_1}) \cdot f(p_2^{e_2}) \cdot \ldots \cdot f(p_k^{e_k})$ where $n = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$ is the factorization of n into distinct primes p_i. If the equality holds also for $\gcd(n, m) \neq 1$ the function is said to be *completely multiplicative*. Such a function satisfies $f(n) = f(p_1)^{e_1} \cdot f(p_2)^{e_2} \cdot \ldots \cdot f(p_k)^{e_k}$.

The Möbius function is multiplicative

$$\mu(n)\ \mu(m) \quad = \quad \begin{cases} \mu(n \cdot m) & \text{if} \quad \gcd(n, m) = 1 \\ 0 & \text{else} \end{cases} \tag{35.1-9}$$

For the cyclotomic polynomials one has

$$x^n - 1 \quad = \quad \prod_{d \backslash n} Y_d(x) \tag{35.1-10}$$

and

$$Y_n(x) \quad = \quad \prod_{d \backslash n} \left(x^d - 1\right)^{\mu(n/d)} \tag{35.1-11}$$

The relation implies a reasonably efficient algorithm for the computation of the cyclotomic polynomials. The method also works when the argument x is not a symbol.

n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$	n :$\mu(n)$
1: +1	11: −1	21: +1	31: −1	41: −1	51: +1	61: −1	71: −1
2: −1	12: 0	22: +1	32: 0	42: −1	52: 0	62: +1	72: 0
3: −1	13: −1	23: −1	33: +1	43: −1	53: −1	63: 0	73: −1
4: 0	14: +1	24: 0	34: +1	44: 0	54: 0	64: 0	74: 1
5: −1	15: +1	25: 0	35: +1	45: 0	55: +1	65: +1	75: 0
6: +1	16: 0	26: +1	36: 0	46: +1	56: 0	66: −1	76: 0
7: −1	17: −1	27: 0	37: −1	47: −1	57: +1	67: −1	77: 1
8: 0	18: 0	28: 0	38: +1	48: 0	58: +1	68: 0	78: −1
9: 0	19: −1	29: −1	39: +1	49: 0	59: −1	69: +1	79: −1
10: +1	20: 0	30: −1	40: 0	50: 0	60: 0	70: −1	80: 0

Figure 35.1-B: Values of the Möbius function $\mu(n)$ for $n \leq 80$.

The pair of relations 35.1-10 and 35.1-11 is actually a special case of the (multiplicative) *Möbius inversion principle*:

$$g(n) \ = \ \prod_{d \backslash n} f(d) \quad \Longleftrightarrow \quad f(n) \ = \ \sum_{d \backslash n} g\,(d)^{\mu(n/d)} \tag{35.1-12}$$

Relation 35.1-10 implies (considering the polynomial degrees only and using relation 35.1-3)

$$n \;=\; \sum_{d\backslash n} \varphi(d) \tag{35.1-13}$$

while relation 35.1-11 corresponds to the equality

$$\varphi(n) \;=\; \sum_{d\backslash n} d\;\mu(n/d) \tag{35.1-14}$$

Relations 35.1-13 and 35.1-14 are a special case of the additive version of the Möbius inversion principle:

$$g(n) \;=\; \sum_{d\backslash n} f(d) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_{d\backslash n} g(d)\;\mu(n/d) \tag{35.1-15}$$

More general, if $h(a\,b) = h(a)\,h(b)$ (see [97, p.447]) then

$$g(n) \;=\; \sum_{d\backslash n} f(d)\,h(n/d) \quad\Longleftrightarrow\quad f(n) \;=\; \sum_{d\backslash n} g(d)\,h(n/d)\;\mu(n/d) \tag{35.1-16}$$

Setting $h(n) = 1$ gives relation 35.1-15. The Möbius inversion principle is nicely explained in [124]. The sequence of the values of the Möbius function (see figure 35.1-B) is entry A008683 of [214].

We note two relations valid for multiplicative functions f:

$$\sum_{d\backslash n} \mu(d)\,f(d) \;=\; \prod_{d\backslash n,\; d \text{ prime}} (1 - f(d)) \tag{35.1-17a}$$

$$\sum_{d\backslash n} \mu(d)^2\,f(d) \;=\; \prod_{d\backslash n,\; d \text{ prime}} (1 + f(d)) \tag{35.1-17b}$$

Relation 35.1-17a with $f(n) = 1/n$ gives relation 35.1-13 and also

$$\varphi(n) \;=\; n \prod_{d\backslash n,\; d \text{ prime}} \left(1 - \frac{1}{d}\right) \tag{35.1-18}$$

35.1.3 Lambert series

A *Lambert series* is a series of the form

$$L(x) \;=\; \sum_{k>0} \frac{a_k\,x^k}{1 - x^k} \;=\; \sum_{k>0}\sum_{j>0} a_k\,x^{kj} \tag{35.1-19}$$

It can be converted to a Taylor series

$$L(x) \;=\; \sum_{k>0} b_k\,x^k \quad \text{where} \quad b_k = \sum_{d\backslash k} a_d \tag{35.1-20}$$

The inversion principle allows us to transform a Taylor series to a Lambert series:

$$a_k = \sum_{d\backslash k} b_d\;\mu(k/d) \tag{35.1-21}$$

With pari/gp the conversion to a Lambert series can be implemented as

```
ser2lambert(t)=
{
/* Let t=[a1,a2,a3, ...], n=length(v), where t(x)=sum_{k=1}^{n}{a_k*x^k};
 * Return  L=[l1,l2,l3,...] so that (up to order n)
 * t(x)=\sum_{j=1}^{n}{l_j*x^j/(1-x^j)}
 */
    local(n, L);
    n = length(t);
    L = vector(n);
    for (k=1, n,  fordiv(k, d, L[k]+=moebius(k/d)*t[d]); );
    return( L );
}
```

The conversion in the other direction is

```
lambert2ser(L)=
{ /* inverse of ser2lambert() */
    local(n, t);
    n = length(L);
    t = sum(k=1, length(L), O('x^(n+1))+L[k]*'x^k/(1-'x^k) );
    t = Vec(t);
    return( t );
}
```

We note a relation that is a useful for the computation of the sum, it is given in [156, p.644, ex.27],

$$L(x) \quad = \quad \sum_{k>0} x^{k^2} \left[a_k + \sum_{j>0} (a_k + a_{k+j})\, x^{k\,j} \right] \tag{35.1-22}$$

The special case $a_k = 1$ is given as relation 37.3-2 on page 739. For the related series

$$P(x) \quad = \quad \sum_{k>0} \frac{a_k\, x^k}{1+x^k} \;=\; -\sum_{k>0} \sum_{j>0} (-1)^j\, a_k\, x^{k\,j} \tag{35.1-23}$$

we find

$$P(x) \quad = \quad L(x) - 2\,L(x^2) \tag{35.1-24}$$

To verify the relation compute the k-th term on both sides: $a_k\, x^k/(1+x^k) = a_k\, x^k/(1-x^k) - 2\, a_k\, x^{2k}/(1-x^{2k})$. The other direction is obtained by repeatedly using $L(x) = P(x) + 2\,L(x^2)$:

$$L(x) \quad = \quad \sum_{k=0}^{\infty} 2^k\, P(x^{2^k}) \tag{35.1-25}$$

Use relations 35.1-22 and 35.1-24 to obtain

$$P(x) \quad = \quad \sum_{k>0} x^{k^2} \left[a_k\,(1 - 2\,x^{k^2}) + \sum_{j>0} (a_k + a_{k+j}) \left(x^{k\,j} - 2\,x^{(k+j)^2 - j^2} \right) \right] \tag{35.1-26}$$

35.1.4 Conversion of series to infinite products

Given a series with constant term one,

$$f(x) \quad = \quad 1 + \sum_{k>0} a_k\, x^k \tag{35.1-27}$$

we want to find an infinite product such that

$$f(x) \quad = \quad \prod_{k>0} (1 - x^k)^{b_k} \tag{35.1-28}$$

We take the logarithm, differentiate, and multiply by x:

$$x\,\frac{f'(x)}{f(x)} \quad = \quad \sum_{k>0} \frac{(-k\,b_k)\,x^k}{1-x^k} \tag{35.1-29}$$

The expression on the right hand side is a Lambert series with coefficients $-k\,b_k$, the expression on the left is easily computable as a power series, and we know how to compute a Lambert series from a power series. Thereby

$$b_k \quad = \quad -\frac{1}{k}\sum_{d\backslash k} q_k\;\mu(k/d) \tag{35.1-30}$$

where the q_k are the coefficients of the power series for $q(x) := x\,f'(x)/f(x)$. With pari/gp the conversion to a product can be implemented as

```
ser2prod(t)=
{
/* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k};
 * Return  p=[p1,p2,p3,...] so that (up to order n)
 * t(x)=\prod_{j=1}^{n}{(1-x^j)^{p_j}}
 */
    local(v);
    v = Ser(t);
    v = v'/v;
    v = vector(#t-1, j, polcoeff(v, j-1));
    v = ser2lambert(v);
    v = vector(#v, j, -v[j]/j);
    return( v );
}
```

A simple example is $f(x) = \exp(x)$, so $x\,f'/f = x$, and

$$\exp(x) \quad = \quad \prod_{k>0} (1-x^k)^{-\,\mu(k)/k} \quad = \quad \frac{(1-x^2)^{1/2}\,(1-x^3)^{1/3}\,(1-x^5)^{1/5}\,\ldots}{(1-x^1)^{1/1}\,(1-x^6)^{1/6}\,(1-x^{10})^{1/10}\,\ldots} \tag{35.1-31}$$

Taking the logarithm, we obtain

$$x \quad = \quad -\sum_{k>0} \frac{\mu(k)}{k}\,\log(1-x^k) \tag{35.1-32}$$

Setting $f(x) = 1-2\,x$ we obtain relation 17.2-6a on page 344 (number of binary Lyndon words):

```
? ser2prod(Vec(1-2*x+O(x^20)))
  [2, 1, 2, 3, 6, 9, 18, 30, 56, 99, 186, 335, 630, 1161, 2182, 4080, 7710, 14532, 27594]
```

Setting $f(x) = 1-x-x^2$ gives the number of binary Lyndon words without the subsequence 00 (entry A006206 of [214]):

```
? ser2prod(Vec(1-x-x^2+O(x^20)))
  [1, 1, 1, 1, 2, 2, 4, 5, 8, 11, 18, 25, 40, 58, 90, 135, 210, 316, 492]
```

The ordinary generating function for the e_k corresponding to the product form $f(x) = \prod (1-x^k)^{e_k}$ is

$$\sum_{k=1}^{\infty} e_k\,x^k \quad = \quad -\sum_{k=1}^{\infty} \frac{\mu(k)}{k}\,\log\big(f(x^k)\big) \tag{35.1-33}$$

This can be seen by using the product form for f on the right hand side, using the power series $\log(1-x) = -(x+x^2/2+x^3/3+\ldots)$, and using the defining property of the Möbius function (relation 35.1-7 on page 657). An example is relation 17.2-6b on page 344. For the cyclotomic polynomials we obtain (via relation 35.1-11 on page 657):

$$-\sum_{k=1}^{\infty} \frac{\mu(k)}{k}\,\log\big(Y_n(x^k)\big) \quad = \quad \sum_{d\backslash n} \mu(d)\,x^{n/d} \tag{35.1-34}$$

For example, setting $n = 2$ we obtain

$$x^2 - x \quad = \quad -\sum_{k=1}^{\infty} \frac{\mu(k)}{k} \log\left(1 + x^k\right) \tag{35.1-35}$$

35.1.4.1 An alternative product form

For the transformation into products of the form $\prod (1 + x^k)^{c_k}$ we set

$$f(x) \quad = \quad \prod_{k>0} (1 + x^k)^{c_k} \tag{35.1-36}$$

and note that

$$x\,\frac{f'(x)}{f(x)} \quad = \quad \sum_{k>0} \frac{(+k\,c_k)\,x^k}{1 + x^k} \tag{35.1-37}$$

So we need a transformation into series of this type. As the Möbius transform is not (easily) applicable we use a greedy algorithm:

```
ser2lambertplus(t)=
{
/* Let t=[a1,a2,a3, ...], n=length(v), where t(x)=sum_{k=1}^{n}{a_k*x^k};
 * Return  L=[l1,l2,l3,...] so that (up to order n)
 * t(x)=\sum_{j=1}^{n}{l_j*x^j/(1+x^j)}
 */
    local(n, L, k4);
    n = length(t);
    L = vector(n);
    for (k=1, n,
        tk = t[k];
        L[k] = tk;
        \\ subtract tk * x^k/(1+x^k):
        forstep(j=k, n, 2*k, t[j] -= tk);
        forstep(j=k+k, n, 2*k, t[j] += tk);
    );
    return( L );
}
```

Now we can compute the product form via

```
ser2prodplus(t)=
{
/* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k};
 * Return  p=[p1,p2,p3,...] so that (up to order n)
 * t(x)=\prod_{j=1}^{n}{(1+x^j)^{p_j}}
 */
    local(v);
    v = Ser(t);
    v = v'/v;
    v = vector(#t-1, j, polcoeff(v, j-1));
    v = ser2lambertplus(v);
    v = vector(#v, j, v[j]/j);
    return( v );
}
```

A product $\prod_{k>0} (1 - x^k)^{b_k}$ can be converted into a product $\prod_{k>0} (1 + x^k)^{c_k}$ via the relation $(1 - x) = \prod_{k\geq 0} (1 + x^{2^k})^{-1}$.

35.1.4.2 Conversion to eta-products

The conversion of a series to a product of the form (eta-product)

$$\prod_{k=1}^{\infty} \left[\eta(x^k)\right]^{u_k} \quad \text{where} \quad \eta(x) := \prod_{j=1}^{\infty} 1 - x^j \tag{35.1-38}$$

```
   r= 2:  ( E(y^2)^3 ) / ( E(y^4) )
   r= 3:  ( E(y^3)^4 ) / ( E(y^9) )
   r= 4:  ( E(y^4)^7 ) / ( E(y^8)^3 )
   r= 5:  ( E(y^5)^6 ) / ( E(y^25) )
   r= 6:  ( E(y^6)^12 E(y^36) ) / ( E(y^12)^4 E(y^18)^3 )
   r= 7:  ( E(y^7)^8 ) / ( E(y^49) )
   r= 8:  ( E(y^8)^15 ) / ( E(y^16)^7 )
   r= 9:  ( E(y^9)^13 ) / ( E(y^27)^4 )
   r=10:  ( E(y^10)^18 E(y^100) ) / ( E(y^20)^6 E(y^50)^3 )
   r=11:  ( E(y^11)^12 ) / ( E(y^121) )
   r=12:  ( E(y^12)^28 E(y^72)^3 ) / ( E(y^24)^12 E(y^36)^7 )
   r=13:  ( E(y^13)^14 ) / ( E(y^169) )
   r=14:  ( E(y^14)^24 E(y^196) ) / ( E(y^28)^8 E(y^98)^3 )
   r=15:  ( E(y^15)^24 E(y^225) ) / ( E(y^45)^6 E(y^75)^4 )
   r=16:  ( E(y^16)^31 ) / ( E(y^32)^15 )
   r=17:  ( E(y^17)^18 ) / ( E(y^289) )
   r=18:  ( E(y^18)^39 E(y^108)^4 ) / ( E(y^36)^13 E(y^54)^12 )
   r=19:  ( E(y^19)^20 ) / ( E(y^361) )
   r=20:  ( E(y^20)^42 E(y^200)^3 ) / ( E(y^40)^18 E(y^100)^7 )
   r=21:  ( E(y^21)^32 E(y^441) ) / ( E(y^63)^8 E(y^147)^4 )
   r=22:  ( E(y^22)^36 E(y^484) ) / ( E(y^44)^12 E(y^242)^3 )
   r=23:  ( E(y^23)^24 ) / ( E(y^529) )
   r=24:  ( E(y^24)^60 E(y^144)^7 ) / ( E(y^48)^28 E(y^72)^15 )
   r=25:  ( E(y^25)^31 ) / ( E(y^125)^6 )
   r=26:  ( E(y^26)^42 E(y^676) ) / ( E(y^52)^14 E(y^338)^3 )
   r=27:  ( E(y^27)^40 ) / ( E(y^81)^13 )
   r=28:  ( E(y^28)^56 E(y^392)^3 ) / ( E(y^56)^24 E(y^196)^7 )
   r=29:  ( E(y^29)^30 ) / ( E(y^841) )
   r=30:  ( E(y^30)^72 E(y^180)^6 E(y^300)^4 E(y^450)^3 ) /
            ( E(y^60)^24 E(y^90)^18 E(y^150)^12 E(y^900) )
   r=31:  ( E(y^31)^32 ) / ( E(y^961) )
   r=32:  ( E(y^32)^63 ) / ( E(y^64)^31 )
   r=33:  ( E(y^33)^48 E(y^1089) ) / ( E(y^99)^12 E(y^363)^4 )
```

Figure 35.1-C: Functions $\eta_r(y) := \prod_{j=0}^{r-1} \eta(\omega^j\, y)$ as products of η-functions.

can be done by a greedy algorithm:

```
etaprod(v)=
{
/* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k};
 * Return  p=[p1,p2,p3,...] so that (up to order n)
 * t(x)=\prod_{j=1}^{n}{eta(x^j)^{p_j}}
 * where eta(x) = prod(k>0, (1-x^k))
 */
    local(n, t);
    v = ser2prod(v);
    n = length(v);
    for (k=1, n,
        t = v[k];
        forstep (j=k+k, n, k, v[j]-=t; );
    );
    return( v );
}
```

Similarly, to convert into a product of the form

$$\prod_{k=1}^{\infty} \left[\eta_+(x^k)\right]^{u_k} \quad \text{where} \quad \eta_+(x) := \prod_{j=1}^{\infty} 1 + x^j \tag{35.1-39}$$

use

```
etaprodplus(v)=
{
/* Let t=[1,a1,a2,a3, ...], n=length(v), where t(x)=1+sum_{k=1}^{n}{a_k*x^k};
 * Return  p=[p1,p2,p3,...] so that (up to order n)
 * t(x)=\prod_{j=1}^{n}{eta_+(x^j)^{p_j}}
 * where eta_+(x) = prod(k>0, (1+x^k))
```

```
 */
    local(n, t);
    v = ser2prodplus(v);
    n = length(v);
    for (k=1, n,
        t = v[k];
        forstep (j=k+k, n, k, v[j]-=t; );
    );
    return( v );
}
```

The routines are useful for computations with the generating functions of partitions of certain types, see section 14.4 on page 316. Here we just give:

$$\eta(-x) = \frac{\eta(x^2)^3}{\eta(x)\,\eta(x^4)} \tag{35.1-40a}$$

$$\eta(+i\,x)\,\eta(-i\,x) = \frac{\eta(x^4)^8}{\eta(x^2)^3\,\eta(x^8)^3} \tag{35.1-40b}$$

Figure 35.1-C gives more product formulas.

35.2 Hypergeometric functions

The *hypergeometric function* $F\left(\begin{smallmatrix}a,\,b\\c\end{smallmatrix}\middle|\,z\right)$ can be defined as

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\middle|\,z\right) := \sum_{k=0}^{\infty} \frac{a^{\overline{k}}\,b^{\overline{k}}}{c^{\overline{k}}}\,\frac{z^k}{k!} \tag{35.2-1}$$

where $z^{\overline{k}} := z\,(z+1)\,(z+2)\,\ldots\,(z+k-1)$ is the *rising factorial power* ($z^{\overline{0}} := 1$). Some sources use the so-called *Pochhammer symbol* $(x)_k$ which is the same: $(x)_k = x^{\overline{k}}$. We'll stick to the factorial notation.

z is the *argument* of the function, a, b and c are the *parameters*. Parameters in the upper and lower row are called upper and lower parameters, respectively.

Note the $k! = 1^{\overline{k}}$ in the denominator of relation 35.2-1. You might want to have the hidden lower parameter 1 in mind:

$$F\left(\begin{matrix}2,\,2\\1\end{matrix}\middle|\,z\right) = \text{“}F\left(\begin{matrix}2,\,2\\1,\,1\end{matrix}\middle|\,z\right)\text{”} \tag{35.2-2}$$

The expression is a sum of perfect squares if z is a square.

We have

$$F\left(\begin{matrix}a,\,b\\c\end{matrix}\middle|\,z\right) = 1 + \frac{a}{1}\,\frac{b}{c}\,z\,\left(1 + \frac{a+1}{2}\,\frac{b+1}{c+1}\,z\,\left(1 + \frac{a+2}{3}\,\frac{b+2}{c+2}\,z\,(1+\ldots)\right)\right) \tag{35.2-3}$$

so by formula 35.2-3 hypergeometric functions with rational arguments can be computed with the binary splitting method described in section 32.1.

Hypergeometric functions can have any number of parameters:

$$F\left(\begin{matrix}a_1,\,\ldots,\,a_m\\b_1,\,\ldots,\,b_n\end{matrix}\middle|\,z\right) = \sum_{k=0}^{\infty} \frac{a_1^{\overline{k}}\ldots a_m^{\overline{k}}}{b_1^{\overline{k}}\ldots b_n^{\overline{k}}}\,\frac{z^k}{k!} \tag{35.2-4}$$

These are sometimes called *generalized hypergeometric functions*. The number of upper and lower parameters are often emphasized as subscripts left and right to the symbol F. For example, ${}_mF_n$ for the hypergeometric function in the last relation.

The functions $F\left(\begin{matrix}a\\b\end{matrix}\middle| z\right)$ (of type ${}_1F_1$) are sometimes written as $M(a,b,z)$ or $\Phi(a;b;z)$. *Kummer's function* $U(a,b,z)$ (or $\Psi(a;b;z)$) is related to hypergeometric functions of type ${}_2F_0$:

$$U(a,b,z) \quad = \quad z^{-a}\, F\left(\begin{matrix}a,\, 1+a-b\\ \end{matrix}\middle| -1/z\right) \tag{35.2-5}$$

Note that series ${}_2F_0$ are not convergent. Still, they can be used as asymptotic series for large values of z.

The so-called *Whittaker functions* are related to hypergeometric functions as follows:

$$M_{a,b}(z) \quad = \quad e^{-z/2}\, z^{b+1/2}\, F\left(\begin{matrix}\frac{1}{2}+b-a\\ 1+2b\end{matrix}\middle| z\right) \tag{35.2-6a}$$

$$W_{a,b}(z) \quad = \quad e^{-z/2}\, z^{b+1/2}\, U\left(\frac{1}{2}+b-a,\, 1+2b,\, z\right) \tag{35.2-6b}$$

$$\quad = \quad e^{-z/2}\, z^{a}\, F\left(\begin{matrix}\frac{1}{2}+b-a,\, \frac{1}{2}-b-a\\ \end{matrix}\middle| -1/z\right) \tag{35.2-6c}$$

Negative integer parameters in the upper row lead to polynomials:

$$F\left(\begin{matrix}-3,\, 3\\ 1\end{matrix}\middle| z\right) \quad = \quad 1-9\,z+18\,z^2-10\,z^3 \tag{35.2-7}$$

The lower parameter must not be zero or a negative integer unless there is a negative upper parameter with smaller absolute value.

Sometimes one finds the notational convention to omit an argument $z=1$:

$$F\left(\begin{matrix}a_1,\,\ldots,\,a_m\\ b_1,\,\ldots,\,b_n\end{matrix}\right) \quad := \quad F\left(\begin{matrix}a_1,\,\ldots,\,a_m\\ b_1,\,\ldots,\,b_n\end{matrix}\middle| 1\right) \tag{35.2-8}$$

In what follows the argument is never omitted.

An in-depth treatment of hypergeometric functions is [15].

35.2.1 Derivative and differential equation

Using the relation

$$\frac{d}{dz^n}\, F\left(\begin{matrix}a,\, b\,\ldots\\ c,\,\ldots\end{matrix}\middle| z\right) \quad = \quad \frac{a^{\overline{n}}\, b^{\overline{n}}\,\ldots}{c^{\overline{n}}\,\ldots}\, F\left(\begin{matrix}a+n,\, b+n\,\ldots\\ c+n,\,\ldots\end{matrix}\middle| z\right) \tag{35.2-9}$$

one can verify that $f(z)=F\left(\begin{matrix}a,b\\c\end{matrix}\middle| z\right)$ is a solution of the differential equation

$$z\,(1-z)\,\frac{d^2 f}{dz^2}+[c-(1+a+b)\,z]\,\frac{d\,f}{dz}-a\,b\,f \quad = \quad 0 \tag{35.2-10}$$

A general form of the differential equation satisfied by $F\left(\begin{matrix}a,b,c,\ldots\\u,v,w,\ldots\end{matrix}\middle| z\right)$ is

$$z\,(\vartheta+a)\,(\vartheta+b)\,(\vartheta+c)\,\ldots\,f(z) \quad = \quad \vartheta\,(\vartheta+u-1)\,(\vartheta+v-1)\,(\vartheta+w-1)\,\ldots\,f(z) \tag{35.2-11}$$

where ϑ is the operator $z\,\frac{d}{dz}$. The leftmost ϑ on the right hand side of the equation takes care of the hidden lower parameter 1: $\vartheta=(\vartheta+1-1)$. See [124] for a beautiful derivation. Relation 10.15-4a on page 268 can be used to rewrite powers of ϑ as polynomials in $\frac{d}{dz}$.

35.2.2 Evaluations for fixed z

A closed form (in terms of the gamma function) evaluation at $z = 1$ can be given for ${}_2F_1$:

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, 1\right) \quad = \quad \frac{\Gamma(c)\,\Gamma(c-a-b)}{\Gamma(c-a)\,\Gamma(c-b)} \qquad \text{if} \quad \mathfrak{Re}(c-a-b) > 0 \quad \text{or } b \in \mathbb{N}, b < 0 \tag{35.2-12}$$

When $c - a - b < 0$ then [245, ex.18, p.299]

$$\lim_{z\to 1-} F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right) / \left(\frac{\Gamma(c)\,\Gamma(a+b-c)}{\Gamma(a)\,\Gamma(b)}\,(1-z)^{c-a-b}\right) \quad = \quad 1 \tag{35.2-13a}$$

and, for $c - a - b = 0$,

$$\lim_{z\to 1-} F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right) / \left(\frac{\Gamma(a+b)}{\Gamma(a)\,\Gamma(b)}\,\log\frac{1}{1-z}\right) \quad = \quad 1 \tag{35.2-13b}$$

For $z = -1$ there is an evaluation due to Kummer:

$$F\left(\begin{matrix}a,\, b\\ 1+a-b\end{matrix}\middle|\, -1\right) \quad = \quad \frac{\Gamma(1-a+b)\,\Gamma(1+a/2)}{\Gamma(1+a)\,\Gamma(1+a/2-b)} \tag{35.2-14a}$$

$$\quad = \quad 2^{-a}\,\pi\,\frac{\Gamma(1-a+b)}{\Gamma(1/2+a/2)\,\Gamma(1+a/2-b)} \tag{35.2-14b}$$

Several evaluations at $z = \frac{1}{2}$ are given in [1], we just give one:

$$F\left(\begin{matrix}a,\, b\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b\end{matrix}\middle|\, \frac{1}{2}\right) \quad = \quad \sqrt{\pi}\,\frac{\Gamma(\frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b)}{\Gamma(\frac{1}{2}+\frac{1}{2}a)\,\Gamma(\frac{1}{2}+\frac{1}{2}b)} \tag{35.2-15}$$

For further information see (chapter 15 of) [1], [240] and [193]. Various evaluations of $F\left(\begin{smallmatrix}-an,\, bn+b_1\\ cn+c_1\end{smallmatrix}\middle|\, z\right)$ for integer a, b, c and n, $1 \le a \le 2$, $-4 \le b \le 4$ and $-4 \le c \le 4$ can be found in [108].

35.2.3 Extraction of even and odd part

Let $E[f(z)] = (f(z)+f(z))/2$ (the even powers of the series of $f(z)$), and $O[f(z)] = (f(z)-f(-z))/2$ (the odd powers). We express the even and odd parts of a hypergeometric series as hypergeometric functions:

$$E\left[F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right)\right] \quad = \quad F\left(\begin{matrix}\frac{a}{2},\, \frac{a+1}{2},\, \frac{b}{2},\, \frac{b+1}{2}\\ \frac{c}{2},\, \frac{c+1}{2},\, \frac{1}{2}\end{matrix}\middle|\, z^2\right) \tag{35.2-16a}$$

$$O\left[F\left(\begin{matrix}a,\, b\\ c\end{matrix}\middle|\, z\right)\right] \quad = \quad \frac{a\,b}{c}\,z\,F\left(\begin{matrix}\frac{a+1}{2},\, \frac{a+2}{2},\, \frac{b+1}{2},\, \frac{b+2}{2}\\ \frac{c+1}{2},\, \frac{c+2}{2},\, \frac{3}{2}\end{matrix}\middle|\, z^2\right) \tag{35.2-16b}$$

The lowers parameters 1/2 and 3/2 are due to the hidden lower parameter 1. The general case for

$$H(z) \quad := \quad F\left(\begin{matrix}a_1,\, \ldots,\, a_m\\ b_1,\, \ldots,\, b_n\end{matrix}\middle|\, z\right) \tag{35.2-17a}$$

is

$$E\left[H(z)\right] \quad = \quad F\left(\begin{matrix}\frac{a_1}{2},\, \frac{a_1+1}{2},\, \ldots,\, \frac{a_m}{2},\, \frac{a_m+1}{2}\\ \frac{b_1}{2},\, \frac{b_1+1}{2},\, \ldots,\, \frac{b_n}{2},\, \frac{b_n+1}{2},\, \frac{1}{2}\end{matrix}\middle|\, X\,z^2\right) \tag{35.2-17b}$$

$$O\left[H(z)\right] \quad = \quad \frac{a_1\,\cdots\,a_m}{b_1\,\cdots\,b_n}\,z\,F\left(\begin{matrix}\frac{a_1+1}{2},\, \frac{a_1+2}{2},\, \ldots,\, \frac{a_m+1}{2},\, \frac{a_m+2}{2}\\ \frac{b_1+1}{2},\, \frac{b_1+2}{2},\, \ldots,\, \frac{b_n+1}{2},\, \frac{b_n+2}{2},\, \frac{3}{2}\end{matrix}\middle|\, X\,z^2\right) \tag{35.2-17c}$$

where $X = 4^{m-n-1}$. For example,

$$E\left[F\left(\begin{matrix}1\\2\end{matrix}\middle| z\right)\right] \;=\; F\left(\begin{matrix}\frac{1}{2},\,1\\1,\,\frac{3}{2},\,\frac{1}{2}\end{matrix}\middle|\,\frac{z^2}{4}\right) \;=\; F\left(\begin{matrix}\\\frac{3}{2}\end{matrix}\middle|\,\frac{z^2}{4}\right) \;=\; \frac{\sinh z}{z} \tag{35.2-18}$$

We indicate a further generalization by the extraction of all terms of $H(z)$ where the exponent of z is divisible by 3:

$$\frac{H(z)+H(\omega\,z)+H(\omega^2\,z)}{3} \;=\; F\left(\begin{matrix}\frac{a_1}{3},\,\frac{a_1+1}{3},\,\frac{a_1+2}{3}\,\ldots,\,\frac{a_m}{3},\,\frac{a_m+1}{3},\,\frac{a_m+2}{3}\\\frac{b_1}{3},\,\frac{b_1+1}{3},\,\frac{b_1+2}{3},\,\ldots,\,\frac{b_n}{3},\,\frac{b_n+1}{3},\,\frac{b_n+2}{3},\,\frac{1}{3},\,\frac{2}{3}\end{matrix}\middle|\,X\,z^3\right) \tag{35.2-19}$$

where $\omega = \exp(2\,i\,\pi/3)$ and $X = 27^{m-n-1}$. For example, with $H(z) = \exp(z) = F\left(\begin{matrix}\\\end{matrix}\middle| z\right)$ we obtain

$$F\left(\begin{matrix}\\\frac{1}{3},\,\frac{2}{3}\end{matrix}\middle|\,\frac{z^3}{27}\right) \;=\; \sum_{k=0}^{\infty}\frac{z^{3k}}{(3k)!} \tag{35.2-20}$$

Define the power series $C_j(z)$, for $j \in \{0,1,2\}$, by

$$C_s(z) \;=\; \sum_{k=0}^{\infty}\frac{z^{3k+s}}{(3k+s)!} \tag{35.2-21a}$$

then (omitting arguments)

$$\det\begin{bmatrix}C_0 & C_1 & C_2\\ C_2 & C_0 & C_1\\ C_1 & C_2 & C_0\end{bmatrix} \;=\; C_0^3 + C_1^3 + C_2^3 - 3\,C_0\,C_1\,C_2 \;=\; 1 \tag{35.2-21b}$$

which is a three power series analogy to the relation $\cosh^2 - \sinh^2 = 1$.

For the extraction of the coefficient at the positions equal to $j \equiv M$ replace every upper and lower parameter A by the M parameters $(A+j)/M$, $(A+j+1)/M$, $(A+j+2)/M$, …, $(A+j+M-1)/M$, and the argument z by $X\,z^M$ where $X = (M^M)^{m-n-1}$.

35.2.4 Transformations

As obvious from the definition, parameters in the upper row can be swapped (capitalized symbols for readability):

$$F\left(\begin{matrix}A,\,B,\,c\\e,\,f,\,g\end{matrix}\middle| z\right) \;=\; F\left(\begin{matrix}B,\,A,\,c\\e,\,f,\,g\end{matrix}\middle| z\right) \tag{35.2-22}$$

The same is true for the lower row. Usually one writes the parameters in ascending order. Identical elements in the lower and upper row can be canceled:

$$F\left(\begin{matrix}a,\,b,\,C\\e,\,f,\,C\end{matrix}\middle| z\right) \;=\; F\left(\begin{matrix}a,\,b\\e,\,f\end{matrix}\middle| z\right) \tag{35.2-23}$$

These trivial transformations are true for any number of elements. The following transformations are only valid for the given structure, unless the list of parameters contain an ellipsis '…'.

35.2.4.1 Elementary relations

$$F\left(\begin{matrix}a,\, b,\, \dots\\ c,\, \dots\end{matrix}\Big|\, z\right) = 1 + z\,\frac{a\,b\,\dots}{c\,\dots}\,F\left(\begin{matrix}a+1,\, b+1,\, \dots,\, 1\\ c+1,\, \dots,\, 2\end{matrix}\Big|\, z\right) \tag{35.2-24}$$

$$(a-b)\,F\left(\begin{matrix}a,\, b,\, \dots\\ c,\, \dots\end{matrix}\Big|\, z\right) = a\,F\left(\begin{matrix}a+1,\, b,\, \dots\\ c,\, \dots\end{matrix}\Big|\, z\right) - b\,F\left(\begin{matrix}a,\, b+1,\, \dots\\ c,\, \dots\end{matrix}\Big|\, z\right) \tag{35.2-25}$$

$$(a-c)\,F\left(\begin{matrix}a,\, b,\, \dots\\ c+1,\, \dots\end{matrix}\Big|\, z\right) = a\,F\left(\begin{matrix}a+1,\, b,\, \dots\\ c+1,\, \dots\end{matrix}\Big|\, z\right) - c\,F\left(\begin{matrix}a,\, b,\, \dots\\ c,\, \dots\end{matrix}\Big|\, z\right) \tag{35.2-26}$$

These are given in [124], the following is taken from [245].

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}a,\, b+1\\ c\end{matrix}\Big|\, z\right) - \frac{a\,z}{c}\,F\left(\begin{matrix}a+1,\, b+1\\ c+1\end{matrix}\Big|\, z\right) \tag{35.2-27}$$

More relations of this type are given in [1].

35.2.4.2 Pfaff's reflection law and Euler's identity

Pfaff's reflection law can be given as either of:

$$\frac{1}{(1-z)^a}\,F\left(\begin{matrix}a,\, b\\ c\end{matrix}\Big|\, \frac{-z}{1-z}\right) = F\left(\begin{matrix}a,\, c-b\\ c\end{matrix}\Big|\, z\right) \tag{35.2-28a}$$

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\Big|\, z\right) = \frac{1}{(1-z)^a}\,F\left(\begin{matrix}a,\, c-b\\ c\end{matrix}\Big|\, \frac{-z}{1-z}\right) \tag{35.2-28b}$$

$$= \frac{1}{(1-z)^b}\,F\left(\begin{matrix}c-a,\, b\\ c\end{matrix}\Big|\, \frac{-z}{1-z}\right) \tag{35.2-28c}$$

Euler's identity is obtained by applying the reflection on both upper parameters:

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\Big|\, z\right) = (1-z)^{(c-a-b)}\,F\left(\begin{matrix}c-a,\, c-b\\ c\end{matrix}\Big|\, z\right) \tag{35.2-29}$$

Euler's transformation can generalized for hypergeometric functions ${}_{r+1}F_r$, see [174]. While Pfaff's transformation cannot be generalized, the following ${}_3F_2$ relation which is reminiscent to the reflection law, is given in [174, p.32]:

$$(1-z)^d\,F\left(\begin{matrix}f-h+2,\, -(f-h+1),\, d\\ f-h+1,\, h\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}f,\, 1+\frac{1}{2}f,\, d\\ \frac{1}{2}f,\, h\end{matrix}\Big|\, \frac{-z}{1-z}\right) \tag{35.2-30}$$

35.2.4.3 A transformation by Gauss

$$F\left(\begin{matrix}2a,\, 2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, 4z(1-z)\right) \quad \text{where} \quad |z| < \frac{1}{2} \tag{35.2-31a}$$

$$F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}2a,\, 2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, \frac{1-\sqrt{1-z}}{2}\right) \tag{35.2-31b}$$

Note that the right hand side of relation 35.2-31a does not change if z is replaced by $1-z$, so it seems that

$$F\left(\begin{matrix}2a,\, 2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}2a,\, 2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, 1-z\right) \tag{35.2-32}$$

However, the relation is true only for terminating series, that is, for polynomials. Rewriting relation 35.2-31a for the argument $\frac{1-z}{2}$ we obtain

$$F\left(\begin{matrix}2a,\,2b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,\frac{1-z}{2}\right) \;=\; F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,1-z^2\right) \tag{35.2-33}$$

35.2.4.4 Whipple's identity and quadratic transformations

Whipple's identity connects two hypergeometric functions ${}_3F_2$:

$$F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2},\,1-a-b-c\\ 1+a-b,\,1+a-c\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \;=\; (1-z)^a\,F\left(\begin{matrix}a,\,b,\,c\\ 1+a-b,\,1+a-c\end{matrix}\Big|\,z\right) \tag{35.2-34}$$

Specializing 35.2-34 for $c=(a+1)/2$ (note the symmetry between b and c so specializing for $c=(b+1)/2$ produces the identical relation) gives

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,z\right) \;=\; \frac{1}{(1-z)^a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{35.2-35}$$

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z\right) \;=\; \left(\frac{2\,(1-\sqrt{1-z})}{z}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{\left(1-\sqrt{1-z}\right)^2}{z}\right) \tag{35.2-36}$$

With $c:=a-b$ in 35.2-35 one obtains:

$$F\left(\begin{matrix}a,\,a-c\\ 1+c\end{matrix}\Big|\,z\right) \;=\; \frac{1}{(1-z)^a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}-\frac{1}{2}a+c\\ 1+c\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{35.2-37}$$

Similarly as for the relations by Gauss, from relations 35.2-35 and 35.2-36:

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,-\frac{1-z}{1+z}\right) \;=\; \left(\frac{1+z}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,1-z^2\right) \tag{35.2-38a}$$

$$F\left(\begin{matrix}a,\,b\\ 1+a-b\end{matrix}\Big|\,-\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) \;=\; \left(\frac{1+\sqrt{1-z^2}}{2}\right)^a F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}-b\\ 1+a-b\end{matrix}\Big|\,z^2\right) \tag{35.2-38b}$$

Relations 35.2-38b and 35.2-38a can be obtained from each other by setting $x=\sqrt{1-y^2}$ (and replacing y by x). The same is true for the next pair of relations:

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,1-z^2\right) \;=\; \left(\frac{2}{1+z}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{1-z}{1+z}\right) \tag{35.2-39a}$$

$$F\left(\begin{matrix}a,\,b\\ a+b+\frac{1}{2}\end{matrix}\Big|\,z^2\right) \;=\; \left(\frac{2}{1+\sqrt{1-z^2}}\right)^{2a} F\left(\begin{matrix}2a,\,a-b+\frac{1}{2}\\ a+b+\frac{1}{2}\end{matrix}\Big|\,-\frac{1-\sqrt{1-z^2}}{1+\sqrt{1-z^2}}\right) \tag{35.2-39b}$$

The transformations

$$F\left(\begin{matrix}a,\,b\\ a-b+1\end{matrix}\Big|\,z\right) \;=\; (1+z)^{-a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a+\frac{1}{2}\\ a-b+1\end{matrix}\Big|\,\frac{4z}{(1+z)^2}\right) \tag{35.2-40a}$$

$$\;=\; (1-z)^{-a}\,F\left(\begin{matrix}\frac{1}{2}a,\,\frac{1}{2}a-b+\frac{1}{2}\\ a-b+1\end{matrix}\Big|\,\frac{-4z}{(1-z)^2}\right) \tag{35.2-40b}$$

$$\;=\; (1\pm\sqrt{z})^{-2a}\,F\left(\begin{matrix}a,\,a-b+\frac{1}{2}\\ 2a-2b+1\end{matrix}\Big|\,\frac{\pm 4\sqrt{z}}{(1\pm\sqrt{z})^2}\right) \tag{35.2-40c}$$

are given in [1]. Specializing for $a = b$ gives

$$F\left(\begin{matrix}a,\, a\\ 1\end{matrix}\Big|\, z\right) = (1+z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}+\frac{1}{2}a\\ 1\end{matrix}\Big|\, \frac{4z}{(1+z)^2}\right) \quad (35.2\text{-}40d)$$

$$= (1-z)^{-a}\, F\left(\begin{matrix}\frac{1}{2}a,\, \frac{1}{2}-\frac{1}{2}a\\ 1\end{matrix}\Big|\, \frac{-4z}{(1-z)^2}\right) \quad (35.2\text{-}40e)$$

$$= (1\pm\sqrt{z})^{-2a}\, F\left(\begin{matrix}a,\, \frac{1}{2}\\ 1\end{matrix}\Big|\, \frac{\pm 4\sqrt{z}}{(1\pm\sqrt{z})^2}\right) \quad (35.2\text{-}40f)$$

Relation 35.2-40e can be obtained by setting $c = 0$ in relation 35.2-35. Observe that the hypergeometric function on the right hand side of relation 35.2-40e does not change when replacing a by $1-a$. The next $({}_3F_2)$ transformation is given in [174]:

$$(1-z)^{-1}\, F\left(\begin{matrix}a,\, b,\, 1\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b,\, 2\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}+\frac{1}{2}a,\, \frac{1}{2}+\frac{1}{2}b,\, 1\\ \frac{1}{2}+\frac{1}{2}a+\frac{1}{2}b,\, 2\end{matrix}\Big|\, 4z(1-z)\right) \quad (35.2\text{-}41)$$

The following are special cases of this transformation:

$$(1-z)^{-1}\, F\left(\begin{matrix}a,\, 1-a\\ 2\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}+\frac{1}{2}a,\, 1-\frac{1}{2}a\\ 2\end{matrix}\Big|\, 4z(1-z)\right) \quad (35.2\text{-}42a)$$

$$(1-z)^{-1}\, F\left(\begin{matrix}a,\, 1\\ 2\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}+\frac{1}{2}a,\, 1+\frac{1}{2}a,\, 1\\ 1+a,\, 2\end{matrix}\Big|\, 4z(1-z)\right) \quad (35.2\text{-}42b)$$

$$(1-z)^{-1}\, F\left(\begin{matrix}a,\, 1\\ \frac{3}{2}+\frac{1}{2}a\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2}+\frac{1}{2}a,\, \frac{3}{2},\, 1\\ \frac{3}{2}+\frac{1}{2}a,\, 2\end{matrix}\Big|\, 4z(1-z)\right) \quad (35.2\text{-}42c)$$

The nonlinear transformation (given in [193, p.21])

$$F\left(\begin{matrix}a,\, b\\ c\end{matrix}\Big|\, z\right) = (1-\omega)^{2a}\sum_{n=0}^{\infty} d_n\,\omega^n \quad (35.2\text{-}43a)$$

where

$$\omega = \frac{-4z}{(1-z)^2}, \qquad z = \frac{\sqrt{1-\omega}-1}{\sqrt{1-\omega}+1} \quad (35.2\text{-}43b)$$

and

$$d_0 = 1 \quad (35.2\text{-}43c)$$

$$d_1 = \frac{2a\,(c-2b)}{c} \quad (35.2\text{-}43d)$$

$$d_{n+2} = \frac{2\,(c-2b)\,(n+1+a)\,d_{n+1} + (n+2a)\,(n+2a+1-c)\,d_n}{(n+2)\,(n+1+c)} \quad (35.2\text{-}43e)$$

maps the complex (z-)plane into the unit circle. Thereby the ω-form of the series converges for all $z \neq 1$.

35.2.4.5 Clausen's product formulas

Clausen's formulas connect hypergeometric functions of type ${}_2F_1$ and ${}_3F_2$:

$$\left[F\left(\begin{matrix}a,\, b\\ a+b+\frac{1}{2}\end{matrix}\Big|\, z\right)\right]^2 = F\left(\begin{matrix}2a,\, a+b,\, 2b\\ a+b+\frac{1}{2},\, 2a+2b\end{matrix}\Big|\, z\right) \quad (35.2\text{-}44a)$$

$$F\left(\begin{matrix}\frac{1}{4}+a,\, \frac{1}{4}+b\\ 1+a+b\end{matrix}\Big|\, z\right) F\left(\begin{matrix}\frac{1}{4}-a,\, \frac{1}{4}-b\\ 1-a-b\end{matrix}\Big|\, z\right) = F\left(\begin{matrix}\frac{1}{2},\, \frac{1}{2}+a-b,\, \frac{1}{2}-a+b\\ 1+a+b,\, 1-a-b\end{matrix}\Big|\, z\right) \quad (35.2\text{-}44b)$$

If $a = b + \frac{1}{2}$ in 35.2-44a then (two parameters on the right hand side cancel)

$$\left[F\left(\begin{matrix} b+\frac{1}{2},\, b \\ 2b+1 \end{matrix}\Big|\, z\right)\right]^2 \;=\; F\left(\begin{matrix} 2b+\frac{1}{2},\, 2b \\ 4b+1 \end{matrix}\Big|\, z\right) \tag{35.2-45}$$

and the right hand side again matches the structure on the left. The corresponding function can be identified (see [52, p.190]) as $G_b(z) := F\left(\begin{smallmatrix} b+\frac{1}{2}, b \\ 2b+1 \end{smallmatrix}\Big|\, z\right) = \left(\frac{1+\sqrt{1-z}}{2}\right)^{-2b}$. One has $G_{n\,m}(z) = [G_n(z)]^m$.

Specializing relation 35.2-44b for $b = -a$ we obtain

$$\left[F\left(\begin{matrix} \frac{1}{4}+a,\, \frac{1}{4}-a \\ 1 \end{matrix}\Big|\, z\right)\right]^2 \;=\; F\left(\begin{matrix} \frac{1}{2}+2a,\, \frac{1}{2},\, \frac{1}{2}-2a \\ 1,\, 1 \end{matrix}\Big|\, z\right) \tag{35.2-46}$$

For $a = 0$, $z = 1$ (or z a sixth power) this relation is an identity between the square of a sum of squares and a sum of cubes:

$$\left[\sum_{n=0}^{\infty}\left[\prod_{j=1}^{n}\frac{1/4+j-1}{j}\right]^2\right]^2 \;=\; \sum_{n=0}^{\infty}\left[\prod_{j=1}^{n}\frac{1/2+j-1}{j}\right]^3 \;=\; 1.39320392968\ldots \tag{35.2-47}$$

The relation can be obtained by setting $\alpha = \beta = 1/4$ and $\gamma = 1/2$ in exercise 16 in [245, p.298]. Setting $a = 1/2$ in exercise 28 in [245, p.301] we find that the quantity equals $\pi/\Gamma(3/4)^4 = \Gamma(1/4)^4/(4\,\pi^3)$. For the square root of the expressions ($\sqrt{1.39320\ldots} = 1.180340\ldots$) we have [109, p.34]:

$$F\left(\begin{matrix} \frac{1}{2},\, \frac{1}{2} \\ 1 \end{matrix}\Big|\, \frac{1}{2}\right) \;=\; \left[\sum_{n=-\infty}^{\infty} e^{-n^2\,\pi}\right]^2 \;=\; 1.180340599016\ldots \tag{35.2-48}$$

We note that relation 35.2-44a on the previous page can be obtained as the special case $c = a + b$ of the following relation given in [245, ex.16, p.298]:

$$F\left(\begin{matrix} a,\, b \\ c+\frac{1}{2} \end{matrix}\Big|\, z\right)\, F\left(\begin{matrix} c-a,\, c-b \\ c+\frac{1}{2} \end{matrix}\Big|\, z\right) \;=\; \sum_{k=0}^{\infty} A_k\, \frac{c^{\overline{k}}}{\left(c+\frac{1}{2}\right)^{\overline{k}}}\, z^k \tag{35.2-49a}$$

where the A_k are defined by

$$(1-z)^{a+b-c}\, F\left(\begin{matrix} 2a,\, 2b \\ 2c \end{matrix}\Big|\, z\right) \;=\; \sum_{k=0}^{\infty} A_k\, z^k \tag{35.2-49b}$$

The following relations are given in [15, p.184]:

$$F\left(\begin{matrix} a,\, b \\ a+b-\frac{1}{2} \end{matrix}\Big|\, z\right)\, F\left(\begin{matrix} a,\, b \\ a+b+\frac{1}{2} \end{matrix}\Big|\, z\right) \;=\; F\left(\begin{matrix} 2a,\, 2b,\, a+b \\ 2a+2b-1,\, a+b+\frac{1}{2} \end{matrix}\Big|\, z\right) \tag{35.2-50a}$$

$$F\left(\begin{matrix} a,\, b \\ a+b-\frac{1}{2} \end{matrix}\Big|\, z\right)\, F\left(\begin{matrix} a,\, b-1 \\ a+b-\frac{1}{2} \end{matrix}\Big|\, z\right) \;=\; F\left(\begin{matrix} 2a,\, 2b-1,\, a+b-1 \\ 2a+2b-2,\, a+b+\frac{1}{2} \end{matrix}\Big|\, z\right) \tag{35.2-50b}$$

35.2.4.6 The Kummer transformation

The Kummer transformation connects two hypergeometric functions of type ${}_1F_1$:

$$\exp(z)\, F\left(\begin{matrix} a \\ a+b \end{matrix}\Big|\, -z\right) \;=\; F\left(\begin{matrix} b \\ a+b \end{matrix}\Big|\, z\right) \tag{35.2-51}$$

The relation is not valid if both a and b are negative integers. In that case one obtains the Padé approximants of $\exp(z)$, see relation 31.2-17 on page 605.

A transformation from ${}_1F_1$ to ${}_2F_3$ is given by

$$F\left(\begin{matrix}a\\b\end{matrix}\middle|\, z\right) F\left(\begin{matrix}a\\b\end{matrix}\middle|\, -z\right) = F\left(\begin{matrix}a,\, b-a\\ b,\, \frac{1}{2}b,\, \frac{1}{2}(b+1)\end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{35.2-52}$$

Setting $b = 2a$ and using 35.2-51 gives

$$\left[F\left(\begin{matrix}a\\2a\end{matrix}\middle|\, z\right)\right]^2 = \exp(z)\, F\left(\begin{matrix}a\\ a+\frac{1}{2},\, 2a\end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{35.2-53}$$

The following transformation connects functions ${}_0F_1$ and ${}_2F_3$:

$$F\left(\begin{matrix}\\a\end{matrix}\middle|\, z\right) F\left(\begin{matrix}\\b\end{matrix}\middle|\, z\right) = F\left(\begin{matrix}\frac{1}{2}(a+b),\, \frac{1}{2}(a+b-1)\\ a,\, b,\, a+b-1\end{matrix}\middle|\, 4z\right) \tag{35.2-54}$$

Setting $b = a$ gives (cancellation of parameters on the right hand side)

$$\left[F\left(\begin{matrix}\\a\end{matrix}\middle|\, z\right)\right]^2 = F\left(\begin{matrix}a-\frac{1}{2}\\ a,\, 2a-1\end{matrix}\middle|\, 4z\right) \tag{35.2-55}$$

From relations 35.2-53 and 35.2-55 one can obtain

$$\exp(z)\, F\left(\begin{matrix}\\a\end{matrix}\middle|\, \frac{z^2}{4}\right) = F\left(\begin{matrix}a-\frac{1}{2}\\ 2a-1\end{matrix}\middle|\, 2z\right) \tag{35.2-56}$$

The following relations can be derived from the preceding ones:

$$\left[F\left(\begin{matrix}\frac{1}{2}(a+b),\, \frac{1}{2}(a+b-1)\\ a,\, b,\, a+b-1\end{matrix}\middle|\, z\right)\right]^2 = F\left(\begin{matrix}a-\frac{1}{2}\\ a,\, 2a-1\end{matrix}\middle|\, z\right) F\left(\begin{matrix}b-\frac{1}{2}\\ b,\, 2b-1\end{matrix}\middle|\, z\right) \tag{35.2-57}$$

$$F\left(\begin{matrix}a\\2a\end{matrix}\middle|\, z\right) F\left(\begin{matrix}a\\2a\end{matrix}\middle|\, -z\right) = \left[F\left(\begin{matrix}\\ a+\frac{1}{2}\end{matrix}\middle|\, \frac{z^2}{16}\right)\right]^2 \tag{35.2-58a}$$

$$F\left(\begin{matrix}a\\2a+1\end{matrix}\middle|\, z\right) F\left(\begin{matrix}a\\2a+1\end{matrix}\middle|\, -z\right) = F\left(\begin{matrix}a\\ 2a+1,\, a+\frac{1}{2}\end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{35.2-58b}$$

$$F\left(\begin{matrix}\\a\end{matrix}\middle|\, z\right) F\left(\begin{matrix}\\1-a\end{matrix}\middle|\, z\right) = \frac{1}{2}\left[1 + F\left(\begin{matrix}\frac{1}{2}\\ a,\, 1-a\end{matrix}\middle|\, 4z\right)\right] \tag{35.2-59a}$$

$$F\left(\begin{matrix}\\a\end{matrix}\middle|\, z\right) F\left(\begin{matrix}\\a+1\end{matrix}\middle|\, z\right) = F\left(\begin{matrix}a+\frac{1}{2}\\ a+1,\, 2a\end{matrix}\middle|\, 4z\right) \tag{35.2-59b}$$

35.2.5 Examples: elementary functions

The 'well-known' functions like exp, log and sin are expressed as hypergeometric functions. In some cases a transformation is applied to give an alternative series.

35.2.5.1 Powers, roots, and binomial series

$$\frac{1}{(1-z)^a} = F\left(\begin{matrix}a\\\end{matrix}\middle|\, z\right) = \sum_{k=0}^{\infty}\binom{a+k-1}{k} z^k = F\left(\begin{matrix}-a\\\end{matrix}\middle|\, \frac{-z}{1-z}\right) \tag{35.2-60a}$$

$$(1+z)^a = F\left(\begin{matrix}-a\\\end{matrix}\middle|\, -z\right) = \sum_{k=0}^{\infty}\binom{a}{k} z^k = F\left(\begin{matrix}a\\\end{matrix}\middle|\, \frac{z}{1+z}\right) \tag{35.2-60b}$$

An important special case of relation 35.2-60a is

$$\frac{1}{1-z} \;=\; F\left(\begin{matrix}1\\ \end{matrix}\middle|\, z\right) \;=\; \sum_{k=0}^{\infty} z^k \;=\; F\left(\begin{matrix}\frac{1}{2},\, 1\\ 2\end{matrix}\middle|\, 4z(1-z)\right) \quad \text{where} \quad z<\frac{1}{2} \tag{35.2-61}$$

The last equality is obtained by setting $a=0$ in relation 35.2-42a on page 669.

$$F\left(\begin{matrix}-s,\, s+1\\ s\end{matrix}\middle|\, z\right) \;=\; (1-2z)\,(1-z)^{s-1} \tag{35.2-62}$$

$$F\left(\begin{matrix}\frac{n}{2},\, \frac{n+1}{2}\\ \frac{1}{2}\end{matrix}\middle|\, z\right) \;=\; \frac{(1-\sqrt{z})^{-n}+(1+\sqrt{z})^{-n}}{2} \tag{35.2-63a}$$

$$F\left(\begin{matrix}\frac{n}{2},\, \frac{n+1}{2}\\ n+1\end{matrix}\middle|\, z\right) \;=\; \left(\frac{1+\sqrt{1-z}}{2}\right)^{-n} \tag{35.2-63b}$$

$$F\left(\begin{matrix}\frac{n}{2},\, \frac{n+1}{2}\\ n\end{matrix}\middle|\, z\right) \;=\; \frac{1}{\sqrt{1-z}}\left(\frac{2}{1+\sqrt{1-z}}\right)^{n-1} \tag{35.2-63c}$$

$$F\left(\begin{matrix}\frac{n+1}{2},\, \frac{n+2}{2}\\ \frac{3}{2}\end{matrix}\middle|\, z\right) \;=\; \frac{(1-\sqrt{z})^{-n}-(1+\sqrt{z})^{-n}}{2\,n\,\sqrt{z}} \qquad \text{if} \quad n\neq 0 \tag{35.2-64a}$$

$$\;=\; \frac{1}{2\sqrt{z}}\,\log\left(\frac{1+\sqrt{z}}{1-\sqrt{z}}\right) \qquad \text{if} \quad n=0 \tag{35.2-64b}$$

$$\frac{(1+z)^n-(1-z)^n}{(1+z)^n+(1-z)^n} \;=\; n\,z\,F\left(\begin{matrix}\frac{1}{2}-\frac{n}{2},\, 1-\frac{n}{2}\\ \frac{3}{2}\end{matrix}\middle|\, z^2\right)\Big/F\left(\begin{matrix}\frac{1}{2}-\frac{n}{2},\, -\frac{n}{2}\\ \frac{1}{2}\end{matrix}\middle|\, z^2\right) \tag{35.2-65a}$$

$$\;=\; n\,z\,F\left(\begin{matrix}\frac{1}{2}-\frac{n}{2},\, \frac{1}{2}+\frac{n}{2}\\ \frac{3}{2}\end{matrix}\middle|\, \frac{z^2}{z^2-1}\right)\Big/F\left(\begin{matrix}\frac{1}{2}-\frac{n}{2},\, \frac{1}{2}+\frac{n}{2}\\ \frac{1}{2}\end{matrix}\middle|\, \frac{z^2}{z^2-1}\right) \tag{35.2-65b}$$

35.2.5.2 Chebyshev polynomials

The Chebyshev polynomials are treated in section 34.2 on page 645.

$$F\left(\begin{matrix}n,\, -n\\ \frac{1}{2}\end{matrix}\middle|\, z\right) \;=\; T_n(1-2z) \tag{35.2-66a}$$

$$T_n(z) \;=\; F\left(\begin{matrix}n,\, -n\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) \tag{35.2-66b}$$

$$U_n(z) \;=\; (n+1)\,F\left(\begin{matrix}-n,\, n+2\\ \frac{3}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) \tag{35.2-66c}$$

Using relation 34.2-14 on page 648 (as $T_n(T_{1/n}(z))=z=\mathrm{id}(z)$) we find that

$$F\left(\begin{matrix}n,\, -n\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right)^{[-1]} \;=\; F\left(\begin{matrix}\frac{1}{n},\, -\frac{1}{n}\\ \frac{1}{2}\end{matrix}\middle|\, \frac{1-z}{2}\right) \tag{35.2-67}$$

near $z=1$ (here $F^{[-1]}$ denotes the inverse function).

35.2.5.3 Hermite polynomials

The Hermite polynomials $H_n(z)$ can be defined by the recurrence

$$H_{n+1}(z) \quad = \quad 2z\,H_n(z) - 2n\,H_{n-1}(x) \tag{35.2-68}$$

where $H_0(z) = 1$ and $H_1(z) = 2z$. The first few are

$$\begin{aligned} H_0 &= 1 \\ H_1 &= 2z \\ H_2 &= 4z^2 - 2 \\ H_3 &= 8z^3 - 12z \\ H_4 &= 16z^4 - 48z^2 + 12 \\ H_5 &= 32z^5 - 160z^3 + 120z \\ H_6 &= 64z^6 - 480z^4 + 720z^2 - 120 \\ H_7 &= 128z^7 - 1344z^5 + 3360z^3 - 1680z \\ H_8 &= 256z^8 - 3584z^6 + 13440z^4 - 13440z^2 + 1680 \\ H_9 &= 512z^9 - 9216z^7 + 48384z^5 - 80640z^3 + 30240z \\ H_{10} &= 1024z^{10} - 23040z^8 + 161280z^6 - 403200z^4 + 302400z^2 - 30240 \end{aligned} \tag{35.2-69}$$

For nonnegative integer n we have

$$H_n(z) \quad = \quad (2z)^n\,F\left(\begin{matrix} -\frac{1}{2}\,n,\ -\frac{1}{2}\,(n-1) \\ \end{matrix}\middle|\, -\frac{1}{z^2}\right) \tag{35.2-70}$$

35.2.5.4 Stirling numbers of the first kind

A generating hypergeometric function for the (unsigned) Stirling numbers of the first kind $s(n,m)$ (see section 10.15 on page 267) is given by

$$F\left(\begin{matrix} 1,\ e \\ \end{matrix}\middle|\, z\right) \quad = \quad \sum_{n=0}^{\infty} e^{\overline{m}}\,z^n \;=\; \sum_{n=0}^{\infty}\left(\sum_{m=1}^{m} e^m\,s(n,m)\right) z^n \tag{35.2-71a}$$

$$\begin{aligned} &= 1 + e\,z + \left(e + e^2\right) z^2 + \left(2e + 3e^2 + e^3\right) z^3 + \\ &\quad + \left(6e + 11e^2 + 6e^3 + e^4\right) z^4 + \\ &\quad + \left(24e + 50e^2 + 35e^3 + 10e^4 + e^5\right) z^5 + \\ &\quad + \left(120e + 274e^2 + 225e^3 + 85e^4 + 15e^5 + e^6\right) z^6 + \ldots \end{aligned} \tag{35.2-71b}$$

35.2.5.5 Logarithm and exponential function

$$\log(1+z) \quad = \quad z\,F\left(\begin{matrix} 1,\ 1 \\ 2 \end{matrix}\middle|\, -z\right) \;=\; \sum_{k=0}^{\infty} \frac{(-1)^k\,z^{k+1}}{k+1} \tag{35.2-72a}$$

$$\log\left(\frac{1+z}{1-z}\right) \quad = \quad 2\,z\,F\left(\begin{matrix} \frac{1}{2},\ 1 \\ \frac{3}{2} \end{matrix}\middle|\, z^2\right) \;=\; \log\left(1 + \frac{2}{1-z}\right) \tag{35.2-72b}$$

For large arguments z the following relation can be useful:

$$\log(1+z) \quad = \quad -\log\left(1 - \frac{z}{1+z}\right) \;=\; \frac{z}{1+z}\,F\left(\begin{matrix} 1,\ 1 \\ 2 \end{matrix}\middle|\, \frac{z}{1+z}\right) \tag{35.2-73}$$

$$\exp(z) \quad = \quad F\left(\begin{matrix} \\ \end{matrix}\middle|\, z\right) \;=\; \sum_{k=0}^{\infty} \frac{z^k}{k!} \tag{35.2-74}$$

35.2.5.6 Bessel functions and error function

The Bessel functions J_n of the first kind, and the modified Bessel functions I_n (as given in [1]):

$$J_n(z) = \frac{(z/2)^n}{n!}\, F\left(\begin{matrix} \\ n+1 \end{matrix}\middle|\, \frac{-z^2}{4}\right) \tag{35.2-75a}$$

$$I_n(z) = \frac{(z/2)^n}{n!}\, F\left(\begin{matrix} \\ n+1 \end{matrix}\middle|\, \frac{z^2}{4}\right) \tag{35.2-75b}$$

$$F\left(\begin{matrix} \\ n \end{matrix}\middle|\, z\right) = \frac{(n-1)!}{z^{(n-1)/2}}\, I_{n-1}(2\sqrt{z}) \tag{35.2-75c}$$

$$F\left(\begin{matrix} \\ 1 \end{matrix}\middle|\, z\right) = I_0(2\sqrt{z}) = \sum_{k=0}^{\infty} \frac{z^k}{k!^2} \tag{35.2-75d}$$

Error function (the Kummer transformation, relation 35.2-51 on page 670, gives relation 35.2-76b):

$$\frac{\sqrt{\pi}}{2}\,\mathrm{erf}(z) := \int_{t=0}^{z} e^{-t^2}\, dt = z\, F\left(\begin{matrix} \frac{1}{2} \\ \frac{3}{2} \end{matrix}\middle|\, -z^2\right) \tag{35.2-76a}$$

$$= z\, e^{-z^2}\, F\left(\begin{matrix} 1 \\ \frac{3}{2} \end{matrix}\middle|\, z^2\right) = z\, e^{-z^2} \sum_{k=0}^{\infty} \frac{(2z^2)^k}{1\cdot 3\cdot 5\cdots(2k+1)} \tag{35.2-76b}$$

35.2.5.7 Trigonometric and hyperbolic functions

Series for sine and cosine:

$$\sin(z) = z\, F\left(\begin{matrix} \\ \frac{3}{2} \end{matrix}\middle|\, \frac{-z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{(-1)^k\, z^{2k+1}}{(2k+1)!} \tag{35.2-77a}$$

$$\sinh(z) = z\, F\left(\begin{matrix} \\ \frac{3}{2} \end{matrix}\middle|\, \frac{z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{z^{2k+1}}{(2k+1)!} \tag{35.2-77b}$$

Applying the transformation 35.2-55 on page 671 to relation 35.2-77a gives

$$[\sin(z)]^2 = z^2\, F\left(\begin{matrix} 1 \\ \frac{3}{2},\, 2 \end{matrix}\middle|\, -z^2\right) \tag{35.2-78}$$

$$\cos(z) = F\left(\begin{matrix} \\ \frac{1}{2} \end{matrix}\middle|\, \frac{-z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{(-1)^k\, z^{2k}}{(2k)!} \tag{35.2-79a}$$

$$\cosh(z) = F\left(\begin{matrix} \\ \frac{1}{2} \end{matrix}\middle|\, \frac{z^2}{4}\right) = \sum_{k=0}^{\infty} \frac{z^{2k}}{(2k)!} \tag{35.2-79b}$$

$$\exp(z)\,\frac{\sinh(z)}{z} = F\left(\begin{matrix} 1 \\ 2 \end{matrix}\middle|\, 2z\right) \tag{35.2-80a}$$

$$\exp(-iz)\,\frac{\sin(z)}{z} = F\left(\begin{matrix} 1 \\ 2 \end{matrix}\middle|\, -2iz\right) \tag{35.2-80b}$$

Further expressions for the sine and cosine are

$$\frac{\sin(a\,z)}{a\,\sin(z)} = F\left(\begin{matrix} \frac{1+a}{2},\, \frac{1-a}{2} \\ \frac{3}{2} \end{matrix}\middle|\, +\sin(z)^2\right) \tag{35.2-81a}$$

$$\cos(a\,z) = F\left(\begin{matrix} +\frac{a}{2},\, -\frac{a}{2} \\ \frac{1}{2} \end{matrix}\middle|\, +\sin(z)^2\right) \tag{35.2-81b}$$

$$\frac{\cos(a\,z)}{\cos(z)} = F\left(\begin{matrix}\frac{1+a}{2},\,\frac{1-a}{2}\\ \frac{1}{2}\end{matrix}\,\middle|\,+\sin(z)^2\right) \tag{35.2-81c}$$

$$\frac{\sin(a\,z)}{a\,\sin(z)\,\cos(z)} = \frac{2\,\sin(a\,z)}{a\,\sin(2\,z)} = F\left(\begin{matrix}1+\frac{a}{2},\,1-\frac{a}{2}\\ \frac{3}{2}\end{matrix}\,\middle|\,+\sin(z)^2\right) \tag{35.2-81d}$$

Relations for the hyperbolic sine and cosine are obtained by replacing sin $\mapsto$ sinh, cos $\mapsto$ cosh, and negating the sign of the argument of the hypergeometric function. For example, from relation 35.2-81b one obtains

$$\cosh(a\,z) = F\left(\begin{matrix}+\frac{a}{2},\,-\frac{a}{2}\\ \frac{1}{2}\end{matrix}\,\middle|\,-\sinh(z)^2\right) \tag{35.2-81e}$$

35.2.5.8 Inverse trigonometric and hyperbolic functions

Series for the inverse tangent and cotangent:

$$\arctan(z) = -\frac{i}{2}\,\log\frac{1+iz}{1-iz} = \mathfrak{Im}\log(1+iz) \tag{35.2-82a}$$

$$= z\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\,\middle|\,-z^2\right) = \sum_{k=0}^{\infty}\frac{(-1)^k\,z^{2k+1}}{2k+1} \tag{35.2-82b}$$

By Pfaff's reflection law (relation 35.2-28b) on can obtain

$$\arctan(z) = \frac{z}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\,\middle|\,\frac{z^2}{1+z^2}\right) = \arccos\frac{1}{\sqrt{1+z^2}} \tag{35.2-82c}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\,\middle|\,\frac{z^2}{1+z^2}\right) \quad \text{by 35.2-28b} \tag{35.2-82d}$$

$$\operatorname{arctanh}(z) = \frac{1}{2}\,\log\frac{1+z}{1-z} \tag{35.2-83a}$$

$$= z\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\,\middle|\,z^2\right) = \sum_{k=0}^{\infty}\frac{z^{2k+1}}{2k+1} \tag{35.2-83b}$$

$$\operatorname{arccoth}(z) = \frac{1}{2}\,\log\frac{z+1}{z-1} = \sum_{k=0}^{\infty}\frac{1}{(2k+1)\,z^{2k+1}} \tag{35.2-83c}$$

$$\log(z) = 2\,\operatorname{arctanh}\frac{z-1}{z+1} = 2\,\operatorname{arccoth}\frac{z+1}{z-1} \tag{35.2-83d}$$

$$\operatorname{arccot}(z) = \arctan\left(\frac{1}{z}\right) = -\frac{i}{2}\,\log\frac{z+i}{z-i} \tag{35.2-84a}$$

$$= \frac{1}{z}\,F\left(\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\,\middle|\,-\frac{1}{z^2}\right) = \sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)\,z^{2k+1}} \tag{35.2-84b}$$

$$= \frac{1}{\sqrt{1+z^2}}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\,\middle|\,\frac{1}{1+z^2}\right) = \arcsin\frac{1}{\sqrt{1+z^2}} \tag{35.2-84c}$$

$$= \frac{z}{1+z^2}\,F\left(\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\,\middle|\,\frac{1}{1+z^2}\right) \tag{35.2-84d}$$

Applying Clausen's product formula (relation 35.2-44a on page 669) gives

$$[\arctan(z)]^2 = \frac{z^2}{1+z^2}\, F\left(\left.\begin{matrix}1,\,1,\,1\\ \frac{3}{2},\,2\end{matrix}\right| \frac{z^2}{1+z^2}\right) \tag{35.2-85a}$$

$$[\operatorname{arccot}(z)]^2 = \frac{1}{1+z^2}\, F\left(\left.\begin{matrix}1,\,1,\,1\\ \frac{3}{2},\,2\end{matrix}\right| \frac{1}{1+z^2}\right) \tag{35.2-85b}$$

Series for the inverse sine and cosine are

$$\arcsin(z) = z\, F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\right| z^2\right) = \arctan\frac{z}{\sqrt{1-z^2}} \tag{35.2-86a}$$

$$= z\, F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| \frac{1-\sqrt{1-z^2}}{2}\right) \quad \text{by 35.2-31b} \tag{35.2-86b}$$

$$= z\sqrt{1-z^2}\, F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| z^2\right) \tag{35.2-86c}$$

The two latter relations suggest the following argument reduction applicable for the inverse sine (and tangent). Let $G(z) = (1-\sqrt{1-z})/2$, then

$$F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| z\right) = \frac{1}{\sqrt{1-z}}\, F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| G(z)\right) \tag{35.2-87a}$$

$$= \frac{1}{\sqrt{1-z}}\,\frac{1}{\sqrt{1-G(z)}}\, F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| G(G(z))\right) = \ldots \tag{35.2-87b}$$

$$F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| z\right) = \left[\prod_{k=0}^{\infty} 1 - z_k\right]^{-1/2} \quad \text{where} \quad z_0 = z, \quad z_{k+1} = G(z_k) \tag{35.2-87c}$$

Clausen's product formula (relation 35.2-44a on page 669) can by applied to obtain

$$[\arcsin(z)]^2 = z^2\, F\left(\left.\begin{matrix}1,\,1,\,1\\ \frac{3}{2},\,2\end{matrix}\right| z^2\right) \tag{35.2-88}$$

$$\arccos(z) = \frac{\pi}{2} - \arcsin(z) = \operatorname{arccot}\frac{z}{\sqrt{1-z^2}} \tag{35.2-89}$$

$$\operatorname{arcsinh}(z) = \log(z+\sqrt{1+z^2}) = z\, F\left(\left.\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ \frac{3}{2}\end{matrix}\right| -z^2\right) \tag{35.2-90a}$$

$$= \frac{z}{\sqrt{1+z^2}}\, F\left(\left.\begin{matrix}\frac{1}{2},\,1\\ \frac{3}{2}\end{matrix}\right| \frac{z^2}{1+z^2}\right) \quad \text{by 35.2-28b} \tag{35.2-90b}$$

$$= z\, F\left(\left.\begin{matrix}1,\,1\\ \frac{3}{2}\end{matrix}\right| \frac{1-\sqrt{1+z^2}}{2}\right) \quad \text{by 35.2-31b} \tag{35.2-90c}$$

35.2.6 The function x^x

Boldly setting $a = 1+z$ in $(1+z)^a = F\left(\left.\begin{matrix}-a\end{matrix}\right| -z\right)$ (relation 35.2-60b on page 671) gives

$$(1+z)^{(1+z)} = F\left(\left.\begin{matrix}-1-z\end{matrix}\right| -z\right) = \exp\left[(1+z)\log(1+z)\right] \tag{35.2-91a}$$

$$= 1+\frac{(z+1)}{1}\,z\left[1+\frac{(z+0)}{2}\,z\left[1+\frac{(z-1)}{3}\,z\left[1+\frac{(z-2)}{4}\,z\left[1+\dots\right]\right]\right]\right] \tag{35.2-91b}$$

$$= 1+z+z^2+\frac{1}{2}\,z^3+\frac{1}{3}\,z^4+\frac{1}{12}\,z^5+\frac{3}{40}\,z^6-\frac{1}{120}\,z^7+\frac{59}{2520}\,z^8-\frac{71}{5040}\,z^9\pm\dots \tag{35.2-91c}$$

This somewhat surprising expression allows the computation of x^x without computing exp() or log(). The series converges for real $z>0$ so we can compute x^x (where $x=1+z$) for real $x>+1$ as

$$x^x = F\left(\begin{matrix}-x\\ \end{matrix}\middle|\, -x+1\right) \tag{35.2-92a}$$

$$= 1+\frac{x-0}{1}\,(x-1)\left[1+\frac{x-1}{2}\,(x-1)\left[1+\frac{x-2}{3}\,(x-1)\left[1+\dots\right]\right]\right] \tag{35.2-92b}$$

We denote the series obtained by truncating after the n-th term of the hypergeometric function by $g_n(x)$. For example, with $n=2$ and $n=4$ we obtain:

$$g_2(x) = \frac{1}{2}\,x^4-\frac{3}{2}\,x^3+\frac{5}{2}\,x^2-\frac{3}{2}\,x+1 \tag{35.2-93a}$$

$$= \frac{1}{2}\,z^4+\frac{1}{2}\,z^3+z^2+z+1 \tag{35.2-93b}$$

$$g_4(x) = \frac{1}{24}\,x^8-\frac{5}{12}\,x^7+\frac{15}{8}\,x^6-\frac{19}{4}\,x^5+\frac{61}{8}\,x^4-\frac{31}{4}\,x^3+\frac{131}{24}\,x^2-\frac{25}{12}\,x+1 \tag{35.2-93c}$$

$$= \frac{1}{24}\,z^8-\frac{1}{12}\,z^7+\frac{1}{8}\,z^6+\frac{1}{12}\,z^5+\frac{1}{3}\,z^4+\frac{1}{2}\,z^3+z^2+z+1 \tag{35.2-93d}$$

We have $g_n(n)=n^n$ and further $g_n(k)=k^k$ for all integer $k\le n$. Thus we have just invented a curious way to find polynomials of degree $2n$ that interpolate k^k for $0\le k\le n$ (setting $0^0:=1$ for our purposes).

The polynomials actually give acceptable estimates for x^x also for non-integer x, especially for x near 1. The (unique) degree-n polynomials $i_n(x)$ that are obtained by interpolating the values k^k have much bigger coefficients and give values far away from x^x for non-integer arguments x.

For $0<x<n$ the interpolating polynomials $i_n(x)$ give an estimate that is consistently worse than $g_n(x)$ for non-integer values of x. The same is true even for the polynomials $i_{2n}(x)$ that interpolate k^k for $0\le k\le 2n$ (so that $\deg(i_{2n})=\deg(g_n)=2n$). In fact, the $i_{2n}(x)$ approximate consistently worse than $i_n(x)$ for non-integer x.

Finally, the Padé approximants $p_{[n,n]}(x)$ for $g_n(x)$ give estimates that are worse than with both $i_n(x)$ or $g_n(x)$. Further, $g_n(x)\neq x^x$ even for integer x and the $p_{[n,n]}(x)$ have a pole on the real axis near $x=1$. That is, we found a surprisingly good and compact polynomial approximation for the function x^x.

The sequence of the n-th derivatives of $(1+z)^{(1+z)}$ at $z=0$ is entry A005727 of [214]:

```
? Vec(serlaplace(exp((1+z)*log(1+z))))
  [1, 1, 2, 3, 8, 10, 54, -42, 944, -5112, 47160, -419760, 4297512, ... ]
```

Many other expressions for the function x^x can be given, we note just one: set $s=2+z$ in relation 35.2-62 on page 672 to obtain

$$(1+z)^{(1+z)} = \frac{1}{1+2\,z}\,F\left(\begin{matrix}-z-2,\ z+3\\ z+2\end{matrix}\middle|\, -z\right) \tag{35.2-94}$$

35.2.7 Elliptic K and E

In order to avoid the factor $\frac{\pi}{2}$ we let $\tilde{K}:=\frac{2\,K}{\pi}$, $\tilde{E}:=\frac{2\,E}{\pi}$, then

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\middle|\, k^2\right) \tag{35.2-95a}$$

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\ \frac{1}{2}\\ 1\end{matrix}\middle|\, k^2\right) \tag{35.2-95b}$$

See section 30.2 on page 575 for explicit expansions. We further set

$$\tilde{N}(k) = F\left(\begin{matrix}-\frac{1}{2}, -\frac{1}{2}\\ 1\end{matrix}\Big| k^2\right) \tag{35.2-95c}$$

$$= 1 + \frac{1}{4}k^2 + \frac{1}{64}k^4 + \frac{1}{256}k^6 + \frac{25}{16384}k^8 + \frac{49}{65536}k^{10} + \frac{441}{1048576}k^{10} + \ldots \tag{35.2-95d}$$

A special value is $\tilde{N}(1) = 4/\pi$. Most of the following relations can be written in several ways by one of the identities

$$k' = \sqrt{1-k^2} \tag{35.2-96a}$$

$$-\frac{k^2}{1-k^2} = -\left(\frac{k}{k'}\right)^2 = -\frac{1-k'^2}{k'^2} \tag{35.2-96b}$$

$$-\left(\frac{k}{1-\sqrt{1-k^2}}\right)^2 = -\frac{1+k'}{1-k'} = -\frac{1-k'^2}{(1-k')^2} = -\frac{(1+k')^2}{1-k'^2} \tag{35.2-96c}$$

$$\frac{2}{1+k'} = 1 + \frac{1-k'}{1+k'} \tag{35.2-96d}$$

$$\frac{4k}{(1+k)^2} = 1 - \left(\frac{1-k}{1+k}\right)^2 \tag{35.2-96e}$$

$$\frac{-4k^2}{(1-k^2)^2} = -\left(\frac{2k}{k'^2}\right)^2 = 1 - \left(\frac{1+k^2}{1-k^2}\right)^2 \tag{35.2-96f}$$

35.2.7.1 Relations for K

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{2}, \frac{1}{2}\\ 1\end{matrix}\Big| k^2\right) \tag{35.2-97}$$

$$= \frac{1}{k'}F\left(\begin{matrix}\frac{1}{2}, \frac{1}{2}\\ 1\end{matrix}\Big| -\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 35.2-28b} \tag{35.2-98}$$

$$\tilde{K}(k) = \frac{1}{1-k'}F\left(\begin{matrix}\frac{1}{2}, \frac{1}{2}\\ 1\end{matrix}\Big| -\frac{1+k'}{1-k'}\right) \tag{35.2-99}$$

From the product form (relation 30.2-8 on page 576) we obtain

$$\tilde{K}(k) = \frac{2}{1+k'}F\left(\begin{matrix}\frac{1}{2}, \frac{1}{2}\\ 1\end{matrix}\Big| \left(\frac{1-k'}{1+k'}\right)^2\right) \tag{35.2-100a}$$

The relation can be written as

$$\tilde{K}(k) = (1+z(k))\ \tilde{K}\left(z(k)\right) \quad \text{where} \quad z(k) := \frac{1-k'}{1+k'} \tag{35.2-100b}$$

Relation 35.2-40f on page 669 with $a = \frac{1}{2}$ gives

$$\tilde{K}(k) = \frac{1}{1+k}\tilde{K}\left(\frac{2\sqrt{k}}{1+k}\right) \tag{35.2-101}$$

$$\tilde{K}(k) = \frac{1}{\sqrt{1-k^2}}F\left(\begin{matrix}\frac{1}{4}, \frac{1}{4}\\ 1\end{matrix}\Big| \frac{-4k^2}{(1-k^2)^2}\right) = \frac{1}{k'}F\left(\begin{matrix}\frac{1}{4}, \frac{1}{4}\\ 1\end{matrix}\Big| 1 - \left(\frac{1+k^2}{1-k^2}\right)^2\right) \tag{35.2-102a}$$

Euler's transform on 35.2-102a gives:

$$\tilde{K}(k) = \frac{1}{k'}\,\frac{1+k^2}{1-k^2}\,F\left(\begin{matrix}\frac{3}{4},\,\frac{3}{4}\\ 1\end{matrix}\Big|\,-\left(\frac{2k}{k'^2}\right)^2\right) \tag{35.2-103}$$

$$\tilde{K}(k) = F\left(\begin{matrix}\frac{1}{4},\,\frac{1}{4}\\ 1\end{matrix}\Big|\,(2\,k\,k')^2\right) \qquad \text{by 35.2-31a} \tag{35.2-104}$$

35.2.7.2 Relations for E

$$\tilde{E}(k) = F\left(\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\Big|\,k^2\right) \tag{35.2-105}$$

$$\tilde{E}(k) = k'\,F\left(\begin{matrix}-\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\Big|\,-\left(\frac{k}{k'}\right)^2\right) \qquad \text{by 35.2-28b} \tag{35.2-106}$$

The following relation resembles relation 35.2-100a:

$$\tilde{E}(k) = \frac{\left(1+\sqrt{1-k^2}\right)}{2}\,F\left(\begin{matrix}-\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\Big|\,\left(\frac{1-\sqrt{1-k^2}}{1+\sqrt{1-k^2}}\right)^2\right) \tag{35.2-107a}$$

$$= \frac{1+k'}{2}\,F\left(\begin{matrix}-\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\Big|\,\left(\frac{1-k'}{1+k'}\right)^2\right) \tag{35.2-107b}$$

$$= (1+z(k))^{-1}\;\tilde{N}\left(z(k)\right) \quad \text{where} \quad z(k) := \frac{1-k'}{1+k'} \tag{35.2-107c}$$

35.2.7.3 Relations for N

$$\tilde{N}(k) = \sqrt{1-k^2}\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{3}{4}\\ 1\end{matrix}\Big|\,\frac{-4\,k^2}{(1-k^2)^2}\right) \qquad \text{by 35.2-40e} \tag{35.2-108a}$$

$$= k'\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{3}{4}\\ 1\end{matrix}\Big|\,-\left(\frac{2k}{k'^2}\right)^2\right) \tag{35.2-108b}$$

$$= \sqrt{1+k^2}\,F\left(\begin{matrix}-\frac{1}{4},\,\frac{1}{4}\\ 1\end{matrix}\Big|\,\left(\frac{2k}{k^2+1}\right)^2\right) \qquad \text{by 35.2-28b} \tag{35.2-108c}$$

Relation 35.2-25 on page 667 with $a = b = -1/2$ and $c = 1$ gives

$$F\left(\begin{matrix}\frac{1}{2},\,-\frac{1}{2}\\ 1\end{matrix}\Big|\,z\right) = \frac{1}{2}\,F\left(\begin{matrix}\frac{3}{2},\,-\frac{1}{2}\\ 1\end{matrix}\Big|\,z\right) + \frac{1}{2}\,F\left(\begin{matrix}\frac{1}{2},\,\frac{1}{2}\\ 1\end{matrix}\Big|\,z\right) \tag{35.2-109}$$

Applying 35.2-28c on page 667 to the second function ($F\left(\begin{matrix}3/2,\,-1/2\\ 1\end{matrix}\Big|\,z\right) = (1-z)\,F\left(\begin{matrix}-1/2,\,-1/2\\ 1\end{matrix}\Big|\,\frac{-z}{1-z}\right)$) and rearranging gives

$$2\,\tilde{E}(k) - \tilde{K}(k) = k'\,\tilde{N}\left(i\,\frac{k}{k'}\right) \tag{35.2-110}$$

Applying the transformation 35.2-40f on page 669 on the defining relation gives the key to fast computation of the function $\tilde{N}(k)$:

$$\tilde{N}(k) = (1+k)\;\tilde{E}\left(\frac{2\sqrt{k}}{1+k}\right) \tag{35.2-111}$$

The relation

$$2\,\tilde{E}(k) - k'^2\,\tilde{K}(k) \quad = \quad \tilde{N}(k) \tag{35.2-112}$$

can be used to rewrite Legendre's relation (equation 30.2-17b on page 577) as either

$$\frac{4}{\pi} \quad = \quad \tilde{N}\,\tilde{K}' + \tilde{K}\,\tilde{N}' - \tilde{K}\,\tilde{K}' \tag{35.2-113}$$

or, by setting $N := \pi/2\,\tilde{N}$,

$$\pi \quad = \quad N\,K' + K\,N' - K\,K' \tag{35.2-114}$$

35.3 Continued fractions

A *continued fraction* is an expression of the form:

$$K(a,b) \quad = \quad a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cdots}}}} \tag{35.3-1}$$

Continued fractions are sometimes expressed in the following form:

$$K(a,b) \quad = \quad a_0 + \frac{b_1}{a_1+}\,\frac{b_2}{a_2+}\,\frac{b_3}{a_3+}\,\frac{b_4}{a_4+}\,\ldots \tag{35.3-2}$$

The a_k and b_k are called the k-th *partial numerators* and *denominators*.

For $k > 0$ let P_k/Q_k be the value of the above fraction if b_{k+1} is set to zero (that is, the continued fraction terminates at index k). The ratio is called the k-th *convergent* of the continued fraction:

$$\frac{P_k}{Q_k} \quad = \quad a_0 + \frac{b_1}{a_1+}\,\frac{b_2}{a_2+}\,\frac{b_3}{a_3+}\,\frac{b_4}{a_4+}\,\ldots\,\frac{b_{k-1}}{a_{k-1}+}\,\frac{b_k}{a_k} \tag{35.3-3}$$

We note that multiplication of a_i, b_i, b_{i+1} by some nonzero value does not change the value of the continued fraction. Thereby

$$a_0 + \frac{b_1}{a_1+}\,\frac{b_2}{a_2+}\,\frac{b_3}{a_3+}\,\frac{b_4}{a_4+}\,\ldots \quad = \quad a_0 + \frac{c_1\,b_1}{c_1\,a_1+}\,\frac{c_2\,c_1\,b_2}{c_2\,a_2+}\,\frac{c_3\,c_2\,b_3}{c_3\,a_3+}\,\frac{c_4\,c_3\,b_4}{c_4\,a_4+}\,\ldots \tag{35.3-4}$$

where all c_i are arbitrary nonzero constants.

35.3.1 Simple continued fractions

Continued fractions where all b_k are equal to one (and all the a_k are positive) are called *simple continued fractions*. Rational numbers have terminating continued fractions. Note that the expression of a rational number as simple continued fraction is not unique:

$$[a_0,\,\ldots,\,a_{n-1},\,a_n] \quad = \quad [a_0,\,\ldots,\,a_{n-1},\,a_n - 1,\,1] \quad \text{if } a_n > 1 \tag{35.3-5a}$$

$$[a_0,\,\ldots,\,a_{n-1},\,1] \quad = \quad [a_0,\,\ldots,\,a_{n-1} + 1] \qquad \text{if} \quad a_n = 1 \tag{35.3-5b}$$

Solutions of the quadratic equation $\alpha\,x^2 + \beta\,x + \gamma = 0$ that are not rational ($\Delta := \beta^2 - 4\,\alpha\,\gamma$ not a square) have simple continued fractions that are eventually periodic. For example:

```
? contfrac(sqrt(5))
  [2, 4, 4, 4, 4, 4, ...]
? contfrac(2+sqrt(3))
  [3, 1,2, 1,2, 1,2, ...]
? contfrac(sqrt(19))
  [4, 2,1,3,1,2,8, 2,1,3,1,2,8, 2,1,3,1,2,8, ...]
```

For the k-th convergent P_k/Q_k (in lowest terms) of the simple continued fraction expansion of some number x then the convergent is a *best approximation* in the following sense: if p/q is any better rational approximation to x (that is, $\left|\frac{p}{q} - x\right| < \left|\frac{P_k}{Q_k} - x\right|$), then one must have $q > Q_k$.

For the simple continued fraction of x one has

$$\left|x - \frac{P_n}{Q_n}\right| \quad \leq \quad \frac{1}{Q_n\, Q_{n-1}} \; < \; \frac{1}{Q_n^2} \tag{35.3-6}$$

and equality can only occur with terminating continued fractions.

35.3.1.1 Computing the simple continued fraction of a number

Given a numerical quantity one can compute the sequence a_k of its simple continued fraction by the following algorithm:

```
procedure number_to_scf(x, n, a[0..n-1])
{
    for k:=0 to n-1
    {
        xi := floor(x)
        a[k] := xi
        x := 1 / (x-xi)
    }
}
```

Here `n` is the number of requested terms a_k. Obviously some check has to be inserted in order to avoid possible division be zero (and indicate a terminating continued fraction as will occur for rational x). If in the process one keeps track of the exact rational values of the simple continued fraction convergents (using the recursion relations) then the algorithm can be used to produce a 'best possible' rational approximation where the denominator does not exceed a certain specified size. Alternatively one can stop as soon as the convergent is within some specified bound.

35.3.1.2 Continued fractions of polynomial roots

```
r = RootOf(z^3 - 2) == 1.2599210...
contfrac(r) == [1, 3, 1, 5, 1, 1, 4, 1, 1, 8, 1, 14, 1, 10, 2, ...]

f=z^3 - 2    r=1.25992104989487  ==> 1
f=z^3 - 3*z^2 - 3*z - 1    r=3.84732210186307  ==> 3
f=10*z^3 - 6*z^2 - 6*z - 1    r=1.18018873554841  ==> 1
f=3*z^3 - 12*z^2 - 24*z - 10    r=5.54973648578239  ==> 5
f=55*z^3 - 81*z^2 - 33*z - 3    r=1.81905335713127  ==> 1
f=62*z^3 + 30*z^2 - 84*z - 55    r=1.22092167902528  ==> 1
f=47*z^3 - 162*z^2 - 216*z - 62    r=4.52649103705930  ==> 4
f=510*z^3 - 744*z^2 - 402*z - 47    r=1.89936756679748  ==> 1
f=683*z^3 + 360*z^2 - 786*z - 510    r=1.11189244188653  ==> 1
f=253*z^3 - 1983*z^2 - 2409*z - 683    r=8.93715413784671  ==> 8
f=17331*z^3 - 14439*z^2 - 4089*z - 253    r=1.06706032616757  ==> 1
f=1450*z^3 - 19026*z^2 - 37554*z - 17331    r=14.9119465584038  ==> 14
```

Figure 35.3-A: Computation of the continued fraction of the positive real root of the polynomial $z^3 - 2$.

Let $r > 1$ be the only real positive root of a polynomial $F(x)$ with integer coefficients and positive leading coefficient. Then the (simple) continued fraction $[a_0, a_1, \ldots, a_n]$ of r can be computed as follows (taken from [160, p.261]):

```
r = RootOf(z^2 - 29) == 5.38516480...
contfrac(r) ==[5,   2, 1, 1, 2, 10,   2, 1, 1, 2, 10,  ...]

f=z^2 - 29    r=5.38516480713450  ==> 5
f=4*z^2 - 10*z - 1    r=2.59629120178363  ==> 2
f=5*z^2 - 6*z - 4    r=1.67703296142690  ==> 1
f=5*z^2 - 4*z - 5    r=1.47703296142690  ==> 1
f=4*z^2 - 6*z - 5    r=2.09629120178363  ==> 2
f=z^2 - 10*z - 4    r=10.3851648071345  ==> 10

f=4*z^2 - 10*z - 1    r=2.59629120178363  ==> 2  <--= restart period
f=5*z^2 - 6*z - 4    r=1.67703296142690  ==> 1
```

Figure 35.3-B: Computation of the continued fraction of the positive real root of the polynomial z^2-29.

1. Set $k=0$, $F_0(x)=F(x)$, and $d=\deg(F)$.
2. Find the (unique) real positive root r_k of $F_k(x)$, set $a_k=\lfloor r \rfloor$. If $k=n$ then stop.
3. Set $G(x)=F_k(x+a_k)$, set $F_{k+1}=-G^*(x)=-x^d\,G(1/x)$.
4. Set $k=k+1$ and goto step 2.

A simple demonstration is

```
f = z^3 - 2
ff(y)=subst(f, z, y)  \\ for solve() function

{ for (k=1, 12,
    print1("  f=", f);
    r = solve(x=0.9, 1e9, ff(x));  \\ lazy implementation
    print1("    r=", r);
    ak = floor( r );
    print1("  ==> ", ak);
    g = subst(f, z, z+ak);  \\ shifted polynomial
    f = -polrecip( g );     \\ negated reciprocal of g
    print();
 ); }
```

The output with $F(x)=x^3-2$ is shown in figure 35.3-A. With quadratic equations one obtains periodic continued fractions, figure 35.3-B shows the computation for $F(x)=x^2-29$. For a comparison of methods for the computation of continued fractions for algebraic numbers see [64].

35.3.2 Computation of the convergents (evaluation)

The computation of the sequence of convergents uses the recurrence

$$P_k = a_k\,P_{k-1}+b_k\,P_{k-2} \tag{35.3-7a}$$
$$Q_k = a_k\,Q_{k-1}+b_k\,Q_{k-2} \tag{35.3-7b}$$

Set $\frac{P_{-1}}{Q_{-1}}:=\frac{1}{0}$ and $\frac{P_0}{Q_0}:=\frac{a_0}{1}$ to initialize. The following is a procedure that computes the sequences of values P_k and Q_k for $k=-1\ldots n$ for a given continued fraction:

```
procedure ratios_from_cf(a[0..n], b[0..n], n, P[-1..n], Q[-1..n])
{
    P[-1] := 1
    Q[-1] := 0

    P[0] := a[0]
    Q[0] := 1

    for k:=1 to n
    {
        P[k] := a[k] * P[k-1] + b[k] * P[k-2]
        Q[k] := a[k] * Q[k-1] + b[k] * Q[k-2]
    }
}
```

If only the last ratio is of interest, the 'backward' variant of the algorithm can be used. The version that computes the numerical value `x` from the first `n` terms of a simple continued fraction is:

```
function ratio_from_cf(a[0..n-1], n)
{
    x := a[n-1]
    for k:=n-2 to 0 step -1
    {
        x := 1/x + a[k]
    }
    return x
}
```

Using rational arithmetic and a general (non-simple) continued fraction, the algorithm becomes:

```
function ratio_from_cf(a[0..n-1], b[0..n-1], n)
{
    P := a[n-1]
    Q := b[n-1]
    for k:=n-2 to 0 step -1
    {
        {P, Q} := {a[k]*P+b[k]*Q, P}  // x := b[k] / x + a[k]
    }
    return P/Q
}
```

Implementation

Converting a number to a simple continued fraction can be done with pari/gp's builtin function `contfrac()`. The final convergent can be computed with `contfracpnqn()`:

```
? default(realprecision,23)
    realprecision = 28 significant digits (23 digits displayed)
? Pi
    3.1415926535897932384626
? cf=contfrac(Pi)
    [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3]
? ?contfracpnqn
    contfracpnqn(x): [p_n,p_{n-1}; q_n,q_{n-1}] corresponding to the continued fraction x.
? m=contfracpnqn(cf)
    [428224593349304 139755218526789]
    [136308121570117 44485467702853]
? 1.0*m[1,1]/m[2,1]
    3.1415926535897932384626
```

The number of terms of the continued fraction depends on the precision used, with greater precision more terms can be computed. The computation of the m-th convergent of a continued fraction given as two vectors `a[]` and `b[]` can be implemented as (backward variant):

```
cfab2r(a,b, m=-2)=
{
    local(n, r);
    n = length(a);
    if ( m>-2, m = min(n, m) ); \\ default: m=n
    if ( m>=n,  m=n-1 );
    if ( m<0, return( 0 ) ); \\ infinity
    r = 0;
    m += 1;
    forstep (k=m, 2, -1,  r = b[k]/(a[k]+r); );
    r += a[1];  \\ b[1] unused
    return( r );
}
```

Alternatively, one can use the recursion relations 35.3-7a and 35.3-7b. We do not store all pairs P_n, Q_n but only return the final pair P_m, Q_m:

```
cfab2pq(a,b,m=-2)=
{
    local(n, p, p1, p2, q, q1, q2, i);
    n = length(a)-1;
    if ( m>-2, m = min(n, m) ); \\ default: m=n
    if ( m<0, return( [1, 0] ) ); \\ infinity
    p1 = 1;
    q1 = 0;
    p = a[1];
    q = 1;  \\ b[1] unused
```

```
    for (k=1, m,
        i = k+1;
        p2 = p1;  p1 = p;
        q2 = q1;  q1 = q;
        p = a[i]*p1 + b[i]*p2;
        q = a[i]*q1 + b[i]*q2;
    );
    return( [p,q] );
}
```

```
    default(realprecision, 55);  \\ use enough precision
    default(format, "g.11"); \\ print with moderate precision
    default(echo, 0);
    \r contfrac.inc.gp  \\ functions cfab2pq and cfab2r

    x=4.0/Pi
    n=15
    /* set up the continued fraction: */
    a=vector(n, j, 2);  a[1]=1;
    b=vector(n, j, (2*j-3)^2);

    /* print convergents and their error: */
    { for(k=0, n-1,
        t=cfab2pq(a,b, k);
        p=t[1]; q=t[2];
        print1(k, ":  ",p, " / ", q);
        print1("\n  d=", x-p/q);
        print();
      ); }

    quit; /* ------ end of script -------- */

    /* ------ start output: ------ */
    15  /* =n */
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]  /* =a */
    [1, 1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729]  /* =b */
    1.2732395447 /* = 4/Pi */

    0:  1 / 1
      d=0.27323954473
    1:  3 / 2
      d=-0.22676045526
    2:  15 / 13
      d=0.11939339088
    3:  105 / 76          /* =p3/q3 */
      d=-0.10833940263    /* =p3/q3-4/Pi */
    4:  945 / 789
      d=0.075520913556
    5:  10395 / 7734
      d=-0.070825622061
    [--snip--]
    13:  213458046676875 / 163842638377950
      d=-0.029583998575
    14:  6190283353629375 / 4964894559637425
      d=0.026428906710
```

Figure 35.3-C: A pari/gp script demonstrating the function `cfab2pq()` that computes the convergents of a continued fraction (top) and its output (bottom, comments added). Here convergence is rather slow.

We use our routines to compute the convergents of the continued fraction for $4/\pi$ given 1658 by Brouncker:

$$\frac{4}{\pi} \;=\; 1+\cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\cfrac{7^2}{2+\cdots}}}} \;=\; a_0+\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{a_3+\cfrac{b_4}{a_4+\cdots}}}} \tag{35.3-8}$$

Figure 35.3-C shows how to set up the vectors containing the a_k and b_k and check the convergents.

Fast evaluation as matrix product

For the evaluation of a continued fraction with a large number of terms rewrite relations 35.3-7a and 35.3-7b as

$$\begin{bmatrix} P_k & Q_k \\ P_{k-1} & Q_{k-1} \end{bmatrix} \;=\; \prod_{j=0}^{k} \begin{bmatrix} a_j & 1 \\ b_j & 0 \end{bmatrix} \tag{35.3-9}$$

Use the binary splitting algorithm (section 32.1 on page 611) for the computation of the matrix product $M(k)$.

35.3.3 Miscellaneous relations for continued fractions

35.3.3.1 Determinantal expressions

The *determinant formula* for the numerators and denominators of successive convergents is

$$\det \begin{bmatrix} P_k & Q_k \\ P_{k-1} & Q_{k-1} \end{bmatrix} \;=\; P_k\, Q_{k-1} - P_{k-1}\, Q_k \;=\; (-1)^{k-1} \prod_{j=1}^{k} b_j \tag{35.3-10}$$

The relation is obtained by taking determinants on both sides of equation 35.3-9. The relation can also be written as

$$\frac{P_k}{Q_k} - \frac{P_{k-1}}{Q_{k-1}} \;=\; \frac{(-1)^{k-1} \prod_{j=1}^{k} b_j}{Q_{k-1}\, Q_k} \tag{35.3-11}$$

For simple continued fractions we have $b_j = 1$ so the product in the numerator equals one. Further, by inserting $P_{k-1} = (P_k - b_k\, P_{k-2})/a_k$ (relation 35.3-7a) and the equivalent expression for Q_{k-1} into relation 35.3-10, one obtains

$$\det \begin{bmatrix} P_k & Q_k \\ P_{k-2} & Q_{k-2} \end{bmatrix} \;=\; P_k\, Q_{k-2} - P_{k-2}\, Q_k \;=\; (-1)^{k}\, a_k \prod_{j=1}^{k-1} b_j \tag{35.3-12}$$

Equivalently,

$$\frac{P_k}{Q_k} - \frac{P_{k-2}}{Q_{k-2}} \;=\; \frac{(-1)^{k}\, a_k \prod_{j=1}^{k-1} b_j}{Q_{k-2}\, Q_k} \tag{35.3-13}$$

This relation tells us (provided all a_j and b_j are positive) that the sequence of even convergents is increasing and the sequence of odd convergents is decreasing. As both converge to a common limit we have $P_o/Q_o \geq P_e/Q_e$ for all even e and odd o. Equality can occur only for terminating continued fractions.

35.3.3.2 Subsequences of convergents

Sometimes the terms a_k, b_k of the continued fraction are given in the form "$a_k = u(k)$ *if* k *even,* $a_k = v(k)$ *else*" (and b_k equivalently). Then one might want to compute the $x = K(a, b)$ in a 'stride two' manner as

$$P_k \;=\; A_k\, P_{k-2} + B_k\, P_{k-4} \tag{35.3-14a}$$
$$Q_k \;=\; A_k\, Q_{k-2} + B_k\, Q_{k-4} \tag{35.3-14b}$$

in order to regularize the involved expressions. We write the recurrence relation three times

$$\begin{aligned} P_k &= a_k\,P_{k-1} + b_k\,P_{k-2} && \text{(35.3-15a)}\\ P_{k-1} &= a_{k-1}\,P_{k-2} + b_{k-1}\,P_{k-3} && \text{(35.3-15b)}\\ P_{k-2} &= a_{k-2}\,P_{k-3} + b_{k-2}\,P_{k-4} && \text{(35.3-15c)} \end{aligned}$$

and eliminate the terms P_{k-1} and P_{k-3}. This gives

$$\begin{aligned} A_k &= \frac{a_k\,b_{k-1} + b_k\,a_{k-2} + a_k\,a_{k-1}\,a_{k-2}}{a_{k-2}} = \frac{a_k\,b_{k-1}}{a_{k-2}} + b_k + a_k\,a_{k-1} && \text{(35.3-16a)}\\ B_k &= \frac{-a_k\,b_{k-1}\,b_{k-2}}{a_{k-2}} && \text{(35.3-16b)} \end{aligned}$$

The stride three version

$$\begin{aligned} P_k &= A_k\,P_{k-3} + B_k\,P_{k-6} && \text{(35.3-17a)}\\ Q_k &= A_k\,Q_{k-3} + B_k\,Q_{k-6} && \text{(35.3-17b)} \end{aligned}$$

leads to the expressions (writing a_n for a_{k-n} to reduce line width):

$$\begin{aligned} A_k &= \frac{a_0b_1b_3 + b_0a_2b_3 + b_0b_2a_4 + a_0a_1a_2b_3 + a_0a_1b_2a_4 + a_0b_1a_3a_4 + b_0a_2a_3a_4 + a_0a_1a_2a_3a_4}{b_3 + a_3a_4} && \text{(35.3-18a)}\\ B_k &= \frac{b_0b_2b_3b_4 + a_0a_1b_2b_3b_4}{b_3 + a_3a_4} && \text{(35.3-18b)} \end{aligned}$$

When setting $a_k = \alpha$, $b_k = \beta$ the expressions for A_k and B_k simplify to the coefficients in relations 34.1-13c on page 641 (stride two) and 34.1-13d (stride three) for recurrences.

35.3.3.3 Relation to alternating series

Using relation 35.3-11 on the preceding page it is possible to rewrite a continued fraction $x = K(a, b)$ with positive a_k, b_k as an alternating series

$$\begin{aligned} x &= a_0 + \sum_{k=1}^{\infty} (-1)^{k+1}\,s_k && \text{(35.3-19)}\\ &= a_0 + \frac{b_1}{Q_0\,Q_1} - \frac{b_1\,b_2}{Q_1\,Q_2} + \frac{b_1\,b_2\,b_3}{Q_2\,Q_3} \pm \ldots + (-1)^{k+1}\,\frac{\prod_{i=1}^{k} b_i}{Q_k\,Q_{k+1}} \pm \ldots \end{aligned}$$

Thereby the algorithm for the accelerated summation of alternating series from section 32.3 can be applied to compute x.

35.3.3.4 Continued fractions from infinite products

A continued fraction for the product

$$P := \prod_{k=0}^{\infty} (1 + Y_k) \qquad \text{(35.3-20a)}$$

in terms of $a = [a_0, a_1, \ldots]$ and $b = [b_0, b_1, \ldots]$ is

$$\begin{aligned} a &= [1, \quad 1, \quad +Y_1 + (1 + Y_1)\cdot Y_0, \quad +Y_2 + (1 + Y_2)\cdot Y_1, \quad +Y_3 + (1 + Y_3)\cdot Y_2, \ \ldots\,] && \text{(35.3-20b)}\\ b &= [1, \quad +Y_0, \quad -1\cdot Y_1\cdot(1 + Y_0), \quad -Y_0\cdot Y_2\cdot(1 + Y_1), \quad -Y_1\cdot Y_3\cdot(1 + Y_2), \quad \ldots\,] && \text{(35.3-20c)} \end{aligned}$$

For a given a vector $y = [Y_0, Y_1, \ldots]$ the computation of individual values a_k and b_k can be implemented as:

```
Y(k) = eval(Str("Y" k))  \\ return symbol Yk
yprod(n)= if (n<=0, 1, prod(j=0, n-1, (1+Y(j))))
n=3
pr = yprod(n)
    ((Y2 + 1)*Y1 + (Y2 + 1))*Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))
yv = vector(n, j, Y(j-1) )
    [Y0, Y1, Y2]
t = cfprod(yv);
a=t[1]
    [1, 1, (Y1 + 1)*Y0 + Y1, (Y2 + 1)*Y1 + Y2]
b=t[2]
    [1, Y0, -Y1*Y0 - Y1, (-Y2*Y1 - Y2)*Y0]
{ for(k=0, n,
    t=cfab2pq(a,b, k);
    p=t[1]; q=t[2];
    print1(k, ":  (",p, ") / (", q,")");
    yp = yprod(k);
    print1("\n   == ", simplify(p/q));
    print();
  ); }
    0:  (1) / (1)
       == 1
    1:  (Y0 + 1) / (1)  \\  (p1) / (q1)
       == Y0 + 1        \\  == yprod(1)
    2:  ((Y1 + 1)*Y0^2 + (Y1 + 1)*Y0) / (Y0)  \\  (p2) / (q2)
       == (Y1 + 1)*Y0 + (Y1 + 1)              \\ == yprod(2) == (1+Y0)*(1+Y1)
    3:  (((Y2 + 1)*Y1^2 + (Y2 + 1)*Y1)*Y0^2 + ((Y2 + 1)*Y1^2 + (Y2 + 1)*Y1)*Y0) / (Y1*Y0)
       == ((Y2 + 1)*Y1 + (Y2 + 1))*Y0 + ((Y2 + 1)*Y1 + (Y2 + 1))
```

Figure 35.3-D: Verification of relations 35.3-20b and 35.3-20c on the preceding page using pari/gp.

```
cfproda(yv, n)=
{
    local( y2, y3, d );
    if ( n<=1, return(1) );
    y3 = yv[n];
    y2 = yv[n-1];
    return( (y2+y3*(1+y2)) );
}

cfprodb(yv, n)=
{
    local( y1, y2, y3 );
    if (0==n, return(1) );  \\ unused
    if (1==n, return(+yv[1+0]) );
    y3 = yv[n];
    y2 = yv[n-1];
    y1 = if ( n==2, 1, yv[n-2] );
    return( -y1*y3*(1+y2) );
}
```

The routine `cfprod()` generates the vectors a and b with $n+1$ terms where n is the length of y:

```
cfprod(yv)=
{
    local(n, a, b);
    n = length(yv);
    n += 1;  \\ n+1 terms in continued fraction
    a = vector(n);
    b = vector(n);
    for (k=0, n-1,
        a[k+1] = cfproda(yv, k);
        b[k+1] = cfprodb(yv, k);
    );
    return( [a, b] );
}
```

Relations 35.3-20b and 35.3-20c can be verified using pari/gp as shown in figure 35.3-D.

35.3.3.5 An expression for a sum of products

Define Z_n as

$$Z_n \quad := \quad z_1 + z_1\,z_2 + z_1\,z_2\,z_3 + z_1\,z_2\,z_3\,z_4 + \ldots \;=\; \sum_{k=1}^{n} \prod_{i=1}^{k} z_i \tag{35.3-21}$$

Then Z_∞ has the continued fraction

$$Z_\infty \quad = \quad \cfrac{z_1}{1 - \cfrac{z_2}{1 + z_2 - \cfrac{z_3}{1 + z_3 - \cfrac{z_4}{1 + z_4 - \cfrac{z_5}{1 + z_5 - \cdots}}}}} \tag{35.3-22}$$

That is, $Z = K(a, b)$ where

$$a \quad = \quad [0,\ 1,\ z_2 + 1,\ z_3 + 1,\ z_4 + 1,\ z_5 + 1,\ z_6 + 1,\ \ldots] \tag{35.3-23a}$$

$$b \quad = \quad [1,\ z_1,\ -z_2,\quad -z_3,\quad -z_4,\quad -z_5,\quad -z_6,\quad \ldots] \tag{35.3-23b}$$

For the n-th convergent P_n/Q_n one has $Q_n = 1$ and $P_n = Z_n$.

To convert the power series of a hypergeometric function (see section 35.2 on page 663)

$$F\left(\begin{matrix} a_1,\, a_2,\, \ldots,\, a_u \\ b_1,\, b_2,\, \ldots,\, b_v \end{matrix}\,\middle|\, z\right) \tag{35.3-24}$$

into a continued fraction, set $z_1 = 1$, and for $k \geq 1$ set

$$z_{k+1} \quad = \quad \frac{z \prod_{j=1}^{u} (a_j + k)}{k \prod_{j=1}^{v} (b_j + k)} \tag{35.3-25}$$

An implementation is

```
hyper2cf(va, vb, n, z='z)=
\\ convert hypergeom(va,vb,z) to continued fraction
{
    local(cfa, cfb, m);
    n += 2;
    cfa = vector(n);
    cfb = vector(n);
    cfa[1] = 0;    cfa[2] = 1;
    cfb[1] = 1;    cfb[2] = 1;
    for (k=3, n,
        m = 1/(k-2);  \\ hidden lower parameter 1:  (n-2) == 1+(n-3)
        m *= prod(j=1, #va, va[j]+(k-3));  \\ upper parameters
        m /= prod(j=1, #vb, vb[j]+(k-3));  \\ lower parameters
        m *= z;  \\ argument
        cfa[k]=(m+1);
        cfb[k]=-m;
    );
    return( [cfa, cfb] );
}
```

We convert $\log(1-z)/z = F\left(\begin{matrix} 1, 1 \\ 2 \end{matrix}\,\middle|\, z\right)$ to a continued fraction and check the result:

```
? N=7;
? va=[1,1];vb=[2];
? t=hyper2cf(va,vb,N);
? cfa=t[1]
  [0, 1, 1/2*z + 1, 2/3*z + 1, 3/4*z + 1, 4/5*z + 1, 5/6*z + 1, 6/7*z + 1, 7/8*z + 1]
? cfb=t[2]
```

```
  [1, 1, -1/2*z, -2/3*z, -3/4*z, -4/5*z, -5/6*z, -6/7*z, -7/8*z]
? t=cfab2pq(cfa,cfb)
  [1/8*z^7 + 1/7*z^6 + 1/6*z^5 + 1/5*z^4 + 1/4*z^3 + 1/3*z^2 + 1/2*z + 1, 1]
? s1=t[1]/t[2]+O(z^N)
  1 + 1/2*z + 1/3*z^2 + 1/4*z^3 + 1/5*z^4 + 1/6*z^5 + 1/7*z^6 + O(z^7)
? s2=hypergeom(va,vb,z,N)+O(z^N)
  1 + 1/2*z + 1/3*z^2 + 1/4*z^3 + 1/5*z^4 + 1/6*z^5 + 1/7*z^6 + O(z^7)
```

For further information on continued fractions see [187] and [124]. An in depth treatment is [172].

Chapter 36

Synthetic Iterations *

It is easy to construct arbitrary many iterations that converge super-linearly. Guided by some special constants that in base 2 can be obtained by recursive constructions we build iterations that allow the computation of the constant in a base independent manner. The iterations lead to functions that typically cannot be identified in terms of known (named) functions. Some of the functions can be expressed as infinite sums or products. For the constructions with repeated string substitutions see chapter 16.

36.1 A variation of the iteration for the inverse

We start with the product form for the most simple iteration, the one for the inverse:

$$\begin{aligned} I(y) &:= \frac{1}{1-y} && \text{(36.1-1a)}\\ &= 1 + y + y^2 + y^3 + y^4 + \ldots && \text{(36.1-1b)}\\ &= (1+y)\,(1+y^2)\,(1+y^4)\,(1+y^8)\,\ldots\,(1+y^{2^k})\,\ldots && \text{(36.1-1c)}\\ &= (1+Y_0)\,(1+Y_1)\,(1+Y_2)\,(1+Y_3)\,\ldots\,(1+Y_k)\,\ldots && \text{(36.1-1d)}\\ &\quad \text{where} \quad Y_0 = y,\ Y_{k+1} = Y_k^2 \end{aligned}$$

We now modify the signs in the infinite product:

$$\begin{aligned} J(y) &:= (1-y)\,(1-y^2)\,(1-y^4)\,(1-y^8)\,\ldots\,(1-y^{2^k})\,\ldots && \text{(36.1-2a)}\\ &= 1 - y - y^2 + y^3 - y^4 + y^5 + y^6 - y^7 - y^8 \pm \ldots && \text{(36.1-2b)}\\ &= (1-Y_0)\,(1-Y_1)\,(1-Y_2)\,(1-Y_3)\,\ldots\,(1-Y_k)\,\ldots && \text{(36.1-2c)}\\ &\quad \text{where} \quad Y_0 = y,\ Y_{k+1} = Y_k^2 \end{aligned}$$

The value of the n-th coefficient equals plus one if the parity of n is zero, else minus one (sequence A106400 of [214], the *Thue-Morse sequence*). The function J can be implemented as

```
fj(y,N=5)=
{
    local(r);
    r = 1;
    for (k=1, N,
        r -= r*y;
        y *= y;
    );
    return(r);
}
```

Replacing the minus by a plus gives the implementation for the function I.

A related constant is the *parity number* (or *Prouhet-Thue-Morse constant*):

$$\begin{aligned} P &= 0.412454033640107597783361368258455283089478374455769557 5\ldots \qquad (36.1\text{-}3)\\ \text{[base 2]} &= 0.0110,1001,1001,0110,1001,0110,0110,1001,1001,0110,0110,1001,\ldots\\ \text{[base 16]} &= 0.6996,9669,9669,6996,9669,6996,6996,9669,9669,6996,6996,9669,\ldots\\ \text{[CF]} &= [0,2,2,2,1,4,3,5,2,1,4,2,1,5,44,1,4,1,2,4,1,1,1,5,14,1,50,15,5,1,1,1,4,2,1,\ldots] \end{aligned}$$

The sequence of zeros and ones in the binary expansions is entry A010060 of [214]. The constant P can be computed defining

$$K(y) = \frac{[I(y)-J(y)]}{2} = y+y^2+y^4+y^7+y^8+y^{11}+y^{13}+y^{14}+y^{16}+\ldots \qquad (36.1\text{-}4)$$

Then

$$P = \frac{1}{2}K\left(\frac{1}{2}\right) = \frac{1}{2}\frac{\left[I(\frac{1}{2})-J(\frac{1}{2})\right]}{2} = \frac{1}{2}-\frac{1}{4}J\left(\frac{1}{2}\right) \qquad (36.1\text{-}5)$$

Thereby (see also [30], item 125),

$$2-4\,P = J\left(\frac{1}{2}\right) = \prod_{k=0}^{\infty}\left(1-\frac{1}{2^{2^k}}\right) = 0.350183865439569608866554526966178\ldots \qquad (36.1\text{-}6)$$

The sequence of bits of the parity number can also be obtained by starting with a single 0 and repeated application of the substitution rules $0 \to 01$ and $1 \to 10$.

The following relations are direct consequences of the definitions of the functions I and J:

$$I(y)\,I(-y) = I(y^2) \qquad (36.1\text{-}7a)$$

$$I(y) = \frac{J(y^2)}{J(y)} \qquad (36.1\text{-}7b)$$

$$I(-y) = \frac{1-y}{1+y}\,I(y) \qquad (36.1\text{-}7c)$$

$$J(-y) = \frac{1+y}{1-y}\,J(y) \qquad (36.1\text{-}7d)$$

We have

$$I(y) = 1+\sum_{k=0}^{\infty}\left[y^{2^k}\prod_{j=0}^{k-1}\left(1+y^{2^j}\right)\right] \qquad (36.1\text{-}8a)$$

$$J(y) = 1-\sum_{k=0}^{\infty}\left[y^{2^k}\prod_{j=0}^{k-1}\left(1-y^{2^j}\right)\right] \qquad (36.1\text{-}8b)$$

A functional equation for K is

$$K(y) = (1-y)\,K(y^2)+\frac{y}{1-y^2} \qquad (36.1\text{-}9)$$

It is solved by

$$K(y) = \sum_{k=0}^{\infty}\left[\frac{y^{2^k}}{1-y^{2^{k+1}}}\prod_{j=0}^{k-1}\left(1-y^{2^j}\right)\right] \qquad (36.1\text{-}10)$$

For the inverse of J we have

$$\frac{1}{J(y)} = 1 + y + 2y^2 + 2y^3 + 4y^4 + 4y^5 + 6y^6 + 6y^7 + 10y^8 + 10y^9 + 14y^{10} + \ldots \quad (36.1\text{-}11a)$$

$$= \left[(1-y)\,(1-y^2)\,(1-y^4)\,(1-y^8)\,\ldots\right]^{-1} = \prod_{k=0}^{\infty} I(y^{2^k}) \quad (36.1\text{-}11b)$$

$$= (1-y)\prod_{k=0}^{\infty}\frac{1+y^{2^k}}{1-y^{2^k}} \quad (36.1\text{-}11c)$$

$$= (1+y)\,(1+y^2)^2\,(1+y^4)^3\,(1+y^8)^4\,(1+y^{16})^5\,\ldots\,(1+y^{2^k})^{k+1}\,\ldots \quad (36.1\text{-}11d)$$

The sequence of coefficients of the even powers of x in relation 36.1-11a is

```
1, 2, 4, 6, 10, 14, 20, 26, 36, 46, 60, 74, 94, 114, 140, 166, 202, ...
```

This is entry A000123 of [214], the number of binary partitions of the even numbers. The sequence $\frac{1}{2}\,[2, 4, 6, 10, 14, \ldots]$ modulo two equals the period-doubling sequence, see section 36.5 on page 700.

Relation 36.1-11d tells us that the number of partitions of n into parts 2^k equals the number of partitions of n into at most one 1, two 2, three 4, ..., $k+1$ parts 2^k. The same relation can be used for a divisionless algorithm for the computation of $1/J$:

```
binpart(y,N=5)=
{
    local(r);
    r = 1;
    for (k=1, N,
        for (j=1, k,  r += r*y; );
        y *= y;
    );
    return(r);
}
```

The generating function $1 + 2\,y + 4\,y^2 + 6\,y^3 + 10\,y^4 + 14\,y^5 + 20\,y^6 + \ldots$ equals

$$\frac{I(y)}{J(y)} = (1+y)^2\,(1+y^2)^3\,(1+y^4)^4\,(1+y^8)^5\,(1+y^{16})^6\,\ldots\,(1+y^{2^k})^{k+2}\,\ldots \quad (36.1\text{-}12)$$

It can be computed via (note the change in the inner loop)

```
binpart2(y,N=5)=
{
    local(r);
    r = 1;
    for (k=1, N,
        for (j=1, k+1,  r += r*y; );
        y *= y;
    );
    return(r);
}
```

For $I(y)$ we have

$$I(y) = \sum_{k=0}^{\infty}\frac{y^{2^k-1}}{1-y^{2^{k+1}}} = \sum_{k=0}^{\infty}\frac{2^k\,y^{2^k-1}}{1+y^{2^k}} \quad (36.1\text{-}13a)$$

Integration gives (compare with relation 28.2-12 on page 546)

$$-\log(1-y) = \sum_{k=0}^{\infty}\frac{1}{2^{k+1}}\,\log\left(\frac{1+y^{2^k}}{1-y^{2^k}}\right) = \sum_{k=0}^{\infty}\log\left(1+y^{2^k}\right) \quad (36.1\text{-}14a)$$

For the derivative of J we have

$$J'(y) = -J(y)\sum_{k=0}^{\infty}\frac{2^k\,y^{2^k-1}}{1-y^{2^k}} \quad (36.1\text{-}15)$$

The following functional equations hold for $I(y)$:

$$0 = B - 2\,A\,B + A^2 \quad \text{where} \quad A = I(y), \quad B = I(y^2) \tag{36.1-16a}$$
$$0 = B - 2\,A\,B - A^2 + 2\,A^2\,B \quad \text{where} \quad A = I(-y), \quad B = I(-y^2) \tag{36.1-16b}$$
$$0 = B - 3\,A\,B + 3\,A^2\,B - A^3 \quad \text{where} \quad A = I(y), \quad B = I(y^3) \tag{36.1-16c}$$
$$0 = B - 5\,A\,B + 10\,A^2\,B - 10\,A^3\,B + 5\,A^4\,B - A^5 \quad \text{where} \quad A = I(y), \quad B = I(y^5) \tag{36.1-16d}$$
$$0 = B\left[(1-A)^k - (-A)^k\right] + (-A)^k \quad \text{where} \quad A = I(y), \quad B = I(y^k) \tag{36.1-16e}$$

From the functional equation for $K(y)$ (relation 36.1-9), the definition of $K(y)$, and relation 36.1-7b one can derive the following relation for $J(y)$:

$$0 = J_2^3 - 2\,J_4\,J_2\,J_1 + J_4\,J_1^2 \quad \text{where} \quad J_1 = J(y), \quad J_2 = J(y^2), \quad J_4 = J(y^4) \tag{36.1-17}$$

This relation is given in entry A106400 of [214], together with

$$0 = J_6\,J_1^3 - 3\,J_6\,J_2\,J_1^2 + 3\,J_6\,J_2^2\,J_1 - J_3\,J_2^3 \tag{36.1-18}$$

where $J_k = J(y^k)$. Relations between J_1, J_2, J_k, and J_{2k} can be derived from relation 36.1-16e by replacing $I(y)$ by $J(y^2)/J(y)$. For example, $k = 5$ gives

$$0 = J_{10}\,J_1^5 - 5\,J_{10}\,J_2\,J_1^4 + 10\,J_{10}\,J_2^2\,J_1^3 - 10\,J_{10}\,J_2^3\,J_1^2 + 5\,J_{10}\,J_2^4\,J_1 - J_5\,J_2^5 \tag{36.1-19}$$

The Komornik-Loreti constant

One has $K(1/\beta) = 1$ for

$$\frac{1}{\beta} = 0.55952455849672652513220976515743228583107647896866030 76\ldots \tag{36.1-20a}$$
$$[\text{base } 2] = 0.10001111001111010000000001100000000011010110001000100 10110\ldots$$
$$\beta = 1.78723165018296593301327489033700838533793140296181099 7\ldots \tag{36.1-20b}$$
$$[\text{base } 2] = 1.11001001100010000000001101101111111010010111010110100 00\ldots$$
$$[\text{CF}] = [1, 1, 3, 1, 2, 3, 188, 1, 12, 1, 1, 22, 33, 1, 10, 1, 1, 7, 1, 9, 1, 1, 20, 2, 15, 1, \ldots]$$

The constant β is the smallest real number in the interval $(1, 2)$ so that 1 has a unique expansion of the form $\sum_{n=1}^{\infty} \delta_n\,\beta^{-n}$ where $\delta_n \in \{0, 1\}$. It is called the *Komornik-Loreti constant* (see [8]). The fact that $\delta_n = 1$ exactly where the Thue-Morse sequence equals 1 was used for the computation of β: one solves $K(y) = 1$ for y. The transcendence of β is proved (using the fact that $J(y)$ is transcendental for algebraic y) in [9].

Third order variants

Variations of the third order iteration for the inverse

$$I(y) := \frac{1}{1-y} = 1 + y + y^2 + y^3 + y^4 + \ldots \tag{36.1-21a}$$
$$= (1 + y + y^2)\,(1 + y^3 + y^6)\,(1 + y^9 + y^{18})\,\ldots\,(1 + y^{3^k} + y^{2\cdot 3^k})\,\ldots \tag{36.1-21b}$$
$$= (1 + Y_0 + Y_0^2)\,(1 + Y_1 + Y_1^2)\,(1 + Y_2 + Y_2^2)\,\ldots\,(1 + Y_k + Y_k^2)\,\ldots \quad \text{where} \quad Y_0 = y,\ Y_{k+1} = Y_k^3 \tag{36.1-21c}$$

lead to series related to the base-3 analogue of the parity. The most simple example may be

$$T(y) = (1 + Y_0 - Y_0^2)\,(1 + Y_1 - Y_1^2)\,(1 + Y_2 - Y_2^2)\,\ldots\,(1 + Y_k - Y_k^2)\,\ldots \tag{36.1-22a}$$
$$= 1 + y - y^2 + y^3 + y^4 - y^5 - y^6 - y^7 + y^8 + y^9 + y^{10} - y^{11} + y^{12} \pm \ldots \tag{36.1-22b}$$

The sign of the n-th coefficient is the parity of the number of twos in the radix-3 expansion of n. We have

$$\begin{aligned}
\frac{1}{2}\,(I(y)-T(y)) &= y^2+y^5+y^6+y^7+y^{11}+y^{14}+y^{15}+y^{16}+y^{18}+y^{19}+y^{21}+\ldots && \text{(36.1-23a)}\\
\frac{1}{4}\left(I\left(\frac{1}{2}\right)-T\left(\frac{1}{2}\right)\right) &= 0.152644523625407582531924921475791679311504514871489254 8\ldots && \text{(36.1-23b)}\\
\text{[base 2]} &= 0.00100111000100111011011000100100111000100111011011000 11\ldots\\
\text{[CF]} &= [0,6,1,1,4,2,1,1,2,4,1,1,4,1,4,2,1,1,1,2,1,18,3,24,1,6,1,3,\ldots]
\end{aligned}$$

The sequence of zeros and ones in the binary expansion is entry A064990 of [214], the *Mephisto Waltz sequence*.

36.2 An iteration related to the Thue constant

We construct a sequence of zeros and ones that can be generated by starting with a single 0 and repeated application of the substitution rules $0 \to 111$ and $1 \to 110$. The evolution starting with a single zero is:

```
T0 =  0
T1 =  111
T2 =  110110110
   == 3 times 11.0
T3 =  110110111110110111110110111
   == 3 times 110110.111
T4 =  110110111110110111110110110110111101101111101101110110111110110111110110110
   == 3 times 110110111110110111.110110110
T --> 110110111110110111110110110110111101101111101101110110111110110111110110111...
```

The crucial observation is that is T_n is three time repeated the string $T'_{n-1}.T_{n-2}$ where T'_k consists of the first and second third of T_k. The length of the n-th string is 3^n. Let $T(y)$ be the function whose power series corresponds to the string T_∞:

$$T(y) = 1+y+y^3+y^4+y^6+y^7+y^8+y^9+y^{10}+y^{12}+y^{13}+y^{15}+y^{16}+y^{17}+y^{18}+\ldots \quad \text{(36.2-1)}$$

It can be computed by the iteration

$$\begin{aligned}
L_0 &= 0, \qquad A_0 = 1+y, \qquad B_0 = y^2, \qquad Y_0 = y && \text{(36.2-2a)}\\
R_n &= A_n + y^2\,L_n && \text{(36.2-2b)}\\
L_{n+1} &= A_n + B_n && \text{(36.2-2c)}\\
Y_{n+1} &= Y_n^3 && \text{(36.2-2d)}\\
A_{n+1} &= R_n\,(1+Y_{n+1}) \quad \to T(y) && \text{(36.2-2e)}\\
B_{n+1} &= R_n\,Y_{n+1}^2 && \text{(36.2-2f)}
\end{aligned}$$

The implementation is slightly tricky:

```
th(y, N=5)=
{
    local(L, R, A, B, y2, y3, t);
    /* correct up to order 3^(N+1)-1 */
    L=0;
    A=1+y;  B=y^2; /* R = A.B */
    for(k=1, N,
      /* (L, A.B) --> (A.B,  A.L.A.L . A.L) */
      y2 = y^2;
      R = A + y2*L; /* A.L */
      L = A + B;   /* next L = A.B */
      y3 = y * y2;
      B = R * (y3*y3);
      A = R * (1+y3);    /* next A = A.L.A.L  */
      y = y3;
    );
    return( A + B )
}
```

The *Thue constant* (which should be *Roth's constant*, see entry A014578 of [214]) can be computed as

$$\begin{aligned}
\frac{1}{2}T(\frac{1}{2}) &= 0.859099796854703104903572502841974202614239955559439087 4\ldots && (36.2\text{-}3)\\
[\text{base } 2] &= 0.110, 110, 111, 110, 110, 111, 110, 110, 110, 110, 110, 111, 110, 110, 111, 110, 110, \ldots\\
[\text{base } 8] &= 0.667, 667, 666, 667, 667, 666, 667, 667, 667, 667, 667, 666, 667, 667, 666, 667, 667, \ldots\\
[\text{CF}] &= [0, 1, 6, 10, 3, 2, 513, 1, 1, 2, 1, 4, 2, 6576668769, 1, 1, 4,\\
&\quad 1, 2, 2, 256, 1, 1, 2, 1, 2, 3, 1, 3, 3, 241785163922925834941235 3,\\
&\quad 1, 2, 3, 1, 3, 2, 1, 2, 1, 1, 256, 2, 2, 1, 4, 2, 3288334384,\\
&\quad 1, 1, 4, 1, 2, 2, 146, 2, 3, 3, 2, 1, 2, 1, 12, X, \ldots]
\end{aligned}$$

The term X in the continued fraction has 74 decimal digits. By construction the bits at positions n not divisible by three are one and otherwise the complement of the bit at position $n/3$. As a functional equation (see also section 36.5 on page 700):

$$y\,T(y) + y^3\,T(y^3) = \frac{y}{1-y} \qquad (36.2\text{-}4)$$

From this relation we can obtain a series for $T(y)$:

$$T(y) = \sum_{n=0}^{\infty} (-1)^n \frac{y^{3^n-1}}{1-y^{3^n}} \qquad (36.2\text{-}5)$$

36.3 An iteration related to the Golay-Rudin-Shapiro sequence

Define $Q(y)$ by

$$\begin{aligned}
L_0 &= 1, \qquad R_0 = y, \qquad Y_0 = y && (36.3\text{-}1a)\\
L_{n+1} &= L_n + R_n \quad \to Q(y) && (36.3\text{-}1b)\\
Y_{n+1} &= Y_n^2 && (36.3\text{-}1c)\\
R_{n+1} &= Y_{n+1}\,(L_n - R_n) && (36.3\text{-}1d)
\end{aligned}$$

then

$$Q(y) = 1 + y + y^2 - y^3 + y^4 + y^5 - y^6 + y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} - y^{13} + y^{14} - y^{15} + \ldots \qquad (36.3\text{-}2)$$

The sequence of coefficients is the *Golay-Rudin-Shapiro sequence* (or *GRS sequence*, entry A020985 of [214], see also page 40). The constant

$$\begin{aligned}
Q &= 0.92924386959737885325397664472205076441287553952432552 22\ldots && (36.3\text{-}3)\\
[\text{base } 2] &= 0.1110, 1101, 1110, 0010, 1110, 1101, 0001, 1101, 1110, 1101, 1110, 0010, \ldots\\
[\text{base } 16] &= 0.ede2, ed1d, ede2, 12e2, ede2, ed1d, 121d, ed1d, ede2, ed1d, ede2, 12e2, \ldots\\
[\text{CF}] &= [0, 1, 13, 7, 1, 1, 15, 4, 1, 3, 1, 2, 2, 1000, 12, 2, 1, 6, 1, 1, 1, 1, 1, 1, 8, 2, 1, 1, 2, 4, 1, 1, 3, \ldots]
\end{aligned}$$

can be computed as

$$Q = \frac{1 + \frac{1}{2}\,Q(\frac{1}{2})}{2} \qquad (36.3\text{-}4)$$

The implementation using pari/gp:

```
qq(y, N=8)=
{
    local(L, R, Lp, Rp);
    /* correct up to order 2**(N+1) */
    L=1;  R=y;
    for(k=0,N, Lp=L+R; y*=y; Rp=y*(L-R); L=Lp; R=Rp);
    return( L + R )
}
```

The following functional relations hold for Q:

$$Q(y^2) = \frac{Q(y)+Q(-y)}{2} \tag{36.3-5a}$$

$$Q(y) = Q\left(y^2\right) + y\,Q\left(-y^2\right) \tag{36.3-5b}$$

$$Q(-y) = Q\left(y^2\right) - y\,Q\left(-y^2\right) \tag{36.3-5c}$$

Combining the latter two gives

$$Q(y) = (1+y)\,Q\left(y^4\right) + \left(y^2-y^3\right)\,Q\left(-y^4\right) \tag{36.3-6}$$

Counting zeros and ones in the binary expansion of Q

The number of ones and zeros in the first 4^k bits of the constant Q can be computed via a string-substitution engine (see chapter 16 on page 331). The hexadecimal expansion can be obtained as:

```
    Number of symbols = 4
    Start: e
    Rules:
      e --> ed
      d --> e2
      2 --> 1d
      1 --> 12
    -------------
    0:   (#=1)
      e
    1:   (#=2)
      ed
    2:   (#=4)
      ede2
    3:   (#=8)
      ede2ed1d
    4:   (#=16)
      ede2ed1dede212e2
    5:   (#=32)
      ede2ed1dede212e2ede2ed1d121ded1d
    6:   (#=64)
      ede2ed1dede212e2ede2ed1d121ded1dede2ed1dede212e2121d12e2ede212e2
```

A few lines of pari/gp code count the occurrences of the symbols (and thereby of zeros and ones):

```
/*   e --> ed  ;  d --> e2  ;  2 --> 1d  ;  1 --> 12    */
/*   e  d      ;  e     2   ;     d     1;        2  1; */
mg= [1, 1, 0, 0;  1, 0, 1, 0;  0, 1, 0, 1;  0, 0, 1, 1];
mg=mattranspose(mg)
{ for (k=0, 40,
    print1( k, ":  " );
    mm=mg^k;
    mv = mm*[1,0,0,0]~;
    t =  sum(i=1,4, mv[i]);
    /* e and d have three ones and one zero */
    /* 1 and 2 have one one and three zeros */
    n0 = 3*(mv[3]+mv[4]) + (mv[1]+mv[1]); /* # of zeros */
    n1 = 3*(mv[1]+mv[2]) + (mv[3]+mv[4]); /* # of ones  */
    print( t, "   ",  mv~,
     "   #0=",  n0, " #1=", n1, "  diff=", n1-n0, "  #1/#0=", 1.0*n1/n0 );
 ) }
```

We obtain the data shown in figure 36.3-A. The sequence of the numbers of ones is entry A005418 of [214]. It is identical to the sequence of numbers of equivalence classes obtained by identifying bit-strings that are mutual reverses or complements, see section 3.8.1.6 on page 130.

```
                #e   #d   #2   #1
   0:      1   [  1,   0,   0,   0]   #0=   2   #1=   3  diff= 1  #1/#0=1.5000000
   1:      2   [  1,   1,   0,   0]   #0=   2   #1=   6  diff= 4  #1/#0=3.0000000
   2:      4   [  2,   1,   1,   0]   #0=   7   #1=  10  diff= 3  #1/#0=1.4285714
   3:      8   [  3,   3,   1,   1]   #0=  12   #1=  20  diff= 8  #1/#0=1.6666666
   4:     16   [  6,   4,   4,   2]   #0=  30   #1=  36  diff= 6  #1/#0=1.2000000
   5:     32   [ 10,  10,   6,   6]   #0=  56   #1=  72  diff=16  #1/#0=1.2857142
   6:     64   [ 20,  16,  16,  12]   #0= 124   #1= 136  diff=12  #1/#0=1.0967741
   7:    128   [ 36,  36,  28,  28]   #0= 240   #1= 272  diff=32  #1/#0=1.1333333
   8:    256   [ 72,  64,  64,  56]   #0= 504   #1= 528  diff=24  #1/#0=1.0476190
   9:    512   [136, 136, 120, 120]   #0= 992   #1=1056  diff=64  #1/#0=1.0645161
  10:   1024   [272, 256, 256, 240]   #0=2032   #1=2080  diff=48  #1/#0=1.0236220
    [--snip--]
  40: 1099511627776   [274878431232, 274877906944, 274877906944, 274877382656]  \
         #0=2199022731264 #1=2199024304128  diff=1572864  #1/#0=1.00000071525
```

Figure 36.3-A: Number of symbols, zeros and ones with the n-th step of the string substitution engine for the GRS sequence. For long strings the ratio of the number of zeros and ones approaches one.

36.4 Iterations related to the ruler function

The *ruler function* $r(n)$ can be defined to be the highest exponent e so that 2^e divides n. Here we consider the function that equals $r(n)+1$ for $n \neq 0$ and zero for $n = 0$. The partial sequences up to indices $2^n - 1$ are

```
R0 = 0
R1 = 0 1
R2 = 0 1 2 1
R3 = 0 1 2 1 3 1 2 1
R4 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
R5 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
R6 = 0 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6 1 ...
```

The limiting sequence is entry A001511 of [214]. Observe that $R_n = R_{n-1}.(R_{n-1} + [n, 0, 0, \ldots, 0])$. Define the function $R(y)$ as the limit of the iteration

$$R_1 = y, \qquad Y_1 = y \tag{36.4-1a}$$

$$Y_{n+1} = Y_n^2 \tag{36.4-1b}$$

$$R_{n+1} = R_n + Y_{n+1}\,[R_n + (1+n)] \quad \to R(y) \tag{36.4-1c}$$

Implementation in pari/gp:

```
r2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A);
    A=y;
    for(k=2, N,  y *= y;  A += y*(A + k); );
    return( A );
}
```

If one replaces in the statement `A += y*(A + k);` by `A += y*(A + 1);` then the iteration computes $\frac{y}{1-y}$. For the function R we have

$$R\left(\frac{1}{q^2}\right) = \frac{1}{2q}\left[(q-1)\,R\left(\frac{1}{q}\right) + (q+1)\,R\left(-\frac{1}{q}\right)\right] \tag{36.4-2a}$$

$$R\left(\frac{1}{q}\right) = R\left(\frac{1}{q^2}\right) + \frac{1}{q-1} \tag{36.4-2b}$$

$$R(y) = R(y^2) + \frac{y}{1-y} = R(y^4) + \frac{y}{1-y} + \frac{y^2}{1-y^2} = \ldots \tag{36.4-2c}$$

and so

$$R(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1-y^{2^n}} \tag{36.4-3}$$

One further has

$$R(y) = \sum_{n=0}^{\infty} (1+n)\,\frac{y^{2^n}}{1-y^{2^{n+1}}} \tag{36.4-4}$$

Define the *ruler constant* as $R := R(1/2)/2$, then

$$\begin{aligned} R &= 0.701968413941089160288103037068604677268819380760945033 7\ldots \\ [\text{base } 2] &= 0.101100111011010000110011101101001011001110110100001100 11\ldots \\ [\text{CF}] &= [0,1,2,2,1,4,2,1,1,1,2,2,1,3,3,4,5,6,1,5,1,1,9,49,1,8,1,1,5,1, \\ &\quad 6,5,1,3,3,1,2,4,3,1,2,4,2,1,1,3,1,9,1,11,18,2,4,5,1,3,2,25,9, \\ &\quad 2,3,1,2,3,1,9,1,2,8,1,3,4,1,1,1,1,31,1,1,6,1,13,1,1,14,1,6,1,\ldots] \end{aligned} \tag{36.4-5}$$

We will now compute the function

$$P(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1+y^{2^n}} = \sum_{n=0}^{\infty} (1-n)\,\frac{y^{2^n}}{1-y^{2^{n+1}}} \tag{36.4-6}$$

The partial sequences of coefficients of the Taylor expansion are

```
P0 = 0
P1 = 0 1
P2 = 0 1 0 1
P3 = 0 1 0 1 -1 1 0 1
P4 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1
P5 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -3 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1
P6 = 0 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -3 1 0 1 -1 1 0 1 -2 1 0 1 -1 1 0 1 -4 1...
```

Observe that $P_n = P_{n-1}.(P_{n-1} - [n-2, 0, 0, \ldots, 0])$. Compute $P(y)$ by

$$P_1 = y, \qquad Y_1 = y \tag{36.4-7a}$$

$$Y_{n+1} = Y_n^2 \tag{36.4-7b}$$

$$P_{n+1} = P_n + Y_{n+1}\,[P_n + (1-n)] \quad \to P(y) \tag{36.4-7c}$$

Implementation in pari/gp:

```
p2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A);
    A=y;
    for(k=2, N,  y *= y;  A += y*(A - (k-2)); );
    return( A );
}
```

One finds for P:

$$P\left(\frac{1}{q^2}\right) = \frac{1}{2q}\left[(q+1)\,P\left(\frac{1}{q}\right) + (q-1)\,P\left(-\frac{1}{q}\right)\right] \tag{36.4-8a}$$

$$P\left(\frac{1}{q}\right) = P\left(\frac{1}{q^2}\right) + \frac{1}{q+1} \tag{36.4-8b}$$

Relations that involve both P and R are

$$P\left(\frac{1}{q^2}\right) = \frac{1}{2}\left[\frac{q+3}{q+1}\,P\left(\frac{1}{q}\right) - \frac{q-1}{q+1}\,R\left(\frac{1}{q}\right)\right] \tag{36.4-8c}$$

$$R\left(\frac{1}{q}\right) = \frac{1}{2}\left[(q+1)\,P\left(\frac{1}{q^2}\right) + (q+3)\,R\left(\frac{1}{q^2}\right)\right] \tag{36.4-8d}$$

$$P\left(\frac{1}{q}\right) = \frac{1}{2}\left[(q+1)\,P\left(\frac{1}{q^2}\right) + (q-1)\,R\left(\frac{1}{q^2}\right)\right] \tag{36.4-8e}$$

$$\frac{R(y)+P(y)}{2} = \frac{y}{1-y} = \sum_{n=1}^{\infty} y^n \tag{36.4-8f}$$

$$\frac{R(y)-P(y)}{2} = R(y^2) \tag{36.4-8g}$$

$$R^2(y) - P^2(y) = \frac{4\,y}{1-y}\,R(y^2) \tag{36.4-8h}$$

36.5 An iteration related to the period-doubling sequence

Define

$$T(y) = \sum_{n=0}^{\infty} \frac{y^{2^n}}{1+(-1)^n\, y^{2^n}} = \sum_{n=0}^{\infty} (-1)^n \frac{y^{2^n}}{1-y^{2^n}} \quad (36.5\text{-}1a)$$

$$= y + y^3 + y^4 + y^5 + y^7 + y^9 + y^{11} + y^{12} + y^{13} + y^{15} + y^{16} + y^{17} + \ldots \quad (36.5\text{-}1b)$$

The function can be computed by the iteration

$$A_1 = 0, \qquad L_1 = y, \qquad R_1 = y, \qquad Y_1 = y \quad (36.5\text{-}2a)$$

$$A_{n+1} = L_n + Y_n\, R_n \quad \to T(y) \quad (36.5\text{-}2b)$$

$$L_{n+1} = L_n + Y_n\, A_n \quad (36.5\text{-}2c)$$

$$R_{n+1} = R_n + Y_n\, A_n \quad (36.5\text{-}2d)$$

$$Y_{n+1} = Y_n^2 \quad (36.5\text{-}2e)$$

Implementation in pari/gp:

```
t2(y, N=11)=
{ /* correct to order = 2^N-1 */
    local(A, L, R, t);
    A=0;  L=y;  R=y;
    for(k=2, N,
        t = y*A;
        A = L + y*R;
        L += t;  R += t;
        y *= y;
    );
    return( A );
}
```

The computed Taylor series can be obtained symbolically by starting with a single zero and applying the substitution rules $0 \to 11$ and $1 \to 10$. The evolution is:

```
T0 =  0
T1 =  1 1
T2 =  1 0 1 0
T3 =  1 0 1 1 1 0 1 1
T4 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0
T5 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1
T6 =  1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 ...
```

Observe that $T_n = L(T_{n-1}).T_{n-2}.R(T_{n-1}).T_{n-2}$ where L and R denote the left and right half of their arguments. The limiting sequence is the so-called *period-doubling sequence*. It is entry A035263 of [214] where it is called the *first Feigenbaum symbolic sequence*. Define the *period-doubling constant* as $T := T(1/2)$, then

$$T = 0.729427023494948405709066206894052617060026944465854741 7\ldots \quad (36.5\text{-}3)$$

$$[\text{base } 2] = 0.10111010101110111011101010111010101110101011101110111010\ldots$$

$$[\text{CF}] = [0, 1, 2, 1, 2, 3, 2, 8, 1, 1, 1, 2, 1, 8, 6, 1, 2, 1, 2, 8, 1, 2, 2, 1, 1, 24, 2, 2, 2, 1,$$
$$8, 2, 1, 2, 1, 8, 6, 1, 2, 1, 2, 2, 1, 1, 1, 13, 1, 1, 8, 2, 13, 1, 1, 1, 2, 2, 1, 2, 1,$$
$$6, 8, 1, 2, 1, 2, 8, 1, 2, 2, 1, 1, 24, 2, 2, 2, 1, 8, 2, 1, 2, 1, 11, 1, 9, 2, 1, 116, \ldots]$$

The transcendence of this constant is proved in [146]. A functional equation for $T(y)$ is

$$T(y) + T(y^2) = \frac{y}{1-y} \quad (36.5\text{-}4)$$

The Taylor series of $T(y^2)$ has coefficients one where the series of $T(y)$ has coefficients zero.

```
              pile_0  pile_+  pile_-  moved    summary  direction
                                       disk              of move
         0:    1111    ....    ....    ....    0 0 0 0
         1:    111.    ....    ...1    ...1    0 0 0 -    1
         2:    11..    ..1.    ...1    ..1.    0 0 + -    0
         3:    11..    ..11    ....    ...1    0 0 + +    1
         4:    1...    ..11    .1..    .1..    0 - + +    1
         5:    1..1    ..1.    .1..    ...1    0 - + 0    1
         6:    1..1    ....    .11.    ..1.    0 - - 0    0
         7:    1...    ....    .111    ...1    0 - - -    1
         8:    ....    1...    .111    1...    + - - -    0
         9:    ...1    1..1    .11.    ...1    + - - +    1
        10:    ..1.    1..1    .1..    ..1.    + - 0 +    0
        11:    ..11    1...    .1..    ...1    + - 0 0    1
        12:    ..11    11..    ....    .1..    + + 0 0    1
        13:    ..1.    11..    ...1    ...1    + + 0 -    1
        14:    ....    111.    ...1    ..1.    + + + -    0
        15:    ....    1111    ....    ...1    + + + +    1
```

Figure 36.5-A: Solution of the towers of Hanoi puzzle for 4 disks. The rightmost column corresponds to the direction of the move made, it is the period-doubling sequence.

36.5.1 Connection to the towers of Hanoi puzzle

The *towers of Hanoi* puzzle consists of three piles and n disks of different size. The initial configuration is that all disks are on the leftmost pile ordered by size (smallest on top). The task is to move all disks to the rightmost pile by moving only one disk at a time and never putting a bigger disk on top of a smaller one.

The puzzle with n disks can be solved in $2^n - 1$ steps. Figure 36.5-A shows the solution for $n = 4$ [FXT: bits/hanoi-demo.cc]. Here the piles are represented as binary words. Note that with each move the lowest bit in one of the three words is moved to another word where it is again the lowest bit.

A simple solution can be obtained by observing that the disk moved with step $k = 1, \ldots, 2^n - 1$ corresponds to the lowest set bit in the binary representation of k and the index of the untouched pile changes by $+1$ mod 3 for n even and -1 mod 3 for n odd. The essential part of the implementation is

```
void
hanoi(ulong n)
{
    ulong f[3];
    f[0] = first_comb(n);  f[1] = 0;  f[2] = 0;  // Initial configuration

    const int dr = (n&1 ? -1 : +1);  // == +1 (if n even),  else  == -1

    // PRINT configuration
    int u;  // index of tower untouched in current move
    if ( dr<0 ) u=2;  else u=1;

    ulong n2 = 1UL<<n;
    for (ulong k=1; k<n2; ++k)
    {
        ulong s = lowest_bit(k);

        ulong j = 3;  while ( j-- )  f[j] ^= s;  // change all piles
        f[u] ^= s;  // undo change for untouched pile

        u += dr;
        if ( u<0 ) u=2;  else if ( u>2 ) u=0;  // modulo 3

        // PRINT configuration
    }
}
```

Now with each step the transferred disk is moved by $+1$ or -1 position (modulo 3). The rightmost column in figure 36.5-A consists of zeros and ones corresponding to the direction of the move. It is the period-doubling sequence. A recursive algorithm for the towers of Hanoi puzzle can be given as [FXT: comb/hanoi-rec-demo.cc]:

```
ulong f[3];  // the three piles
void hanoi(int k, ulong A, ulong B, ulong C)
// Move k disks from pile A to pile C
{
    if ( k==0 ) return;
```

```
    // 1. move k-1 disks from pile A to pile B:
    hanoi(k-1, A, C, B);

    // 2. move disk k from pile A to pile C:
    ulong b = 1UL << (k-1);
    f[A] ^= b;
    f[C] ^= b;

    print_hanoi(b); // visit state

    // 3. move k-1 disks from pile B to pile C:
    hanoi(k-1, B, A, C);
}
```

The piles are represented by the binary words `f[A]`, `f[B]`, and `f[C]`, the variable `k` is the number of the disk moved. The routine is called as follows

```
    ulong n = 5;
    // Initial configuration:
    f[0] = first_comb(n);  // n ones as lowest bits
    f[1] = 0;  f[2] = 0;   // empty

    // visit initial state

    hanoi(n-1, 0, 1, 2);  // solve
```

36.5.2 Generalizations of the period-doubling sequence

```
? default(realprecision,85);
? tm(y,m,N=8)=sum(n=0,N,(-1)^n*(y^(m^n)/(1-y^(m^(n)))))
? for(m=2,9,print(m,": ",tm(1.0/10,m,4));print("   ",tm(1.0/10^m,m,4)))
    2: 1.111.1.1.111.111.111.1.1.111.111.111.1.1.111.111.111.1.1.111.111.111.1.1.111.111.111
       .1...1.1.1...1...1...1.1.1...1.1.1...1.1.1...1...1...1.1.1...1.1.1...1.1.1...1...1...

    3: 11.11.11111.11.11111.11.11.11.11.11111.11.11111.11.11.11.11.11111.11.11111.11.11111.1
       ..1..1.....1..1.....1..1..1..1.....1..1.....1..1..1..1.....1..1.....1..1.....1..1.....1.

    4: 111.111.111.1111111.111.111.1111111.111.111.1111111.111.111.111.111.111.111.1111111.1
       ...1...1...1.......1...1...1.......1...1...1.......1...1...1...1...1...1...1.......1.

    5: 1111.1111.1111.1111.111111111.1111.1111.1111.111111111.1111.1111.1111.111111111.1111.
       ....1....1....1....1.........1....1....1....1.........1....1....1....1.........1....1

    6: 11111.11111.11111.11111.11111.1111111111.11111.11111.11111.11111.1111111111.11111.1
       .....1.....1.....1.....1.....1...........1.....1.....1.....1.....1...........1.....1.

    7: 111111.111111.111111.111111.111111.111111.1111111111111.111111.111111.111111.111111.1
       ......1......1......1......1......1......1.............1......1......1......1......1.

    8: 1111111.1111111.1111111.1111111.1111111.1111111.1111111.11111111111111.1111111.11111
       .......1.......1.......1.......1.......1.......1.......1...............1.......1.....

    9: 11111111.11111111.11111111.11111111.11111111.11111111.11111111.11111111.111111111111
       ........1........1........1........1........1........1........1........1............
```

Figure 36.5-B: Visual verification of the higher order analogues of the period-doubling sequence. In the output lines the leading '0.' was removed and all zeros were replaced by dots.

The functional equation for the period-doubling sequence, relation 36.5-4 can be generalized in several ways. For example, one can look for a function for which $F_3(y) + F_3(y^3) = \frac{y}{1-y}$. It is given by

$$F(y) = \sum_{k=0}^{\infty} (-1)^n \frac{y^{3^n}}{1-y^{3^n}} \tag{36.5-5a}$$

$$= z + z^2 + z^4 + z^5 + z^7 + z^8 + z^9 + z^{10} + z^{11} + z^{13} + z^{14} + z^{16} + z^{17} + \dots \tag{36.5-5b}$$

We can compute a constant:

$$F_3(1/2) = 0.859099796854703104903572502841974202614239955559439087 4\dots \tag{36.5-6}$$

But this is just the Thue constant, see section 36.2 on page 695. Large terms occur in the continued fraction expansion of this constant. Even larger terms occur in the continued fractions of $F_m(1/2)$ for $m > 3$ (replace 3 by m in relation 36.5-5a). For example, the 45-th term of $F_5(1/2)$ has 565 digits. In

contrast, the greatest of the first 1630 terms of the continued fraction of $F_2(1/2) = T(1/2)$ equals 288. Some sequences corresponding to the higher order analogues of the period-doubling sequence are shown in figure 36.5-B.

A different way to generalize is to search functions for which, for example, the following functional equation holds:

$$F(y) + F(y^2) + F(y^3) \quad = \quad \frac{y}{1-y} \tag{36.5-7}$$

The equation can be solved by writing $F(y) = y/(1-y) - F(y^2) - F(y^3)$ and using recursions that terminate when a prescribed order is reached.

```
F(z, R)=
{/* solve F(y) + F(y^2) + F(y^3) = y/(1-y) */
    local(s, y);
    y = z + R;
    s = y/(1-y);
    if ( y^2!=R,  s -= F(z^2, R) );
    if ( y^3!=R,  s -= F(z^3, R) );
    return(s);
}
```

We obtain

```
? N=55;
? default(seriesprecision,N);
? s=F(y,O(y^N))
  y + y^4 + y^5 + y^6 + y^7 + y^9 + y^11 - y^12 + y^13 + y^16 + y^17 -
  y^18 + y^19 + y^20 + y^23 + 2*y^24 + y^25 + y^28 + y^29 + y^30 + y^31 +
  y^35 + 3*y^36 + y^37 + y^41 + y^42 + y^43 + y^44 + y^45 + y^47 -
  2*y^48 + y^49 + y^52 + y^53 + 2*y^54 + O(y^55)
```

To verify that the function F actually satisfies the given functional equation we show the sequences of coefficients of the power series of $F(y)$, $F(y^2)$, $F(y^3)$ and their sum:

```
  F(y)=   [0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1,-1, 1, 0, 0, 1, 1,-1, 1, 1, 0, 0, 1, 2, 1, ...]
  F(y^2)= [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0,-1, 0, ...]
  F(y^3)= [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, ...]
  sum=    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]
```

$F(y) = y + y^4 + y^5 + y^7 + y^9 + y^{11} + y^{13} + y^{16} + y^{17} + y^{19} + y^{20} + y^{23} + y^{25} + y^{28} + y^{29} + \ldots$

```
F(y)=   .1..11.1.1.1.1..11.11..1.1..11.1...111...1.111.1.1..11.1...1.1.111.11..1.1
F(y^2)= ..1.....1.1...1...1...1...1.....1.1...1.1.....1...1.....1.1...1.......1.1.
F(y^3)= ...1........1..1.....1.....1.....1.....1........1..1.....1..1........1....
F(y^6)= ......1.................1.....1..........1..........1..........1.......
sum=    .1111111111111111111111111111111111111111111111111111111111111111111111
```

Figure 36.5-C: Visual demonstration that $F(y)$ satisfies the functional equation $F(y)+F(y^2)+F(y^3)+F(y^6) = y/(1-y)$. Dots are used for zeros.

The Taylor series of $F(y)$ where $F(y)+F(y^2)+F(y^3)+F(y^6) = y/(1-y)$ contains only ones and zeros. One has

$$F(y) \quad = \quad \sum_{k=1}^{\infty} R(k)\,\frac{x^k}{1-x^k} \quad \text{where} \quad R(k) \;=\; \begin{cases} (-1)^{e_2+e_3} & \text{if} \quad k = 2^{e_2}\,3^{e_3} \\ 0 & \text{else} \end{cases} \tag{36.5-8}$$

This is a Lambert series, it can be converted into a Taylor series by relation 35.1-20 on page 658. It turns out that with $k = u\,2^{e_2}\,3^{e_3}$ (where neither 2 nor 3 divides u), we have

$$r(k) \quad = \quad \begin{cases} 0 & \text{if either of } e_2 \text{ or } e_3 \text{ is odd} \\ 1 & \text{else} \quad \text{(i.e. both } e_2 \text{ and } e_3 \text{ are even)} \end{cases} \tag{36.5-9}$$

36.6 An iteration from substitution rules with sign

Substitution rules: 0 → L1, 1 → 10, and L → L0.

```
D0 =  1
D1 =  10
D2 =  10L1
D3 =  10L1L010
D4 =  10L1L010L01L10L1
D5 =  10L1L010L01L10L1L01L10L010L1L010
D --> 10L1L010L01L10L1L01L10L010L1L010L01L10L010L1L01L10L1L010L01L...
```

Identify L with -1 and observe that for $n > 1$ one has $D_n = D_{n-1}.(-D_{n-2}).D_{n-2}$. The length of the strings is 2^n. Define $D(y)$ by the iteration

$$\begin{aligned} L_0 &= 1, \qquad R_0 = 1(+0\,y), \qquad Y_0 = y && \text{(36.6-1a)}\\ L_{n+1} &= R_n && \text{(36.6-1b)}\\ R_{n+1} &= R_n + Y_n^2\,(-L_n + Y_n\,L_n) \quad \to D(y) && \text{(36.6-1c)}\\ Y_{n+1} &= Y_n^2 && \text{(36.6-1d)} \end{aligned}$$

Implementation in pari/gp:

```
dd(y, N=7)=
{
    local(R, L, y2, t);
    /* correct up to order 2^(N) */
    L = 1;
    R = 1 + 0*y;
    for(k=1, N,
      /* (L, R) --> (R,  R.(-L).L) */
      y2 = y^2;
      t = R;
      R = R + y2*(-L + y*L);  /* R.(-L).L */
      L = t;
      y = y2;
    );
    return( R )
}
```

We have

$$D(y) = 1 - y^2 + y^3 - y^4 + y^6 - y^8 + y^{10} - y^{11} + y^{12} - y^{14} + y^{15} - y^{16} + y^{18} \pm \ldots \quad \text{(36.6-2)}$$

and (see relation 36.1-4 on page 692)

$$\frac{y}{1-y}\,D(y) = K(y) = y + y^2 + y^4 + y^7 + y^8 + y^{11} + \ldots \quad \text{(36.6-3)}$$

The coefficients are one where the Thue-Morse sequence equals minus one. Thereby the parity number can be computed as $P = \frac{1}{2}\,D\left(\frac{1}{2}\right)$. A functional equation for D is

$$D\,(y) = y\,\frac{1-y}{1+y}\,D\left(y^2\right) + \frac{1}{1+y} \quad \text{(36.6-4)}$$

36.7 Iterations related to the sum of digits

The sequence of the sum of binary digits of the natural numbers starting with zero (entry A035263 of [214]) can be constructed as follows:

```
S0 = 0
S1 = 0 1
S2 = 0 1 1 2
S3 = 0 1 1 2 1 2 2 3
S4 = 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4
S5 = 0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2 3 3 4 2 3 3 4 3 4 4 5
```

Observe that $S_n = S_{n-1}.(S_{n-1} + I_{n-1})$ where I_n is a sequence of n ones and addition is element-wise.

Define $S(y)$ by

$$I_1 = 1, \qquad A_1 = y, \qquad Y_1 = y \tag{36.7-1a}$$

$$I_{n+1} = I_n\,(1+Y_n) = \sum_{k=0}^{2^n-1} y^k \tag{36.7-1b}$$

$$Y_{n+1} = Y_n^2 \tag{36.7-1c}$$

$$A_{n+1} = A_n + Y_{n+1}\,(I_{n+1}+A_n) \quad \to S(y) \tag{36.7-1d}$$

Implementation in pari/gp:

```
s2(y, N=7)=
{
    local(in, A);
    /* correct to order = 2^N-1 */
    in = 1; /* 1+y+y^2+y^3+...+y^(2^k-1) */
    A = y;
    for(k=2, N,
        in *= (1+y);
        y *= y;
        A += y*(in + A);
    );
    return( A );
}
```

The Taylor series is

$$S(y) = 0 + y + y^2 + 2y^3 + y^4 + 2y^5 + 2y^6 + 3y^7 + y^8 + 2y^9 + 2y^{10} + 3y^{11} + 2y^{12} + 3y^{13} + \ldots \tag{36.7-2}$$

Define the *sum-of-digits constant* as $S := S(1/2)/2$, then

$$S = 0.596063172117821679423793925862790645462361238478109932 6\ldots \tag{36.7-3}$$

$$[\text{base } 2] = 0.10011000100101111001100010010110100110001001010111100110001\ldots$$

$$[\text{CF}] = [0, 1, 1, 2, 9, 1, 3, 5, 1, 2, 1, 1, 1, 1, 8, 2, 1, 1, 2, 1, 12, 19, 24, 1, 18, 12, 1, \ldots]$$

We have (see [221])

$$S(y) = \frac{1}{1-y}\sum_{k=0}^{\infty}\frac{y^{2^k}}{1+y^{2^k}} \tag{36.7-4}$$

and also

$$S(y) = \sum_{k=0}^{\infty}\left[\frac{y^{2^k}}{1-y^{2^k}}\prod_{j=0}^{k-1}\left[1+y^{2^j}\right]\right] \tag{36.7-5}$$

The last relation follows from the functional relation for S,

$$S(y) = (1+y)\,S(y^2) + \frac{y}{1-y^2} \tag{36.7-6}$$

It is of the form $F(y) = A(y)\,F(y^2) + B(y)$ where $A(y) = 1+y$ and $B(y) = y/(1-y^2)$ and has the solution

$$F(y) = \sum_{k=0}^{\infty}\left[B(y^{2^k})\prod_{j=0}^{k-1}\left[A(y^{2^j})\right]\right] \tag{36.7-7}$$

This can be seen by applying the functional equation several times:

$$F(y) = A(y)\,F(y^2) + B(y) \tag{36.7-8a}$$

$$= A(y)\,\left[A(y^2)\,F(y^4) + B(y^2)\right] + B(y) \tag{36.7-8b}$$

$$= A(y)\,\left[A(y^2)\,\left[A(y^4)\,F(y^8) + B(y^4)\right] + B(y^2)\right] + B(y) = \ldots \tag{36.7-8c}$$

$$= B(y) + A(y)\,B(y^2) + A(y)\,A(y^2)\,B(y^4) + \ldots \tag{36.7-8d}$$

Weighted sum of digits

Define $W(y)$ by

$$I_1 = \frac{1}{2}, \qquad A_1 = 1, \qquad Y_1 = y \tag{36.7-9a}$$

$$l_{n+1} = I_n \frac{(1+Y_n)}{2} = \frac{1}{2^n} \sum_{k=0}^{2^n-1} y^k \tag{36.7-9b}$$

$$Y_{n+1} = Y_n^2 \tag{36.7-9c}$$

$$A_{n+1} = A_n + Y_{n+1}\,(I_{n+1} + A_n) \quad \to W(y) \tag{36.7-9d}$$

Implementation in pari/gp:

```
w2(y, N=7)=
{
    local(in, y2, A);
    /* correct to order = 2^N-1 */
    in = 1/2; /* 1/2^k * (1+y+y^2+y^3+...+y^(2^k-1)) */
    A = y/2;
    for(k=2, N,
        in *= (1+y)/2;
        y *= y;
        A += y*(in + A);
    );
    return( A );
}
```

In the Taylor series

$$\begin{aligned} W(y) = {} & 0 + \frac{1}{2}y + \frac{1}{4}y^2 + \frac{3}{4}y^3 + \frac{1}{8}y^4 + \frac{5}{8}y^5 + \frac{3}{8}y^6 + \frac{7}{8}y^7 + \\ & + \frac{1}{16}y^8 + \frac{9}{16}y^9 + \frac{5}{16}y^{10} + \frac{13}{16}y^{11} + \frac{3}{16}y^{12} + \frac{11}{16}y^{13} + \frac{7}{16}y^{14} + \frac{15}{16}y^{15} + \\ & + \frac{1}{32}y^{16} + \frac{17}{32}y^{17} + \frac{9}{32}y^{18} + \frac{25}{32}y^{19} + \frac{5}{32}y^{20} + \frac{21}{32}y^{21} + \frac{13}{32}y^{22} + \frac{29}{32}y^{23} + \\ & + \frac{3}{32}y^{24} + \frac{19}{32}y^{25} + \frac{11}{32}y^{26} + \frac{27}{32}y^{27} + \frac{7}{32}y^{28} + \frac{23}{32}y^{29} + \frac{15}{32}y^{30} + \frac{31}{32}y^{31} + \frac{1}{64}y^{32} + \ldots \end{aligned} \tag{36.7-10}$$

the coefficient of the y^n is the weighted sum of digits $w(n) = \sum_{i=0}^{\infty} 2^{-(i+1)}\, b_i$ where $b_0, b_1, \ldots$ is the base-2 representation of n. Note that the numerator in the n-th coefficient is the reversed binary expansion of n.

The corresponding *weighted sum-of-digits constant* or *revbin constant* is $W := W(1/2)$. Then

$$\begin{aligned} W &= 0.448526550676272378923687721254526097616278813538448133 6\ldots \\ [\text{base } 2] &= 0.011100101101001010100010111010010100010101101001010100 01\ldots \\ [\text{CF}] &= [0, 2, 4, 2, 1, 4, 18, 1, 2, 6, 5, 17, 2, 14, 1, 1, 1, 2, 1, 1, 2, 1, 3, 1, 29, 4, 1, \ldots] \end{aligned} \tag{36.7-11}$$

For the function W we have the following functional relation:

$$W\left(\frac{1}{q}\right) = \frac{1}{2}\left[\frac{1+q}{q}\, W\left(\frac{1}{q^2}\right) + \frac{q}{q^2-1}\right] \tag{36.7-12}$$

36.8 Iterations related to the binary Gray code

36.8.1 Series where coefficients are the Gray code of exponents

We construct a function with Taylor series coefficients that are the binary Gray code of the exponent of y. A list of the Gray codes is given below (see section 1.15 on page 36):

```
k ==
0 1 2 3 4 5 6 7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0 1 3 2 6 7 5 4 12 13 15 14 10 11  9  8 24 25 27 26 30 31 29 28 20 21 23 22 18 19 17 16
== graycode(k)
```

The sequence of Gray codes is entry A003188 of [214]. Define the function $G(y)$ as the limit of the iteration

$$\begin{aligned}
F_1 &= y, \qquad B_1 = 1, \qquad Y_1 = y, \qquad I_1 = 1+y && \text{(36.8-1a)}\\
Y_n &= Y_{n-1}^2 && \text{(36.8-1b)}\\
F_n &= (F_{n-1} \qquad\quad) + Y_n\,(B_{n-1} + 2\,I_{n-1}) \quad \to G(y) && \text{(36.8-1c)}\\
B_n &= (F_{n-1} + 2\,I_{n-1}) + Y_n\,(B_{n-1} \qquad\quad) && \text{(36.8-1d)}\\
I_n &= I_{n-1}\,(1+Y_n) && \text{(36.8-1e)}
\end{aligned}$$

Implementation in pari/gp:

```
gg(y, N=15)=
{
    local(t, ii, F, B, Fp, Bp);
    /* correct up to order 2^N-1 */
    F=0+y;  B=1+0;  ii=1+y;
    for(k=2,N,
      y *= y;
      ii *= 2;  /* delete for sum of digits */
      Fp = (F     ) +  y * (B + ii);
      Bp = (F + ii) +  y * (B     );
      F = Fp;  B = Bp;
      ii *= (1+y);
    );
    return( F )
}
```

In the algorithm `F` contains the approximation so far and `B` contains the reversed polynomial:

```
 ---- k = 1 :
F = (y)
B = (1)
 ---- k = 2 :
F = (2y^3+3y^2+y)
B = (y^2+3y+2)
 ---- k = 3 :
F = (4y^7+5y^6+7y^5+6y^4+2y^3+3y^2+y)
B = (y^6+3y^5+2y^4+6y^3+7y^2+5y+4)
 ---- k = 4 :
F = (8y^15+9y^14+11y^13+10y^12+14y^11+15y^10+13y^9+12y^8+4y^7+5y^6+7y^5+6y^4+2y^3+3y^2+y)
B = (y^14+3y^13+2y^12+6y^11+7y^10+5y^9+4y^8+12y^7+13y^6+15y^5+14y^4+10y^3+11y^2+9y+8)
```

We obtain the series

$$\begin{aligned}
G(y) = {} & 0 + 1y + 3y^2 + 2y^3 + 6y^4 + 7y^5 + 5y^6 + 4y^7 + \\
& +12y^8 + 13y^9 + 15y^{10} + 14y^{11} + 10y^{12} + 11y^{13} + 9y^{14} + 8y^{15} + \\
& +24y^{16} + 25y^{17} + 27y^{18} + 26y^{19} + 30y^{20} + 31y^{21} + 29y^{22} + 28y^{23} + \\
& +20y^{24} + 21y^{25} + 23y^{26} + 22y^{27} + 18y^{28} + 19y^{29} + 17y^{30} + 16y^{31} + \ldots
\end{aligned} \qquad \text{(36.8-2)}$$

We define the *Gray code constant* as $G := G(1/2)$:

$$\begin{aligned}
G &= 2.302218287787689301229333006391310761000431077704369505\ldots && \text{(36.8-3)}\\
[\text{base } 2] &= 10.010011010101111000101101011111100100110100011110001 01\ldots\\
[\text{CF}] &= [2, 3, 3, 4, 4, 1, 4, 4, 1, 2, 1, 1, 1, 2, 24, 205, 1, 4, 2, 2, 1, 1, 4, 10, 8, 1, 9, 1, \ldots]
\end{aligned}$$

For the function G we have

$$G\left(\frac{1}{q^2}\right) = \frac{1}{4}\left[\frac{q}{q+1}\,G\left(\frac{1}{q}\right) + \frac{q}{q-1}\,G\left(-\frac{1}{q}\right)\right] \tag{36.8-4a}$$

$$G\left(\frac{1}{q}\right) = \frac{2\,(q+1)}{q}\,G\left(\frac{1}{q^2}\right) + \frac{q^2}{(q-1)\,(q^2+1)} \tag{36.8-4b}$$

$$G(y) = 2\,(1+y)\,G(y^2) + \frac{y}{(1-y)\,(1+y^2)} \tag{36.8-4c}$$

$$G\left(-\frac{1}{q}\right) = \frac{2\,(q-1)}{q}\,G\left(\frac{1}{q^2}\right) - \frac{q^2}{(q+1)\,(q^2+1)} \tag{36.8-4d}$$

$$G\left(\frac{1}{q}\right) = \frac{2\,(q^2+1)}{q\,(q-1)}\,G\left(-\frac{1}{q^2}\right) + \frac{q^2\,(q^4+4\,q^3+4\,q+1)}{(q^4+1)\,(q^2+1)\,(q-1)} \tag{36.8-4e}$$

$$G\left(\frac{1}{q}\right) = \frac{1}{2\,q^2}\left[\frac{(q+1)\,(q^4+4\,q^3+4\,q+1)}{(q^2+1)}\,G\left(\frac{1}{q^2}\right) - \frac{(q^4+1)}{(q-1)}\,G\left(-\frac{1}{q^2}\right)\right] \tag{36.8-4f}$$

The function $G(y)$ can be expressed as

$$G(y) = \frac{1}{1-y}\sum_{k=0}^{\infty}\frac{2^k\,y^{2^k}}{1+y^{2^{k+1}}} \tag{36.8-5}$$

36.8.2 Differences of the Gray code

If one defines $F(y) = (1-y)\,G(y)$ to obtain the successive differences of the Gray code itself, the Taylor coefficients are powers of two in magnitude:

$$F(y) = 0 + y + 2y^2 - y^3 + 4y^4 + y^5 - 2y^6 - y^7 + 8y^8 + y^9 + 2y^{10} - y^{11} - 4y^{12} + y^{13} \pm \ldots \tag{36.8-6}$$

We have

$$F\,(y) = 2\,F\left(y^2\right) + \frac{y}{1+y^2} \tag{36.8-7}$$

Now, as $y/(1+y) = q/(1+q)$ for $q = 1/y$,

$$F\,(y) = F\left(\frac{1}{y}\right) = \sum_{k=0}^{\infty} 2^k\,\frac{y^{2^k}}{1+y^{2^{k+1}}} \tag{36.8-8}$$

Thereby $F(y)$ can be computed everywhere except on the unit circle. The sum

$$\sum_{k=0}^{\infty} 2^k\,\frac{y^{2^k}}{1-y^{2^{k+1}}} \tag{36.8-9}$$

leads to a series with coefficients

```
0 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 32 1 ...
```

corresponding to the (exponential) version of the ruler function which is defined as the highest power of two that divides n. The ruler function (see section 36.4 on page 698)

```
. 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 0 1 ...
```

is the base-2 logarithm of that series.

36.8.3 Sum of Gray code digits

The sequence of the sum of digits of the Gray code of $k \geq 0$ is (entry A005811 in [214]):

```
0 1 2 1 2 3 2 1 2 3 4 3 2 3 2 1 2 3 4 3 4 5 4 3 2 3 4 3 2 3 2 1 2 3 ...
```

Omit the factor 2 in relations 36.8-1c and 36.8-1d on page 707. That is, in the implementation simply remove the line

```
ii *= 2;  /* delete for sum of digits */
```

Let $R(y)$ be the corresponding function, and define the *sum of Gray code digits constant* as $R := R(1/2)/2$, then

$$\begin{aligned} R &= 0.701472376403734520735595521064133208822798986165421295 4\ldots \quad (36.8\text{-}10)\\ [\text{base } 2] &= 0.101100111001001110110001100100111011001110010001101100 0\ldots \\ [\text{CF}] &= [0, 1, 2, 2, 1, 6, 10, 1, 9, 53, 1, 1, 3, 10, 1, 2, 1, 3, 2, 14, 2, 1, 2, 1, 3, 4, 2, \\ &\quad 1, 34, 1, 1, 3, 1, 1, 109, 1, 1, 4, 2, 9, 1, 642, 51, 4, 3, 2, 2, 2, 2, 1, 2, 3, \ldots] \end{aligned}$$

One finds:

$$R\left(\frac{1}{q^2}\right) = \frac{1}{2}\left[\frac{q}{q+1}\,R\left(\frac{1}{q}\right) + \frac{q}{q-1}\,R\left(-\frac{1}{q}\right)\right] \quad (36.8\text{-}11a)$$

$$R\left(\frac{1}{q}\right) = \frac{q+1}{q}\,R\left(\frac{1}{q^2}\right) + \frac{q^2}{q^3-q^2+q-1} \quad (36.8\text{-}11b)$$

$$R\left(-\frac{1}{q}\right) = \frac{q-1}{q}\,R\left(\frac{1}{q^2}\right) - \frac{q^2}{q^3+q^2+q+1} \quad (36.8\text{-}11c)$$

The function $R(y)$ can be expressed as

$$R(y) = \frac{1}{1-y}\sum_{k=0}^{\infty}\frac{y^{2^k}}{1+y^{2^{k+1}}} \quad (36.8\text{-}12)$$

Define the constant P as $P = (R+1)/2$

$$\begin{aligned} P &= 0.850736188201867260367797760532066604411399493082710647 7\ldots \quad (36.8\text{-}13)\\ [\text{base } 2] &= 0.110110011100100111011000110010011101100111001000110110 0\ldots \\ [\text{CF}] &= [0, 1, 5, 1, 2, 3, 21, 1, 4, 107, 7, 5, 2, 1, 2, 1, 1, 2, 1, 6, 1, 2, 6, 1, 1, 8, 1, \\ &\quad 2, 17, 3, 1, 1, 3, 1, 54, 3, 1, 1, 1, 2, 1, 4, 2, 321, 102, 2, 6, 1, 4, 1, 5, 2, \ldots] \end{aligned}$$

Its binary expansion is the *paper-folding sequence* (or *dragon curve sequence*), entry A014577 of [214].

36.8.4 Differences of the sum of Gray code digits

Now define $E(y) = (1-y)\,R(y)$ to obtain the differences of the sum of Gray code digits. From this definition (and relation 36.8-12) one sees that

$$E(y) = \sum_{k=0}^{\infty}\frac{y^{2^k}}{1+y^{2^{k+1}}} \quad (36.8\text{-}14)$$

All Taylor coefficients (except for the constant term) are either plus or minus one:

$$E(y) = 0 + y + y^2 - y^3 + y^4 + y^5 - y^6 - y^7 + y^8 + y^9 + y^{10} - y^{11} - y^{12} + y^{13} \pm \ldots \quad (36.8\text{-}15)$$

We have

$$E\left(\frac{1}{q}\right) = E\left(\frac{1}{q^2}\right) + \frac{q}{q^2+1} \tag{36.8-16a}$$

$$E(y) = E\left(y^2\right) + \frac{y}{y^2+1} \tag{36.8-16b}$$

(use $\frac{y}{y^2+1} = \frac{q}{q^2+1}$ where $q = \frac{1}{y}$ for the latter relation), thereby

$$E(y) = E\left(\frac{1}{y}\right) \tag{36.8-17}$$

So we can compute $E(y)$ everywhere except on the unit circle. For $y < 1$ compute $E(y)$ by the iteration

$$L_0 = 0, \qquad R_0 = 1, \qquad Y_0 = y \tag{36.8-18a}$$

$$L_{n+1} = L_n + Y_n R_n \quad \to E(y) \tag{36.8-18b}$$

$$R_{n+1} = (L_n + 1) + Y_n (R_n - 2) \tag{36.8-18c}$$

$$Y_{n+1} = Y_n^2 \tag{36.8-18d}$$

Implementation in pari/gp:

```
ge(y, N=7)=
{
    local(L, R, Lp, Rp);
    /* correct up to order 2^N-1 */
    L=0;  R=1;
    for(k=2, N,
      Lp = (L    ) +  y * (R    );
      Rp = (L + 1) +  y * (R - 2);
      L = Lp;  R = Rp;
      y *= y;
    );
    return( L + y*R )
}
```

The symbolic representations of the polynomials `L` and `R` should make the underlying idea clear:

```
L = 0
R = +

L = 0+
R = +-

L = 0++-
R = ++--

L = 0++-++--
R = +++--+--

L = 0++-++--+++--+--
R = +++-++---++--+--

L = 0++-++--+++--+--+++-++---++--+--
R = +++-++--+++--+---++-++---++--+--

L = 0++-++--+++--+--+++-++---++--+--+++-++--+++--+---++-++---++--+--
R = +++-++--+++--+--+++-++---++--+---++-++--+++--+---++-++---++--+--
```

The limit of the sequence `L` is entry A034947 of [214]. It is a signed version of the *paper-folding sequence*. The sequence after the initial zero is identical to the sequence of the Kronecker symbols $\left(\frac{-1}{n}\right)$ for $n = 0, 1, 2, 3, \ldots$, see section 37.8 on page 749. Quick verification:

```
? for(n=1,88,print1(if(-1==kronecker(-1,n),"-","+")))
  ++-++--+++--+--+++-++---++--+--+++-++--+++--+---++-++---++--+--+++-++--+++--+--+++-++---
```

We note that the (divisionless) algorithm is fast in practice, we compute the constant R as $R = E(1/2)$ to more than 600 thousand decimal digits:

```
? N=21 ; \\ number of iterations
? B=2^N-1  \\ precision in bits
  2097151
? D=ceil(B * log(2)/log(10)) + 1  \\ precision in decimal digits
  631307
? default(realprecision,D);
```

```
? r=ge(1.0/2, N)
  0.701472376403734520735595521064 [... + > 600k digits]
? ##
  ***   last result computed in 2,726 ms.
? r-ge(1.0/2, N+2)  \\ checking precison
  -3.744192708419695540526787319381583798 4 E-631306
? p=(1+r)/2
  0.850736188201867260367797760532 [... + > 600k digits]
```

36.8.5 Weighted sum of Gray code digits

Define $H(y)$ by its Taylor series $H(y) = \sum_{k=1}^{\infty} h(k)y^k$ where $h(k)$ is the weighted sum of digits of the Gray code:

```
wgs(k)=
{
    local(g,t,s);  s=0;  g=gray(k);
    for(n=0,33, s+=if(bittest(g,n),1/2^(n+1),0));
    return(s);
}
for(k=1,33,  print1(" ",wgs(k)))
  1/2
  3/4 1/4
  3/8 7/8 5/8 1/8
 3/16 11/16 15/16 7/16 5/16 13/16 9/16 1/16
 3/32 19/32 27/32 11/32 15/32 31/32 23/32 7/32 5/32 21/32 29/32 13/32 9/32 25/32 17/32 1/32
 3/64 35/64 ...
```

An iteration for the computation of $H(y)$ is:

$$\begin{aligned}
F_1 &= y/2, \qquad B_1 = 1/2, \qquad Y_1 = y, \qquad I_1 = (1+y)/2 && \text{(36.8-19a)}\\
Y_n &= Y_{n-1}^2 && \text{(36.8-19b)}\\
K_n &= I_{n-1}/2 && \text{(36.8-19c)}\\
F_n &= (F_{n-1} \qquad\quad) + Y_n\,(B_{n-1} + K_n) \quad \to H(y) && \text{(36.8-19d)}\\
B_n &= (F_{n-1} + K_n) + Y_n\,(B_{n-1} \qquad\quad) && \text{(36.8-19e)}\\
I_n &= K_n\,(1 + Y_n) && \text{(36.8-19f)}
\end{aligned}$$

Implementation in pari/gp:

```
gw(y, N=11)=
{
    local(t, ii, F, B, Fp, Bp);
    /* correct up to order 2^N-1 */
    F=0+y;  B=1+0;  ii=1+y;
    ii /= 2;  F /= 2;  B /= 2;
    for(k=2,N,
      y *= y;
      ii /= 2;
      Fp = (F     ) +  y * (B + ii);
      Bp = (F + ii) +  y * (B     );
      F = Fp;  B = Bp;
      ii *= (1+y);
    );
    return( F )
}
```

We define the *weighted sum of Gray code digits constant* as $H := H(1/2)$, then

$$\begin{aligned}
H &= 0.533700488639284991958880481482124285854919322545611891 1\ldots && \text{(36.8-20)}\\
[\text{base } 2] &= 0.100010001010000010011000011000001001000010100000011110 0\ldots\\
[\text{CF}] &= [0, 1, 1, 6, 1, 11, 4, 5, 6, 1, 13, 1, 3, 1, 18, 5, 77, 1, 2, 2, 3, 1, 2, 1, 1, \ldots]
\end{aligned}$$

We have:

$$H\left(\frac{1}{q^2}\right) = \frac{q}{q+1}\,H\left(\frac{1}{q}\right) + \frac{q}{q-1}\,H\left(-\frac{1}{q}\right) \tag{36.8-21a}$$

$$H\left(\frac{1}{q}\right) = \frac{1}{2}\left[\frac{q+1}{q}\,H\left(\frac{1}{q^2}\right) + \frac{q^2}{q^3-q^2+q-1}\right] \tag{36.8-21b}$$

$$H\left(-\frac{1}{q}\right) = \frac{1}{2}\left[\frac{q-1}{q}\,H\left(\frac{1}{q^2}\right) - \frac{q^2}{q^3+q^2+q+1}\right] \tag{36.8-21c}$$

36.9 A function that encodes the Hilbert curve

We define a function $H(y)$ by the following iteration:

$$H_1 = +i\,y + 1\,y^2 - i\,y^3 \tag{36.9-1a}$$

$$R_1 = +i\,y - 1\,y^2 - i\,y^3 \tag{36.9-1b}$$

$$Y_1 = y \tag{36.9-1c}$$

$$Y_{n+1} = Y_n^4 \tag{36.9-1d}$$

$$H_{n+1} = -i\,R_n + Y_n\,(+i + H_n + Y_n\,(+1 + H_n + Y_n\,(-i + i\,R_n))) \tag{36.9-1e}$$

$$R_{n+1} = +i\,H_n + Y_n\,(+i + R_n + Y_n\,(-1 + R_n + Y_n\,(-i - i\,H_n))) \tag{36.9-1f}$$

$$H_{2\,n} \to H(y) \tag{36.9-1g}$$

As the real and imaginary parts are swapped with each step we agree on iterating an even number of times. The resulting function $H(y)$ is

$$H(y) = 0 + y + i\,y^2 - y^3 + i\,y^4 + i\,y^5 + y^6 - i\,y^7 + y^8 + i\,y^9 + y^{10} - i\,y^{11} \pm \ldots \tag{36.9-2}$$

The coefficients of the series are ± 1 and $\pm i$, except for the constant term which is zero. If the sequence of coefficients is interpreted as follows:

```
 0  :=  goto start  '0'
+1  :=  move right  '>'
-1  :=  move left   '<'
+i  :=  move up     '^'
-i  :=  move down   'v'
```

Then, symbolically:

```
H = 0>^<^^>v>^>vv<v>>^>v>>^<^>^<<v<^^^>v>>^<^>^<<v<^<<v>vv<^<v<^^>^< ...
```

Follow the signs to walk along the Hilbert curve, see figure 1.19-A on page 51. An implementation is

```
hh(y, N=4)=
{
    /* correct to order = 4^N-1 */
    local(H, R, tH, tR);
    H= +I*y + 1*y^2 -I*y^3;  R= +I*y - 1*y^2 -I*y^3;
    for(k=2, N,
        y=y^4;
        tH = -I*R + y*(+I + H + y*(+1 + H + y*(-I + I*R)));
        tR = +I*H + y*(+I + R + y*(-1 + R + y*(-I - I*H)));
        H=tH;  R=tR;
    );
    return( H );
}
```

The value of $H(y)$ for $y < 1$ gives the limiting point in the complex plane when the walk according to the coefficients is done with decreasing step lengths: step number k has step length y^k. The least positive y where the real and imaginary part of the endpoint are equal is $y_1 = 0.5436890126920\ldots$. It turns out that y_1 is the real solution of the polynomial $y^3 + y^2 + y - 1$. On might suspect that $M(y) := \mathfrak{Re}\,H(y) - \mathfrak{Im}\,H(y)$, the difference between the real and the imaginary part of $H(y)$, has the factor $y^3 + y^2 + y - 1$. Indeed, $M(y) = y\,(y^3 + y^2 + y - 1)\,(y^{12} + y^8 + y^4 - 1)\cdots$ and a similar statement

is true for the $P(y) := \mathfrak{Re}\, H(y) + \mathfrak{Im}\, H(y)$. We use this observation for the construction of a simplified and quite elegant algorithm for the computation of $H(y)$.

36.9.1 A simplified algorithm

Define the function $P(y)$ as the result of the iteration

$$\begin{aligned} Y_1 &= y, \qquad P_1 = 1 && \text{(36.9-3a)}\\ P_{n+1} &= P_n \left(+1 + Y_n + Y_n^2 - Y_n^3\right) && \text{(36.9-3b)}\\ Y_{n+1} &= Y_n^4 && \text{(36.9-3c)}\\ P_n - 1 &\to P(y) && \text{(36.9-3d)} \end{aligned}$$

and the function $M(y)$ by

$$\begin{aligned} Y_1 &= y, \qquad M_1 = 1 && \text{(36.9-4a)}\\ M_{n+1} &= M_n \left(-1 + Y_n + Y_n^2 + Y_n^3\right) && \text{(36.9-4b)}\\ Y_{n+1} &= Y_n^4 && \text{(36.9-4c)}\\ y\, M_n &\to M(y) && \text{(36.9-4d)} \end{aligned}$$

Now the function $H(y)$ can be computed as

$$H(y) = \frac{1}{2}\Big[(P(y) + M(y)) + i\,(P(y) - M(y))\Big] \qquad \text{(36.9-5)}$$

The following implementations compute the series up to order $4^N - 1$:

```
fpp(y, N=4)=
{
    local( t, Y );
    t = 1;  Y=y;
    for (k=1, N,  t *= (+1+Y+Y^2-Y^3);  Y=Y^4; );
    return( t-1 );
}

fmm(y, N=4)=
{
    local( t, Y );
    t = 1;  Y=y;
    for (k=1, N,  t*= (-1+Y+Y^2+Y^3);  Y=Y^4; );
    return( t*y-Y );
}

hhpm(y, N=4)=
{
    local( tp, tm );
    tp = fpp(y);
    tm = fmm(y);
    return( ((tp+tm) + I*(tp-tm))/2 );
}
```

With a routine `tdir()` that prints a power series with coefficients $\in \{-1, 0, +1\}$ symbolically we obtain:

```
? N=4;
? tdir(fpp(y));tdir(fmm(y));
  0++-+++-++----++++-+++-+++----++++-+++-++----+---+---+---++++-++
  0+---+++-+++-+++-++++---+---+----++++---+---+----+++---+---+----

? tdir((fpp(y)+fmm(y))/2);tdir((fpp(y)-fmm(y))/2);
  0+0-00+0+0+00-0++0+0++0-0+0--0-000+0++0-0+0--0-0--0+00-0-0-00+0-00
  00+0++0-0+0--0-00+0-00+0+0+00-0+++0-00+0+0+00-0+00-0--0+0-0++0+0++
```

The n-th coefficient of the Taylor series of $P(y)$ equals the parity of the number of threes in the radix-4 representation of n. This fact can be used for an efficient bit level algorithm, see section 1.19 on page 51.

The coefficients of the Taylor series of the function P can be obtained by a string substitution engine:

```
Number of symbols = 2
Start: +
Rules:
  + --> +++-
  - --> ---+
-------------
0:   (#=1)
  +
1:   (#=4)
  +++-
2:   (#=16)
  +++-+++-+++----+
3:   (#=64)
  +++-+++-+++----++++-+++-+++----++++-+++-+++----+---+---+---++++-
4:   (#=256)
  +++-+++-+++----++++-+++-+++----++++-+++-+++----+---+---+---++++-+++-+++-++ ...

  0++-+++-+++----++++-+++-+++----++++-+++-+++----+---+---+---++++-++ == fpp()
```

Similarly for M:

```
Number of symbols = 2
Start: +
Rules:
  + --> -+++
  - --> +---
-------------
0:   (#=1)
  +
1:   (#=4)
  -+++
2:   (#=16)
  +----+++-+++-+++
3:   (#=64)
  -++++---+---+---++++-+++-++++---+++-+++-++++---+++-+++-+++
4:   (#=256)
  +----+++-+++-+++-++++---+---+----++++---+---+----++++---+---+----++++---+- ...

 0+----+++-+++-+++-++++---+---+----++++---+---+----++++---+---+---- == fmm()
```

36.9.2 The turns of the Hilbert curve

We compute a function with series coefficients $\in \{-1, 0, +1\}$ that correspond to the turns of the Hilbert curve. We use $+1$ for a right turn, -1 for a left turn and zero for no turn. The sequences of turns starts as

```
0--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+--+ \
0++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++- \
-++-0--++--0-++0+--+0++--++0+--00--+0++--++0+--+0++-0--++--0-++0 \
+--+0++--++0+--0-++-0--++--0-++00++-0--++--0-++-0--+0++--++0+-- ...
```

The computation is slightly tricky:

```
hht(y, N=4)=
{
    /* correct to order = 4^N-1 */
    local( t, Y, F, s, p );
    t = 1;  Y=y;  p = 1;
    F = y + y^2;
    for(k=2, N,
        Y = Y^4;
        t = -F + Y*F + Y^2*F  - Y^3*F;
        p *= 4;
        if ( 0==(k%2),
            t +=  y^(1*p-1);
            t +=  y^(3*p);
            t -=  (y+1)*y^(2*p-1);
            , /* else */
            t +=  y^(1*p);
            t +=  y^(3*p-1);
        );
        F = t;
    );
    if ( 1==N%2,  F = -F );  \\ same result for even and odd N
    return( F );
}
```

36.10 Sparse variants of the inverse

36.10.1 A fourth order iteration

Define the function $F(y)$ as the result of the iteration

$$\begin{aligned} F_0 &= 1, \qquad Y_0 = y && \text{(36.10-1a)}\\ F_{n+1} &= F_n\,(1+Y_n) \quad \to F(y) && \text{(36.10-1b)}\\ Y_{n+1} &= Y_n^4 && \text{(36.10-1c)} \end{aligned}$$

The terms of the continued fraction of $F(1/10)$ grow doubly exponentially:

```
[1, 9, 1, 90, 110, 909000, 11001100000, 9090000090900000000000,
 1100110000000000110011000000000000000000000000,
 909000009090000000000000000000000090900000909000000000000000000000000000000000000000000000000000000
  ... ]
```

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$\begin{aligned} t_0 &= 1 && \text{(36.10-2a)}\\ t_1 &= q-1 && \text{(36.10-2b)}\\ t_2 &= 1 && \text{(36.10-2c)}\\ t_3 &= q^2-q && \text{(36.10-2d)}\\ t_4 &= q^2+q && \text{(36.10-2e)}\\ t_5 &= q^6-q^5+q^4-q^3 && \text{(36.10-2f)}\\ t_6 &= q^{10}+q^9+q^6+q^5 && \text{(36.10-2g)}\\ t_7 &= q^{22}-q^{21}+q^{20}-q^{19}+q^{14}-q^{13}+q^{12}-q^{11} && \text{(36.10-2h)}\\ t_8 &= q^{42}+q^{41}+q^{38}+q^{37}+q^{26}+q^{25}+q^{22}+q^{21} && \text{(36.10-2i)}\\ t_9 &= q^{86}-q^{85}+q^{84}-q^{83}+q^{78}-q^{77}+q^{76}-q^{75}+ && \text{(36.10-2j)}\\ &\quad +q^{54}-q^{53}+q^{52}-q^{51}+q^{46}-q^{45}+q^{44}-q^{43} \end{aligned}$$

For $j \geq 4$ we have

$$\frac{t_j}{t_{j-2}} = q^{2J}+q^J = q^J\,(q^J+1) \quad \text{where} \quad J = 2^{j-4} \qquad \text{(36.10-3)}$$

A functional relation for F is given by

$$F(y)\,F(y^2) = \frac{1}{1-y} \qquad \text{(36.10-4)}$$

The relation is (mutatis mutandis) also true for the truncated product. The binary expansions of $F(y)$, $F(y^2)$ and their product for $y = 1/2$ are (dots for zeros):

```
11..11.........11..11.....................................................
1.1.....1.1.....................1.1.....1.1.....................................
1111111111111111111111111111111111111111111111111111111111111111111111....
```

Since the expansions are palindromes also their correlation is a sequence of ones. Now set $F_k := F(y^k)$ then, by relations 36.10-4 and 36.1-16e on page 694 (for $k = 2$, 3, and 5),

$$\begin{aligned} 0 &= F_1^2\,F_2 - 2\,F_1\,F_2\,F_4 + F_4 && \text{(36.10-5a)}\\ 0 &= F_1^3\,F_2^3 - F_3\,F_6\,\left(3\,F_1^2\,F_2^2 - 3\,F_1\,F_2 + 1\right) && \text{(36.10-5b)}\\ 0 &= F_1^5\,F_2^5 - F_{10}\,F_5\,\left(5\,F_1^4\,F_2^4 - 10\,F_1^3\,F_2^3 + 10\,F_1^2\,F_2^2 - 5\,F_1\,F_2 + 1\right) && \text{(36.10-5c)} \end{aligned}$$

We note that for power series over $GF(2)$ relation 36.10-4 becomes $F(y)^3 = 1/(1-y)$. That is, $F(y) = \sqrt[3]{1-y}$. In general, an iteration for the inverse (2^w-1)-st root is obtained by replacing relation 36.10-1c with $Y_{k+1} = Y_k^{2^w}$.

36.10.2 A different fourth order iteration

To define the function $F(y)$ we modify the third order iteration for $\frac{1}{1+y}$

```
inv3m(y, N=6)=  /* third order --> 1/(1+y) */
{ /* correct to order 3^N */
    local(T);
    T = 1;
    for(k=1, N,
      T *= ( 1 - y + y^2 );
      y = y^3;
    );
    return( T );
}
```

to obtain a fourth-order iteration:

```
f43(y, N=6)=
{ /* correct to order 4^N */
    local(T, yt);
    T = 1;
    for(k=1, N,
      T *= ( 1 - y + y^2 );
      y = y^4;  /* ! */
    );
    return( T );
}
```

That is,

$$\begin{aligned} F_0 &= 1, \qquad Y_0 = y && \text{(36.10-6a)}\\ F_{n+1} &= F_n\,(1 - Y_n + Y_n^2) \quad \to F(y) && \text{(36.10-6b)}\\ Y_{n+1} &= Y_n^4 && \text{(36.10-6c)} \end{aligned}$$

The first few terms of the power series are

$$F(y) = 1 - y + y^2 - y^4 + y^5 - y^6 + y^8 - y^9 + y^{10} - y^{16} + y^{17} - y^{18} + y^{20} - y^{21} \pm \ldots \quad \text{(36.10-7)}$$

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$\begin{aligned} t_0 &= 0 && \text{(36.10-8a)}\\ t_1 &= 1 && \text{(36.10-8b)}\\ t_2 &= q && \text{(36.10-8c)}\\ t_3 &= q && \text{(36.10-8d)}\\ t_4 &= q^2 - q && \text{(36.10-8e)}\\ t_5 &= q^4 + q^3 - q - 1 && \text{(36.10-8f)}\\ t_6 &= q^8 - q^7 + q^6 - q^4 + q^3 - q^2 && \text{(36.10-8g)}\\ t_7 &= q^{16} + q^{15} - q^{13} + q^{11} + q^{10} - q^8 - q^7 + q^5 - q^3 - q^2 && \text{(36.10-8h)} \end{aligned}$$

For $j \geq 6$ we have

$$\frac{t_j}{t_{j-2}} = q^{6J} + q^{4J} + q^{3J} + q^J = q^J\,(q^J + 1)\,(q^{2J} - q^J + 1)\,(q^{2J} + 1) \quad \text{(36.10-9)}$$

where $J = 2^{j-6}$. The terms of the continued fraction of $F(1/q)$ for integer q grow doubly exponentially:

```
? contfrac(f43(0.5))
  [0, 1, 2, 2, 2, 21, 180, 92820, 3032435520, 26126907554432455680,
   240254294248527099500117907463345274880,
   164001256750215347067944129734442019102853751066678216639025390799096507269120,
   ... ]
/* number of decimal digits of the terms in the CF: */
  [-, 1, 1, 1, 1, 2, 3, 5, 10, 20, 39, 78, 154, 309, ... ]
```

By construction,

$$F(y) = \left(1 - y + y^2\right)\, F\left(y^4\right) \tag{36.10-10a}$$
$$F(-y) = \left(1 + y + y^2\right)\, F\left(y^4\right) \tag{36.10-10b}$$

The equivalent forms with $y = 1/q$ are

$$F\left(\frac{1}{q}\right) = \frac{q^2 - q + 1}{q^2}\, F\left(\frac{1}{q^4}\right) \tag{36.10-11a}$$
$$F\left(-\frac{1}{q}\right) = \frac{q^2 + q + 1}{q^2}\, F\left(\frac{1}{q^4}\right) \tag{36.10-11b}$$

Now $q^2 - q + 1 = p^2 - p + 1$ if $p = 1 - q$, so

$$F\left(1 - \frac{1}{q}\right) = \frac{q^2 - q + 1}{q^2}\, F\left(\left(1 - \frac{1}{q}\right)^4\right) \quad \text{where} \quad q > 1 \tag{36.10-12a}$$
$$F\left(1 + \frac{1}{q}\right) = \frac{q^2 + q + 1}{q^2}\, F\left(\left(1 + \frac{1}{q}\right)^4\right) \quad \text{where} \quad q < -1 \tag{36.10-12b}$$

Adding relations $\alpha\times$ (36.10-11a) and $\beta\times$ (36.10-12a) and simplifying gives

$$\frac{\alpha\, F(y) + \beta\, F(1-y)}{\alpha\, F(y^4) + \beta\, F((1-y)^4)} = y^2 - y + 1 \quad \text{where} \quad \alpha, \beta \in \mathbb{C} \tag{36.10-13}$$

36.10.3 A sixth order iteration

Define the function $F(y)$ by the iteration

$$F_0 = 1, \qquad Y_0 = y \tag{36.10-14a}$$
$$F_{n+1} = F_n\left(1 + Y_n + Y_n^2\right) \quad \to F(y) \tag{36.10-14b}$$
$$Y_{n+1} = Y_n^6 \tag{36.10-14c}$$

Let $[t_0, t_1, t_2, \ldots]$ be the continued fraction of $F(1/q)$, then

$$t_0 = 1 \tag{36.10-15a}$$
$$t_1 = q - 1 \tag{36.10-15b}$$
$$t_2 = q + 1 \tag{36.10-15c}$$
$$t_3 = q^2 - q \tag{36.10-15d}$$
$$t_4 = q^{10} + q^9 + q^8 + q^4 + q^3 + q^2 \tag{36.10-15e}$$
$$t_5 = q^8 - q^7 + q^5 - q^4 \tag{36.10-15f}$$
$$t_6 = q^{64} + q^{63} + q^{62} + q^{58} + q^{57} + q^{56} + q^{52} + q^{51} + q^{50} + \\ + q^{28} + q^{27} + q^{26} + q^{22} + q^{21} + q^{20} + q^{16} + q^{15} + q^{14} \tag{36.10-15g}$$
$$t_7 = q^{44} - q^{43} + q^{41} - q^{40} + q^{26} - q^{25} + q^{23} - q^{22} \tag{36.10-15h}$$
$$t_8 = q^{388} + q^{387} + \ldots + q^{87} + q^{86} \tag{36.10-15i}$$

For $j \geq 4$ we have

$$\frac{t_j}{t_{j-2}} = \begin{cases} q^J + q^{J/2} & \text{if} \quad j \quad \text{odd} \\ \left(q^{10J} + q^{9J} + q^{8J} + q^{4J} + q^{3J} + q^{2J}\right) / \left(q^J + 1\right) & \text{else} \end{cases} \tag{36.10-16}$$

where $J = 6^{j-4}$.

36.11 An iteration related to the Fibonacci numbers

The so-called *rabbit constant* is

$$\begin{aligned} A &= 0.709803442861291314641787399444575597012502205767860516 9\ldots \quad (36.11\text{-}1) \\ [\text{base } 2] &= 0.1011010110110101101011011010110110101101011011010110101\ldots \\ [\text{CF}] &= [0, 1, 2, 2, 4, 8, 32, 256, 8192, 2097152, 17179869184, \\ &\quad 36028797018963968, 618970019642690137449562112, \ldots] \\ &= [0, 2^0, 2^1, 2^1, 2^2, 2^3, 2^5, 2^8, 2^{13}, \ldots, 2^{F_n}, \ldots] \end{aligned}$$

The sequence of zeros and ones after the decimal point is referred to as *rabbit sequence* or *infinite Fibonacci word*, entry A005614 of [214]. The rabbit sequence can be obtained by starting with a single zero and repeated application of the following substitution rules: simultaneously replace all zeros by one ($0 \to 1$, 'young rabbit gets old') and all ones by one-zero ($1 \to 10$, 'old rabbit gets[1] child'). The evolution is:

```
A0 = 0
A1 = 1
A2 = 10
A3 = 101
A4 = 10110
A5 = 10110101
A6 = 1011010110110
A7 = 101101011011010110101
A -->1011010110110101101011011010110110101101...
```

The crucial observation is that each element A_n can be obtained by appending A_{n-2} to A_{n-1}, that is $A_n = A_{n-1}.A_{n-2}$. To compute the value of the rabbit constant in base 2 to N bits precision the whole process requires only copying N bits of data is the minimal conceivable work for a (non-sparse) computation.

We define a function $A(y)$ that has the special value

$$A = \frac{1}{2} A\left(\frac{1}{2}\right) \quad (36.11\text{-}2)$$

by the equivalent operation for power series. We obtain the iteration

$$\begin{aligned} L_0 &= 0 \quad R_0 = 1 \quad l_0 = 1 \quad r_0 = y && (36.11\text{-}3a) \\ l_{n+1} &= r_n = y^{F_n} && (36.11\text{-}3b) \\ r_{n+1} &= r_n\, l_n = y^{F_{n+1}} && (36.11\text{-}3c) \\ L_{n+1} &= R_n && (36.11\text{-}3d) \\ R_{n+1} &= R_n + r_{n+1}\, L_n = R_n + y^{F_{n+1}}\, L_n \quad \to A(y) && (36.11\text{-}3e) \end{aligned}$$

Here F_n denotes the n-th Fibonacci number (sequence A000045 of [214]):

```
n   0  1  2  3  4  5  6   7   8   9  10  11   12   13   14   15 ...
Fn  0  1  1  2  3  5  8  13  21  34  55  89  144  233  377  610 ...
```

After the n-th step the series in y is correct up to order $F_{n+2} - 1$. That is, the order of convergence equals $\frac{\sqrt{5}+1}{2} \approx 1.6180$. The function $A(y)$ has the power series

$$A(y) = 1 + y^2 + y^3 + y^5 + y^7 + y^8 + y^{10} + y^{11} + y^{13} + y^{15} + y^{16} + y^{18} + y^{20} + y^{21} \ldots \quad (36.11\text{-}4)$$

The sequence of exponents of y in the series is entry A022342 of [214], the Fibonacci-even numbers. A pari/gp implementation is

```
fa(y, N=10)=
{
    local(t, yl, yr, L, R, Lp, Rp);
    /* correct up to order fib(N+2)-1 */
    L=0;  R=1;   yl=1;  yr=y;
```

[1] No sex, no death.

```
    for(k=1, N,
      t=yr;  yr*=yl;  yl=t;
      Lp=R;  Rp=R+yr*L;  L=Lp;  R=Rp;
    );
    return( R )
}
```

36.11.1 Fibonacci representation

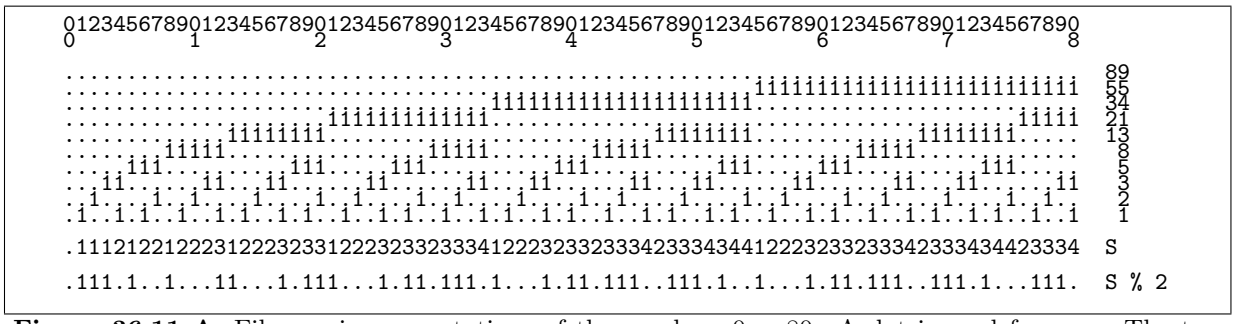


Figure 36.11-A: Fibonacci representations of the numbers $0 \ldots 80$. A dot is used for zero. The two lower lines are the sum of digits and the sum of digits modulo two, the *Fibonacci parity*.

The greedy algorithm to obtain the *Fibonacci representation* (or *Zeckendorf representation*) of an integer repeatedly subtracts the largest Fibonacci number that is greater or equal to it until the number is zero. The Fibonacci representations of the numbers $0 \ldots 80$ are shown in figure 36.11-A.

The sequence of lowest Fibonacci bits is

```
0100101001001010010100100101001001010010100100101001010010100100101001...
```

Interpreted as the binary number $x = 0.1001010010010_2 \ldots = 0.5803931 \ldots$ it turns out that $A = 1 - x/2$ (that is, $x = 2 - A(1/2)$). Alternatively, one can compute the number as $x = A[1, 0, 1, 1/2]$.

The sequence of numbers of digits in the Fibonacci representations (second lowest row in figure 36.11-A) is entry A007895 of [214]. This sequence modulo two gives the *Fibonacci parity*, it can be computed by initializing $L_0 = 1$ and changing relation 36.11-3e on the facing page to

$$R_{n+1} \quad = \quad R_n - r_{n+1}\, L_n \;=\; R_n - y^{F_{n+1}}\, L_n \quad \to A_p(y) \tag{36.11-5}$$

Let the corresponding function be $A_p(y)$. We define the *Fibonacci parity constant* A_p as

$$\begin{aligned} A_p &= 1 - A_p(1/2)/2 && \text{(36.11-6a)} \\ &= 0.910533470863561763804686886771098007344581229006937645 4\ldots && \text{(36.11-6b)} \\ \text{[base 2]} &= 0.111010010001100010111000101101110100010110111001110100 10\ldots \\ \text{[CF]} &= [0, 1, 10, 5, 1, 1, 1, 3, 4, 2, 6, 25, 4, 5, 1, 1, 3, 5, 1, 3, 2, 1, 1, 1, 3, 1, 3, 22, 1, \\ &\quad 10, 1, 2, 3, 2, 73, 1, 111, 46, 1, 51, 2, 1, 1, 5, 1, 65, 3, 1, 3, 2, 5, 6, 1, 4, 1, 2, \ldots] \end{aligned}$$

The sequence of bits in the binary expansion of A_p is entry A095076 of [214].

The sequence of the Fibonacci representations interpreted as binary numbers is

```
0, 1, 2, 4, 5, 8, 9, 10, 16, 17, 18, 20, 21, 32, 33, 34, 36, 37, 40,
41, 42, 64, 65, 66, 68, 69, 72, 73, 74, 80, 81, 82, 84, 85, 128, 129, ...
```

This is entry A003714 of [214], where the numbers are called *Fibbinary numbers*. Define $F_2(y)$ to be the function that has the same sequence of power series coefficients:

$$F(y) \quad = \quad 0 + 1\,y + 2\,y^2 + 4\,y^3 + 5\,y^4 + 8\,y^5 + 9\,y^6 + 10\,y^7 + 16\,y^8 + 17\,y^9 + 18\,y^{10} + \ldots \tag{36.11-7}$$

A slightly more general function $F_b(y)$ (which for $b = 2$ gives the power series above) can be computed by the iteration

$$\begin{aligned} L_0 &= 0, \quad R_0 = y, \quad l_0 = y, \quad r_0 = y && \text{(36.11-8a)}\\ A_0 &= 1, \quad B_0 = 1, \quad b = 2 && \text{(36.11-8b)}\\ A_{n+1} &= b\,B_n && \text{(36.11-8c)}\\ B_{n+1} &= b\,[B_n + r_n\,A_n] && \text{(36.11-8d)}\\ l_{n+1} &= r_n = y^{F_{n+1}} && \text{(36.11-8e)}\\ r_{n+1} &= r_n\,l_n = y^{F_{n+2}} && \text{(36.11-8f)}\\ L_{n+1} &= R_n && \text{(36.11-8g)}\\ R_{n+1} &= R_n + r_{n+1}\,[L_n + A_{n+1}] \quad \to F_b(y) && \text{(36.11-8h)} \end{aligned}$$

A pari/gp implementation is

```
ffb(y, b=2, N=13)=
{ /* correct up to order fib(N+3)-1 */
    local(t, yl, yr, L, R, Lp, Rp, Ri, Li);
    L=0;  R=0+1*y;
    Li=1;  Ri=1;
    yl=y;  yr=y;
    for (k=1, N,
        Li*=b;  Ri*=b;
        Lp=Ri;  Rp=Ri+yr*Li;  Li=Lp;  Ri=Rp;
        t=yr;  yr*=yl;  yl=t;
        Lp=R;  Rp=R+yr*(L+Li);  L=Lp;  R=Rp;
    );
    return( R )
}
```

The sequence of coefficients

```
  1, 6, 14, 35, 90, 234, 611, 1598, 4182, 10947, 28658, 75026, 196419, 514230, ...
```

coincides (disregarding the initial one) with entries A032908 and A093467 of [214]. Let $B(x)$ be the function with power series coefficients equal to one if the exponent is a Fibbinary number and zero else:

$$B(x) := 1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^{10} + x^{16} + x^{17} + x^{18} + x^{20} + \ldots \qquad \text{(36.11-9)}$$

Then a functional equation for $B(x)$ is (see entry A003714 of [214])

$$B(x) = x\,B(x^4) + B(x^2) \qquad \text{(36.11-10)}$$

We turn the relation into an recursion for the computation of $B(x)$ correct up to the term x^N:

```
fibbi(x, N=25, nr=1)=
{
    if ( nr>N, return( 1 ) );
    return( x*fibbi(x^4,N,4*nr)+fibbi(x^2,N,2*nr) );
}
```

We check the functional relation:

```
? N=30;  R=O(x^(N+1));  \\ R is used to truncate terms of order >N
? t=fibbi(x,N)+R
  1 + x + x^2 + x^4 + x^5 + x^8 + x^9 + x^10 + x^16 + x^17 + x^18 + x^20 + x^21 + O(x^31)
? t2=fibbi(x^2,N)+R
  1 + x^2 + x^4 + x^8 + x^10 + x^16 + x^18 + x^20 + O(x^31)
? t4=x*fibbi(x^4,N)+R
  x + x^5 + x^9 + x^17 + x^21 + O(x^31)
? t-(t4+t2)
  O(x^31)
```

36.11.2 Digit extract algorithms for the rabbit constant

The *spectrum* of a real number x is the sequence of integers $\lfloor k \cdot x \rfloor$ where $k = 1, 2, 3, \ldots$. As mentioned in [240], the spectrum of the golden ratio $g = \frac{\sqrt{5}+1}{2} = 1.61803\ldots$ gives the exponents of y where the series for $y\,A(y)$ has coefficient one:

```
bt(x, n=25)=
{
    local(v);
    v = vector(n);
    for (k=1, n, v[k]=floor(x*k));
    return ( v );
}
```

```
g=(sqrt(5)+1)/2
 1.6180339887498948482045868343656381177203091798057628 62
```

```
n=40;
bt(g, n)
  [1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, 22, 24, 25, 27, 29,
  30, 32, 33, 35, 37, 38, 40, 42, 43, 45, 46, 48, 50, 51, 53, 55, 56,
  58, 59, 61, 63, 64]
```

```
t=taylor(y*fa(y),y)
  y + y^3 + y^4 + y^6 + y^8 + y^9 + y^11 + y^12 + y^14 + y^16 + y^17 +
  y^19 + y^21 + y^22 + y^24 + y^25 + y^27 + y^29 + y^30 + y^32 + y^33 +
  y^35 + y^37 + y^38 + y^40 + y^42 + y^43 + y^45 + y^46 + y^48 + y^50 +
  y^51 + y^53 + y^55 + y^56 + y^58 + y^59 + y^61 + y^63 + y^64 + O(y^66)
```

The sequence [1, 3, 4, 6, ...] of exponents where the coefficient equals one is sequence A000201 of [214]. There is a digit extract algorithm for the binary expansion of the rabbit constant. We use a binary search algorithm:

```
bts(x, k)=
{ /* return 0 if k is not in the spectrum of x, else return index >=1  */
    local(nlo, nhi, t);
    if ( 0==k, return(0) );
    t = 1 + ceil(k/x);  \\ floor(t*x)>=k
    nlo = 1;  nhi = t;
    while ( nlo!=nhi,
        t = floor( (nlo+nhi)/2 );
        if ( floor(t*x) < k,  nlo=t+1,  nhi=t);
    );
    if ( floor(nhi*x) == k,  return(nhi),  return (0));
}
```

```
g=(sqrt(5)+1)/2
for(k=1,65,if(bts(g,k),print1("1"),print1("0")));print();
  10110101101101011010110110101101101011010110110101101011011010110
```

The algorithm is very fast, we compute 1000 bits starting from position 1,000,000,000,000:

```
g=(sqrt(5)+1)/2
dd=10^12;  /* digits starting at position dd... */
for(k=dd,dd+1000,if(bts(g,k),print1("1"),print1("0")));print();
  1101101011011010110101101101011010110101101101011010110110101101011011010110
  [--snip--]
  ***   last result computed in 236 ms.
```

The connection between the sequence of lowest Fibonacci bits and the rabbit constants allows even more. Subtracting the Fibonacci numbers > 1 until zero or one is reached, gives the complement of the rabbit sequence:

```
fpn=999;
vpv=vector(fpn, j, fibonacci(j+2)); /* vpv=[2,3,5,8,...] */
t=vpv[length(vpv)];
log(t)/log(10)
  208.8471...  /* ok for range up to >10^200 (!) */
```

```
flb(x)=
{ /* return the lowest bit of the Fibonacci representation */
    local(k, t);
    k=bsearchgeq(x, vpv);
    while ( k>0,
        t = vpv[k];
        if (x>=t,  x-=t);
        k-- );
    return ( x );
}
```

```
dd=0;
for(k=dd,dd+40,t=flb(k);print1(1-t))
     10110101101101011010110110101101101011010
/* 0.10110101101101011010110110101101101011010  rabbit constant */
```

The routine `bsearchgeq()` does a binary search (see section 3.2 on page 117) for the first element that is greater or equal to the element sought:

```
bsearchgeq(x, v)=
{ /* return index of first element in v[] that is >=x, return 0 if x>max(v[])  */
    local(nlo, nhi, t);
    nlo = 1;  nhi = length(v);
    while ( nlo!=nhi,
        t = floor( (nlo+nhi)/2 );
        if ( v[t] < x,  nlo=t+1,  nhi=t);
    );
    if ( v[nhi] >= x,  return(nhi),  return (0));
}
```

We compute the first 1000 bits starting from position 10^{100}:

```
dd=10^100-1;
for (k=dd, dd+1000, t=flb(k); print1(1-t))
  1101011011010110101101101011011010110110101101011011010
  [--snip--]
  ***   last result computed in 1,305 ms.
```

36.12 Iterations related to the Pell numbers

Replacement rules: simultaneously replace all zeros by one ($0 \to 1$) and all ones by one-one-zero ($1 \to 110$).

```
    B0 =  0
    B1 =  1
    B2 =  110
    B3 =  1101101
    B4 =  11011011101101110
    B5 =  1101101110110111011011011101101110110110101
    B --> 1101101110110111011011011101101110110110111011011101101110111011...
```

Now the construction is $B_n = B_{n-1}.B_{n-1}.B_{n-2}$. The length of the n-th string is

$$p_n \quad = \quad 1,\ 1,\ 3,\ 7,\ 17,\ 41,\ 99,\ 239,\ \ldots \qquad p_k = 2p_{k-1} + p_{k-2}$$

This sequence is entry A001333 of [214], the numerators of the continued fraction of $\sqrt{2}$. The sequence `B` of zeros and ones is entry A080764. The Pell numbers are the first differences (and the denominators of the continued fraction of $\sqrt{2}$), sequence A000129:

$$0,\ 1,\ 2,\ 5,\ 12,\ 29,\ 70,\ 169,\ 408,\ 985,\ 2378,\ 5741,\ \ldots$$

Now define the function $B(y)$ by the iteration

$$\begin{aligned}
L_0 &= 1, \qquad R_0 = 1+y, \qquad l_0 = y, \qquad r_0 = y && \text{(36.12-1a)}\\
l_{n+1} &= r_n && \text{(36.12-1b)}\\
r_{n+1} &= r_n^2\, l_n && \text{(36.12-1c)}\\
L_{n+1} &= R_n && \text{(36.12-1d)}\\
R_{n+1} &= R_n + r_{n+1}\, R_n + r_{n+1}^2\, L_n \quad \to B(y) && \text{(36.12-1e)}
\end{aligned}$$

After the n-th step the series in y is correct up to order p_n. That is, the order of convergence is $\sqrt{2}+1 \approx 2.4142$. Implementation in pari/gp:

```
fb(y, N=8)=
{
    local(t, yr, yl, L, R, Lp, Rp);
    L=1;  R=1+y;  yl=y;  yr=y;
    for(k=1,N,
      t=yr; yr*=yr*yl; yl=t;
      Lp=R; Rp=R+yr*R+yr^2*L; L=Lp; R=Rp;
    );
    return( R )
}
```

We obtain the series

$$B(y) = 1 + y + y^3 + y^4 + y^6 + y^7 + y^8 + y^{10} + y^{11} + y^{13} + y^{14} + y^{15} + y^{17} + y^{18} + y^{20} + \ldots \quad (36.12\text{-}2)$$

Define the *Pell constant* B as $B = \frac{1}{2}\,B(\frac{1}{2})$, then

$$\begin{aligned} B &= 0.85826765646100205579226030843337514866490519008350677786\ldots && (36.12\text{-}3) \\ [\text{base 2}] &= 0.11011011101101110110110111011011101101101110110111011011\ldots \\ [\text{CF}] &= [0, 1, 6, 18, 1032, 16777344, 288230376151842816, \\ &\quad 1393796574908163946345982392042721617379328, \ldots] \end{aligned}$$

For the terms of the continued fraction we note

```
                                      6 ==   2^2 +  2^1
                                     18 ==   2^4 +  2^1
                                   1032 ==  2^10 +  2^3
                               16777344 ==  2^24 +  2^7
                     288230376151842816 ==  2^58 + 2^17
1393796574908163946345982392042721617379328 == 2^140 + 2^41
```

36.12.1 Pell palindromes

Define the function $P(y)$ by

$$\begin{aligned} L_0 &= 1, \qquad R_0 = 1 + y^2, \qquad l_0 = y, \qquad r_0 = y && (36.12\text{-}4a) \\ l_{n+1} &= r_n && (36.12\text{-}4b) \\ r_{n+1} &= r_n^2\, l_n && (36.12\text{-}4c) \\ L_{n+1} &= R_n && (36.12\text{-}4d) \\ R_{n+1} &= R_n + r_{n+1}\, L_n + r_{n+1}\, l_{n+1}\, R_n \quad \to P(y) && (36.12\text{-}4e) \end{aligned}$$

Note that R_0 is a palindrome and in relation 36.12-4e the combination of the parts gives a palindrome. For $R_0 = 1 + y + y^2$ the iteration computes $\frac{1}{1-y}$.

Define the *Pell palindromic constant* as $P = P(1/2)/2$, then

$$\begin{aligned} P &= 0.73216043306353283716459018717730446572729865896041123390\ldots && (36.12\text{-}5) \\ [\text{base 2}] &= 0.10111011011011101101110110111011011011101101110110111011011011\ldots \\ [\text{CF}] &= [0, 1, 2, 1, 2, 1, 3, 17, 1, 7, 2063, 1, 63, 268437503, 1, 8191, 590295810358974087167, \\ &\quad 1, 1073741823, 374144419156711147060143317175958748842277436653567, 1, \ldots] \end{aligned}$$

By construction, the binary expansion is a palindrome up to lengths $1, 3, 7, 17, 41, 99, 239, \ldots$.

The sequence of zeros and ones in the binary expansion of P is entry A104521 of [214]. It can be obtained by the replacement rules ($0 \to 1$) and ($1 \to 101$):

```
P0 =  0
P1 =  1
P2 =  101
P3 =  1011101
P4 =  10111011011011101
P5 =  10111011011011101101110110111011011011101
P --> 1011101101101110110111011011101101101110110111011011011101101...
```

36.12.2 Pell representation

To obtain the Pell representation of a given number n, set $t = n$ repeatedly subtract from t the largest number $p_k \in \{1, 3, 7, 17, 41, 99, 239, \ldots\}$ that is not greater than t. Stop when $t = 0$. The number of times that p_k has been subtracted gives the k-th digit of the representation. The resulting digits are 0, 1, or 2. If the k-th digit equals 2 then the $(k-1)$-th digit will be zero.

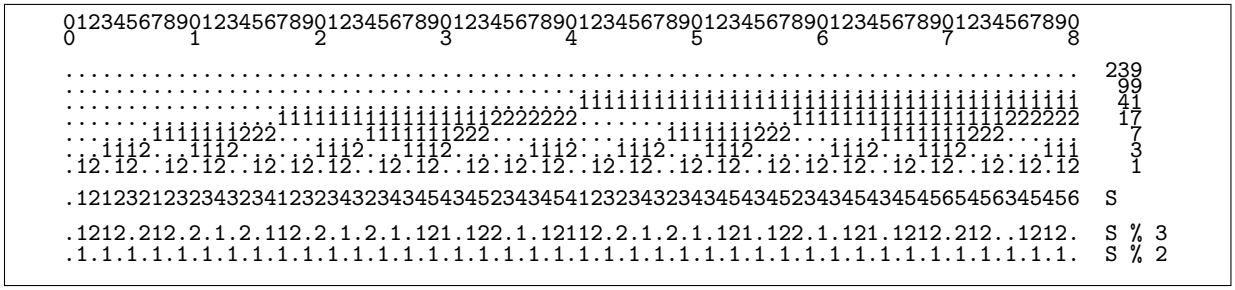


Figure 36.12-A: Pell representations of the numbers $0 \ldots 80$ (dots for zeros). The three lower lines are the sum of digits and the sum of digits modulo three and two.

The function $S(y)$ that has Taylor series coefficients corresponding to the sum of Pell digits

$$S(y) \;=\; 0 + 1y + 2y^2 + 1y^3 + 2y^4 + 3y^5 + 2y^6 + 1y^7 + 2y^8 + 3y^9 + 2y^{10} + 3y^{11} + 4y^{12} + \ldots \qquad (36.12\text{-}6)$$

can be computed via the iteration (see section 36.7 on page 704)

$$\begin{aligned}
L_0 &= 0, \qquad R_0 = 0 + y + 2\,y^2, \qquad l_0 = y, \qquad r_0 = y && (36.12\text{-}7a)\\
A_0 &= 1, \qquad B_0 = 1 + y + y^2 && (36.12\text{-}7b)\\
l_{n+1} &= r_n && (36.12\text{-}7c)\\
r_{n+1} &= r_n^2\, l_n && (36.12\text{-}7d)\\
L_{n+1} &= R_n && (36.12\text{-}7e)\\
R_{n+1} &= R_n + r_{n+1}\,(R_n + B_n) + r_{n+1}^2\,(L_n + 2\,A_n) \quad \to S(y) && (36.12\text{-}7f)\\
A_{n+1} &= B_n && (36.12\text{-}7g)\\
B_{n+1} &= B_n + r_{n+1}\,B_n + r_{n+1}^2\,A_n && (36.12\text{-}7h)
\end{aligned}$$

Implementation in pari/gp:

```
fs(y, N=8)=
{
    local(t, yr, yl, L, R, Lp, Rp, Li, Ri);
    L =  0;  R = 0+y+2*y^2;
    Li = 1;  Ri = 1+y+y^2;
    yl = y;  yr = y;
    for(k=1,N,
      t=yr; yr*=yr*yl; yl=t;
      Lp=R;  Rp=R+yr*(R+Ri)+yr^2*(L+2*Li);  L=Lp; R=Rp;
      Lp=Ri; Rp=Ri+yr*Ri+yr^2*Li;   Li=Lp; Ri=Rp;
    );
    return( R )
}
```

The series coefficients grow slowly, so the first few of them can nicely be displayed as

$$S\left(\frac{1}{10}\right) \;=\; 0.12123212323432341232343234345434523434541232343234345 \ldots \qquad (36.12\text{-}8)$$

36.12.3 Pell Gray code

Figure 36.12-B gives a Gray code for the Pell representations. The Gray code can be constructed recursively as shown in figure 36.12-C. In the algorithm each block is split into a left and a right part (indicated by the '^'. The next block is created by appending to the current block its reverse with ones on top and appending the left part with twos on top. The iteration can actually be started with a block of a single

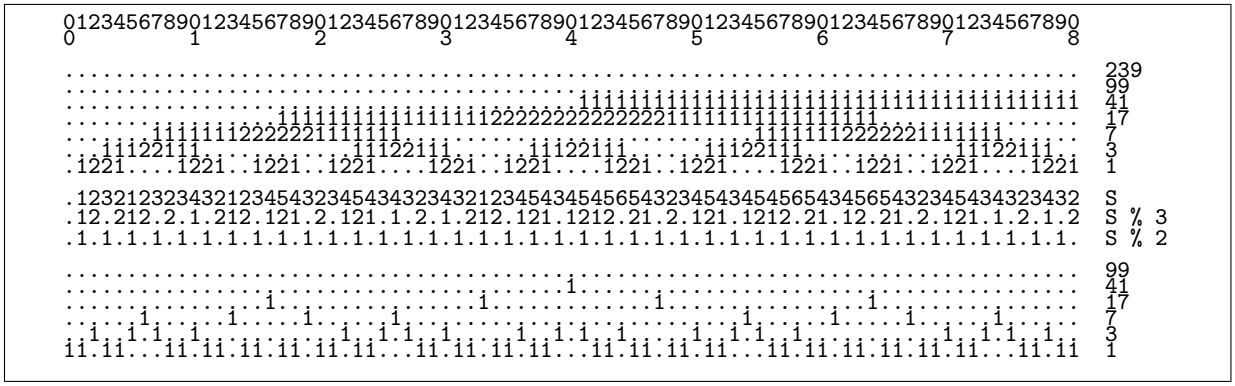


Figure 36.12-B: A Gray code for Pell representations. A dot is used for zero. The three following lines are the sum of digits and the sum of digits modulo three and two. The sequence is $0, 1, 2, 5, 4, 3, 6, 13, 10, 11, 12, 9, 8, 7, 14 \ldots$, the difference between successive elements is a Pell number. The lowest block gives the Pell representations of the (absolute) differences, the *Pell ruler function*.

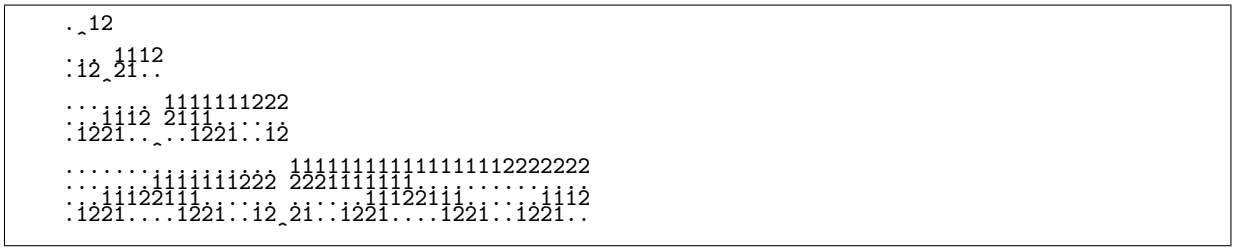


Figure 36.12-C: Construction for a Gray code for Pell representations.

zero (the left part being also a single zero). This is done in the following algorithm.

$$F_0 = 0, \quad F'_0 = 0, \quad B_0 = 0, \quad B'_0 = 0 \tag{36.12-9a}$$

$$I_0 = 1, \quad I'_0 = 1, \quad Y_0 = y, \quad Y'_0 = y \tag{36.12-9b}$$

$$b_n = 4^{n-1}, \quad c_n = 2\,b_n \tag{36.12-9c}$$

$$F_{n+1} = (F_n \qquad) + Y_n\,(B_n + b_n\,I_n) + \; Y_n^2\,(F'_n + c_n\,I'_n) \quad \to G_b(y) \tag{36.12-9d}$$

$$B_{n+1} = (B'_n + c_n\,I'_n) + Y'_n\,(F_n + b_n\,I_n) + Y'_n\,Y_n\,(B_n \qquad) \tag{36.12-9e}$$

$$I_{n+1} = I_n + Y_n\,I_n + Y_n^2\,I'_n \tag{36.12-9f}$$

$$Y_{n+1} = Y_n^2\,Y'_n \tag{36.12-9g}$$

$$F'_{n+1} = F_n, \quad B'_{n+1} = B_n \tag{36.12-9h}$$

$$I'_{n+1} = I_n, \quad Y'_{n+1} = Y_n \tag{36.12-9i}$$

Implementation in pari/gp:

```
pgr(y, N=11)= /* Pell Gray code */
{
    local(iir, iil, yl, yr, Fl, F, Bl, B, b, c);
    local(t, tf, tb);
    /* correct up to order pell(N+1)-1 */
    F=0;  Fl=0;   B=0;  Bl=0;
    iil=1;  iir=1;  yl=y;  yr=y;
    for(k=1, N,
      b = 4^(k-1);  c = 2*b;  /* b = pell(k);*/
      tf = (F         ) +  yr * (B + b*iir) + yr*yr * (Fl + c*iil);
      tb = (Bl + c*iil) +  yl * (F + b*iir) + yl*yr * (B         );
      Fl = F;   Bl = B;
      F = tf;   B = tb;
      t = iir;  iir += yr*(iir + yr*iil);  iil = t;
      t = yr;  yr *= (yr*yl);  yl = t;
    );
```

```
 ----- k = 0
yl = (y)  yr = (y)
iir = 1
iil = 1
F = 0
B = 0
 ----- k = 1    b=1
yl = (y)  yr = (y^3)
iir = (y^2 + y + 1)
iil = 1
F = (2 y^2 + y)
B = (y + 2)
 ----- k = 2    b=4
yl = (y^3)  yr = (y^7)
iir = (y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
iil = (y^2 + y + 1)
F = (8 y^6 + 4 y^5 + 5 y^4 + 6 y^3 + 2 y^2 + y)
B = (y^5 + 2 y^4 + 6 y^3 + 5 y^2 + 4 y + 8)
 ----- k = 3   b=16
yl = (y^7)  yr = (y^17)
iir = (y^16 + y^15 + y^14 + y^13 + y^12 + y^11 + y^10 + y^9
      + y^8 + y^7 + y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
iil = (y^6 + y^5 + y^4 + y^3 + y^2 + y + 1)
F = (34 y^16 + 33 y^15 + 32 y^14
      + 16 y^13 + 17 y^12 + 18 y^11 + 22 y^10 + 21 y^9 + 20 y^8 + 24 y^7
      + 8 y^6 + 4 y^5 + 5 y^4 + 6 y^3 + 2 y^2 + y)
B = (y^15 + 2 y^14 + 6 y^13 + 5 y^12 + 4 y^11 + 8 y^10
      + 24 y^9 + 20 y^8 + 21 y^7 + 22 y^6 + 18 y^5 + 17 y^4 + 16 y^3
      + 32 y^2 + 33 y + 34)
```

Figure 36.12-D: Quantities with the computation of the series related to the Pell Gray code.

```
    return( F )
}
```

It is instructive to look at the variables in the first few steps of the iteration, see figure 36.12-D.

The Taylor series for $G_2(y)$ is

$$\begin{aligned} G_2(y) &= 0 + 1y + 2y^2 + 6y^3 + 5y^4 + 4y^5 + 8y^6 + 24y^7 + 20y^8 + 21y^9 + 22y^{10} + 18y^{11} + \\ &\quad +17y^{12} + 16y^{13} + 32y^{14} + 33y^{15} + 34y^{16} + 98y^{17} + 97y^{18} + 96y^{19} + 80y^{20} + 81y^{21} + \\ &\quad +82y^{22} + 86y^{23} + 85y^{24} + 84y^{25} + 88y^{26} + 72y^{27} + 68y^{28} + 69y^{29} + 70y^{30} + 66y^{31} + \ldots \end{aligned} \tag{36.12-10}$$

The coefficients corresponds to the Pell representations interpreted as binary numbers, each Pell-digit occupying two bits (figure 36.12-B).

When relation 36.12-9c on the previous page is changed to $b_n = P_n$ (indicated in the code, the function can be defined as `pell(k)=if(k<=1, 1, return(2*pell(k-1)+pell(k-2)));`) then the function $G_P(y)$ which has the Pell Gray code sequence as coefficients is computed:

$$\begin{aligned} G_P(y) &= 0 + 1y + 2y^2 + 5y^3 + 4y^4 + 3y^5 + 6y^6 + 13y^7 + 10y^8 + 11y^9 + 12y^{10} + 9y^{11} + \\ &\quad +8y^{12} + 7y^{13} + 14y^{14} + 15y^{15} + 16y^{16} + 33y^{17} + 32y^{18} + 31y^{19} + 24y^{20} + 25y^{21} + \\ &\quad +26y^{22} + 29y^{23} + 28y^{24} + 27y^{25} + 30y^{26} + 23y^{27} + 20y^{28} + 21y^{29} + 22y^{30} + 19y^{31} + \ldots \end{aligned} \tag{36.12-11}$$

Section 12.4 on page 288 gives a recursive algorithm to compute the words of the Pell Gray code.

Define the *Pell Gray code constant* as

$$G_P = G_P\left(\frac{1}{2}\right) \tag{36.12-12a}$$

$$= 2.245567348365072195720956572438998819867495229140192012\ldots \tag{36.12-12b}$$

$$[\text{base } 2] = 10.001111101101110110000000011100100011000110000100001111\ldots \tag{36.12-12c}$$

$$[\text{CF}] = [2, 4, 13, 1, 5, 1, 1, 1, 27, 1, 9, 1, 3, 8, 1, 2, 1, 1, 3, 14, 1, 8, 1, 1, 6, 3, 1, 1, 1, 2, 1, 7, 210, 1, 1, 3, 2, 1, 1, 10, 1, 1, 6, 1, 1, 2, 1, 2, 1, 4, 6, 12, 1, \ldots]$$

Setting $b_n = 1,\ c_n = 2$ in the algorithm gives a function whose series coefficients are the sum of Pell Gray code digits:

$$G_{[1,2]}\left(\frac{1}{10}\right) \quad = \quad 0.123212323432123454323454343234321234543454565432345434 5\ldots \quad (36.12\text{-}13)$$

Using $b_n = 1,\ c_n = 0$ counts the ones in the Pell Gray code:

$$G_{[1,0]}\left(\frac{1}{10}\right) \quad = \quad 0.101210123212101012123234321232121010121012321212323432 3\ldots \quad (36.12\text{-}14)$$

while $b_n = 0,\ c_n = 1$ counts the twos:

$$G_{[0,1]}\left(\frac{1}{10}\right) \quad = \quad 0.011001100110011221100110011001100112211221122110011001100 110\ldots \quad (36.12\text{-}15)$$

The continued fraction of this constant has very large terms:

```
[0, 90, 1, 8, 1, 5501100164, 8, 5, 3, 2, 19, 2, 1, 2, 2, 1, 1, 5, 1,
54, 6, 1, 5, 22, 2, 6, 2, 2, 1, 22445, 1, 45, 2, 2, 5, 1, 5, 1, 8,
5446094555544609447167274, 1, 5, 2, 2, 7, 1, 1, 1, 2, 1, 27, 2, 2,
1, 17, 1, 1, 1, 1, 4, 2, 1, 4, 3, 3, 3, 1, 3, 1, 2, 1, 1, 29, 1, ...]
```

Finally, $(1-y)\,G_P(y)$ gives the (signed) Pell ruler function, the first 100 series coefficients are:

```
 0 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +17 -1 -1 -7 +1 +1
+3 -1 -1 +3 -7 -3 +1 +1 -3 -1 -1 +17 +1 +1 +3 -1 -1 +3 +41 -3 +1 +1
-3 -1 -1 -17 +1 +1 +3 -1 -1 +3 +7 -3 +1 +1 -3 -1 -1 +7 +1 +1 -17 -1
-1 -7 +1 +1 +3 -1 -1 +3 -7 -3 +1 +1 -3 -1 -1 +41 +1 +1 +3 -1 -1 +3
+7 -3 +1 +1 -3 -1 -1 +7 +1 +1 +99 -1
```

Part V

Algorithms for finite fields

Chapter 37

Modular arithmetic and some number theory

This chapter introduces some basic concepts of number theory. We start by implementing the arithmetical operations modulo m. Basic concepts of number theory, like the order of an element, quadratic residues and primitive roots are introduced. Selected algorithms such as the Rabin-Miller compositeness test and several primality tests are presented. The chapter ends with recreational material, such as multigrades and solutions of certain Diophantine equations.

Modular arithmetic and the concepts of number theory are fundamental for many areas like cryptography, error correcting codes, and signal processing.

37.1 Implementation of the arithmetic operations

The first part in the implementation of modular arithmetics consists of the implementing the operations addition, subtraction, multiplication, powering and division. However, for a practically useful C++ class one also has to implement several other routines like the determination of the order of an element (which involves integer factorization and computation of Euler's φ) computation of square roots and elements of given order.

37.1.1 Addition and subtraction

Addition and subtraction modulo m can easily be implemented as [FXT: mod/modarith.h]:

```
inline umod_t sub_mod(umod_t a, umod_t b, umod_t m)
{
    if ( a>=b )  return  a - b;
    else         return  m - b + a;
}

inline umod_t add_mod(umod_t a, umod_t b, umod_t m)
{
    if ( 0==b )  return  a;
    // return sub_mod(a, m-b, m);
    b = m - b;
    if ( a>=b )  return  a - b;
    else         return  m - b + a;
}
```

The type `umod_t` is an unsigned 64-bit integer. Care has been taken to avoid any overflow of intermediate results. A 'set'-, increment-, decrement- and negation function will further be useful:

```
inline umod_t set_mod(umod_t x, umod_t m)
{ if ( x>=m )  x %= m;  return x; }

inline umod_t incr_mod(umod_t a, umod_t m)
{ a++; if ( a==m )  a = 0;   return a; }

inline umod_t decr_mod(umod_t a, umod_t m)
{ if ( a==0 )  a = m - 1;  else a--;  return a; }

inline umod_t neg_mod(umod_t b, umod_t m)
{ if ( 0==b )  return 0;  else  return m - b; }
```

Two addition tables for the moduli 13 and 9 are shown in figure 37.1-A

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12	0
2	2	3	4	5	6	7	8	9	10	11	12	0	1
3	3	4	5	6	7	8	9	10	11	12	0	1	2
4	4	5	6	7	8	9	10	11	12	0	1	2	3
5	5	6	7	8	9	10	11	12	0	1	2	3	4
6	6	7	8	9	10	11	12	0	1	2	3	4	5
7	7	8	9	10	11	12	0	1	2	3	4	5	6
8	8	9	10	11	12	0	1	2	3	4	5	6	7
9	9	10	11	12	0	1	2	3	4	5	6	7	8
10	10	11	12	0	1	2	3	4	5	6	7	8	9
11	11	12	0	1	2	3	4	5	6	7	8	9	10
12	12	0	1	2	3	4	5	6	7	8	9	10	11

	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8	0
2	2	3	4	5	6	7	8	0	1
3	3	4	5	6	7	8	0	1	2
4	4	5	6	7	8	0	1	2	3
5	5	6	7	8	0	1	2	3	4
6	6	7	8	0	1	2	3	4	5
7	7	8	0	1	2	3	4	5	6
8	8	0	1	2	3	4	5	6	7

Figure 37.1-A: Addition modulo 13 (left) and modulo 9 (right).

37.1.2 Multiplication

Multiplication is a bit harder: if one would use something like

```
inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    return  (a * b) % m;
}
```

Then the modulus would be restricted to half of the word size.

One can use almost all bits for the modulus if the following trick is used. The technique also works if the product $a \cdot b$ does not fit into a machine integer.

Let $\langle x \rangle_y$ denote x modulo y, let $\lfloor x \rfloor$ denote the integer part of x. For $0 \leq a, b < m$:

$$a \cdot b \quad = \quad \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m + \langle a \cdot b \rangle_m \tag{37.1-1}$$

Rearranging and taking both sides modulo $z > m$ (where $z = 2^k$ on a k-bit machine):

$$\left\langle a \cdot b - \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \quad = \quad \langle \langle a \cdot b \rangle_m \rangle_z \tag{37.1-2}$$

The right hand side equals $\langle a \cdot b \rangle_m$ because $m < z$.

$$\langle a \cdot b \rangle_m \quad = \quad \left\langle \langle a \cdot b \rangle_z - \left\langle \left\lfloor \frac{a \cdot b}{m} \right\rfloor \cdot m \right\rangle_z \right\rangle_z \tag{37.1-3}$$

The expression on the right can be translated into a few lines of C-code. The code given here assumes that one has 64-bit integer types `int64` (signed) and `uint64` (unsigned) and a floating point type with 64-bit mantissa, `float64` (typically `long double`).

```
uint64 mul_mod(uint64 a, uint64 b, uint64 m)
{
    uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);  // floor(a*b/m)
    y = y * m;           // m*floor(a*b/m) mod z
    uint64 x = a * b;    // a*b mod z
    uint64 r = x - y;    // a*b mod z - m*floor(a*b/m) mod z
    if ( (int64)r < 0 )  // normalization needed ?
    {
        r = r + m;
        y = y - 1;       // (a*b)/m  quotient, omit line if not needed
    }
    return  r;           // (a*b)%m  remnant
}
```

The technique uses the fact that integer multiplication computes the least significant bits of the result $\langle a \cdot b \rangle_z$ whereas float multiplication computes the most significant bits of the result. The above routine works if $0 <= a, b < m < 2^{63} = \frac{z}{2}$. The normalization is not necessary if $m < 2^{62} = \frac{z}{4}$.

When working with a fixed modulus the division by `p` may be replaced by a multiplication with the inverse modulus, that only needs to be computed once:

precompute: `float64 i = (float64)1/m;`

and replace the line `uint64 y = (uint64)((float64)a*(float64)b/m+(float64)1/2);`

by `uint64 y = (uint64)((float64)a*(float64)b*i+(float64)1/2);`

so any division inside the routine is avoided. Beware that the routine cannot be used for $m >= 2^{62}$: it very rarely fails for moduli of more than 62 bits, due to the additional error when inverting and multiplying as compared to dividing alone. This technique is ascribed to Peter Montgomery. An implementation in is [FXT: mod/modarith.h]:

```
inline umod_t mul_mod(umod_t a, umod_t b, umod_t m)
{
    umod_t x = a * b;
    umod_t y = m * (umod_t)( (ldouble)a * (ldouble)b/m + (ldouble)1/2 );
    umod_t r = x - y;
    if ( (smod_t)r < 0 )  r += m;
    return  r;
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	4	6	8	1	3	5	7
3	0	3	6	0	3	6	0	3	6
4	0	4	8	3	7	2	6	1	5
5	0	5	1	6	2	7	3	8	4
6	0	6	3	0	6	3	0	6	3
7	0	7	5	3	1	8	6	4	2
8	0	8	7	6	5	4	3	2	1

Figure 37.1-B: Multiplication modulo 13 (left) and modulo 9 (right).

Two multiplication tables for the moduli 13 and 9 are shown in figure 37.1-B. Note that for the modulus 9 some products $a \cdot b$ are zero though neither of a or b is zero. The tables can be obtained with [FXT: mod/modarithtables-demo.cc].

37.1.3 Exponentiation

The algorithm used for exponentiation (powering) is the binary exponentiation algorithm shown in section 27.6 on page 537:

```
inline umod_t pow_mod(umod_t a, umod_t e, umod_t m)
// Right-to-left scan
{
    if ( 0==e )  { return 1; }
    else
    {
        umod_t z = a;
        umod_t y = 1;
        while ( 1 )
        {
            if ( e&1 )  y = mul_mod(y, z, m);  // y *= z;
            e >>= 1;
            if ( 0==e )  break;
            z = sqr_mod(z, m);  // z *= z;
        }
        return  y;
    }
}
```

37.1.4 Inversion and division

Subtraction is the inverse of addition. In order to subtract an element b from another element a one can add the *additive inverse* $-b := m - b$ to a. Each element has an additive inverse.

Division is the inverse of multiplication. In order to divide an element a by another element b one can multiply a by the *multiplicative inverse*. But not all elements have a multiplicative inverse, only those elements b that are coprime to the modulus m (that is, $\gcd(b, m) = 1$). These elements are called *invertible* (modulo m) or *units*. For a prime modulus all elements except zero are invertible.

The computation of the GCD uses the *Euclidean algorithm* [FXT: mod/gcd.h]:

```
template <typename Type>
Type gcd(Type a, Type b)
// Return greatest common divisor of a and b.
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;
    Type r;
    do
    {
        r = a % b;
        a = b;
        b = r;
    }
    while ( r!=0 );
    return a;
}
```

A variant of the algorithm that avoids most of the (expensive) modular reductions is called the *binary GCD algorithm* [FXT: mod/binarygcd.h]:

```
template <typename Type>
Type binary_ugcd(Type a, Type b)
// Return greatest common divisor of a and b.
// Version for unsigned types.
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;

    Type r = a % b;
    a = b;
    b = r;
    if ( b==0 )  return a;

    ulong k = 0;
    while ( !((a|b)&1) )  // both even
    {
```

```
        k++;
        a >>= 1;
        b >>= 1;
    }

    while ( !(a&1) )  a >>= 1;
    while ( !(b&1) )  b >>= 1;

    while ( 1 )
    {
        if ( a==b )  return  a << k;

        if ( a < b )  swap2(a, b);
        Type t = (a-b) >> 1;  // t>0

        while ( !(t&1) )  t >>= 1;
        a = t;
    }
}
```

The complexity of this algorithm for N-bit numbers is $O(N^2)$. We note that an $O(N \log(N))$ algorithm is given in [220].

The *least common multiple* (LCM) of two numbers is

$$\operatorname{lcm}(a,\, b) \quad = \quad \frac{a \cdot b}{\gcd(a,\, b)} \quad = \quad \left(\frac{a}{\gcd(a,\, b)} \right) \cdot b \tag{37.1-4}$$

The latter form avoids overflow when using integer types of fixed size.

For modular inversion one can use the extended Euclidean algorithm (EGCD), which for two integers a and b finds $d = \gcd(a, b)$ and u, v so that $a\,u + b\,v = d$. Applying EGCD to b and m where $\gcd(b, m) = 1$ one obtains u and v so that $m\,u + b\,v = 1$. That is: $b\,v \equiv 1 \pmod{m}$, so v is the inverse of b modulo m and $a/b := a\,b^{-1} = a\,v$.

The following code implements the EGCD algorithm as given in [155]:

```
template <typename Type>
Type egcd(Type u, Type v, Type &tu1, Type &tu2)
// Return u3 and set u1,v1 so that
//   gcd(u,v) == u3 == u*u1 + v*u2
// Type must be a signed type.
{
    Type u1 = 1,  u2 = 0;
    Type v1 = 0,  v3 = v;
    Type u3 = u,  v2 = 1;
    while ( v3!=0 )
    {
        Type q = u3 / v3;

        Type t1 = u1 - v1 * q;
        u1 = v1;  v1 = t1;

        Type t3 = u3 - v3 * q;
        u3 = v3;  v3 = t3;

        Type t2 = u2 - v2 * q;
        u2 = v2;  v2 = t2;
    }
    tu1 = u1;  tu2 = u2;
    return u3;
}
```

Another algorithm for the computation of the modular inversion uses exponentiation. It is given only after the concept of the *order* of an element has been introduced (section 37.4 on page 739).

37.2 Modular reduction with structured primes

The modular reduction with Mersenne primes $M = 2^k - 1$ is especially easy: let u and v be in the range $0 \le u, v < M = 2^k - 1$, then with the ordinary (non-reduced) product written as $u\,v = 2^k\,r + s$ (where $0 \le r, s < M = 2^k - 1$) the reduction is simply $u\,v \equiv r + s \pmod{M}$.

```
2^64-2^32+1 == 2^(32*2)-2^(32*1)+1
2^96-2^32+1 == 2^(32*3)-2^(32*1)+1
2^224-2^96+1 == 2^(32*7)-2^(32*3)+1
2^320-2^288+1 == 2^(32*10)-2^(32*9)+1
2^512-2^32+1 == 2^(32*16)-2^(32*1)+1
2^512-2^288+1 == 2^(32*16)-2^(32*9)+1
2^544-2^32+1 == 2^(32*17)-2^(32*1)+1
2^544-2^96+1 == 2^(32*17)-2^(32*3)+1
2^576-2^512+1 == 2^(64*9)-2^(64*8)+1
2^672-2^192+1 == 2^(32*21)-2^(32*6)+1
2^832-2^448+1 == 2^(64*13)-2^(64*7)+1
2^992-2^832+1 == 2^(32*31)-2^(32*26)+1
2^1088-2^608+1 == 2^(32*34)-2^(32*19)+1
2^1184-2^768+1 == 2^(32*37)-2^(32*24)+1
2^1376-2^32+1 == 2^(32*43)-2^(32*1)+1
2^1664-2^256+1 == 2^(128*13)-2^(128*2)+1
2^1856-2^1056+1 == 2^(32*58)-2^(32*33)+1
2^1920-2^384+1 == 2^(128*15)-2^(128*3)+1
2^1984-2^544+1 == 2^(32*62)-2^(32*17)+1
```

```
2^80-2^48+1 == 2^(16*5)-2^(16*3)+1
2^176-2^48+1 == 2^(16*11)-2^(16*3)+1
2^176-2^80+1 == 2^(16*11)-2^(16*5)+1
2^368-2^336+1 == 2^(16*23)-2^(16*21)+1
2^384-2^80+1 == 2^(16*24)-2^(16*5)+1
2^400-2^160+1 == 2^(16*25)-2^(16*10)+1
2^528-2^336+1 == 2^(16*33)-2^(16*21)+1
2^544-2^304+1 == 2^(16*34)-2^(16*19)+1
2^560-2^112+1 == 2^(16*35)-2^(16*7)+1
2^576-2^240+1 == 2^(16*36)-2^(16*15)+1
2^672-2^560+1 == 2^(16*42)-2^(16*35)+1
2^688-2^96+1 == 2^(16*43)-2^(16*6)+1
2^784-2^48+1 == 2^(16*49)-2^(16*3)+1
2^832-2^432+1 == 2^(16*52)-2^(16*27)+1
2^880-2^368+1 == 2^(16*55)-2^(16*23)+1
2^912-2^32+1 == 2^(16*57)-2^(16*2)+1
2^944-2^784+1 == 2^(16*59)-2^(16*49)+1
2^1008-2^144+1 == 2^(16*63)-2^(16*9)+1
2^1024-2^880+1 == 2^(16*64)-2^(16*55)+1
```

Figure 37.2-A: The complete list of primes of the form $p = x^k - x^j + 1$ where $x = 2^G$, $G = 2^i$, $G \geq 32$ and p up to 2048 bits (left), and the equivalent list for $x = 2^{16}$ and p up to 1024 bits (right).

```
? M=x^20-x^15+x^10-x^5+1;
? n=poldegree(M);
? P=sum(i=0,2*n-1,eval(Str("p_"i))*x^i)
  p_39*x^39 + p_38*x^38 + [--etc--] + p_3*x^3 + p_2*x^2 + p_1*x + p_0
? R=P%M;
? for(i=0,n-1,print("  ",eval(Str("r_"i))," = ",polcoeff(R,i)))
  r_0 = p_0 + (-p_20 - p_25)
  r_1 = p_1 + (-p_21 - p_26)
  r_2 = p_2 + (-p_22 - p_27)
  r_3 = p_3 + (-p_23 - p_28)
  r_4 = p_4 + (-p_24 - p_29)
  r_5 = p_5 + (p_20 - p_30)
  r_6 = p_6 + (p_21 - p_31)
  r_7 = p_7 + (p_22 - p_32)
  r_8 = p_8 + (p_23 - p_33)
  r_9 = p_9 + (p_24 - p_34)
  r_10 = p_10 + (-p_20 - p_35)
  r_11 = p_11 + (-p_21 - p_36)
  r_12 = p_12 + (-p_22 - p_37)
  r_13 = p_13 + (-p_23 - p_38)
  r_14 = p_14 + (-p_24 - p_39)
  r_15 = p_15 + p_20
  r_16 = p_16 + p_21
  r_17 = p_17 + p_22
  r_18 = p_18 + p_23
  r_19 = p_19 + p_24
```

Figure 37.2-B: Computation of the reduction rule for the 640-bit prime $Y_{50}(2^{32})$.

A modular reduction algorithm that uses only shifts, additions and subtractions can be found also for *structured primes* (called *generalized Mersenne primes* in [216]). Let the modulus M be of the form

$$M = \sum_{i=0}^{n} m_i \, x^i \tag{37.2-1}$$

where $x = 2^k$ and the leading coefficient m_n equal to one. For simplicity we further assume that $m_i = \pm 1$ and $m_{n-1} = -1$ (so that the numbers fit into n bits). The reduction algorithm can be found using polynomial arithmetic. Write the non-reduced product P as

$$P = \sum_{i=0}^{2n-1} p_i \, x^i \tag{37.2-2}$$

where $0 \le p_i < x$. Then the reduced product R is

$$R = \sum_{i=0}^{n-1} r_i \, x^i \; := \; P \pmod{M} \tag{37.2-3}$$

where $0 \le r_i < x$. A script that will print the rule for moduli of the form $x^k - x^j + 1$ is

```
M=x^k - x^j + 1  \\ modulus as polynomial
n=poldegree(M);
P=sum(i=0, 2*n-1, eval(Str("p_" i)) * x^i) \\ unreduced product
R = P % M;  \\ reduced product
\\ print rules:
for (i=0, n-1, print("  ",eval(Str("r_" i)), " = ", polcoeff(R, i)))
```

With $k = 3$ and $j = 2$ we obtain the reduction rules (last three lines)

```
? k=3;j=2;
? M=x^k-x^j+1
  x^3 - x^2 + 1
? n=poldegree(M);
? P=sum(i=0,2*n-1,eval(Str("p_"i))*x^i)
  p_5*x^5 + p_4*x^4 + p_3*x^3 + p_2*x^2 + p_1*x + p_0
? R=P%M;
? for(i=0,n-1,print("  ",eval(Str("r_"i))," = ",polcoeff(R,i)))
  r_0 = p_0 + (-p_3 + (-p_4 - p_5))
  r_1 = p_1 + (-p_4 - p_5)
  r_2 = p_2 + (p_3 + p_4)
```

A list of primes of the form $p = x^k - x^j + 1$ where $x = 2^G$, G a power of two and $G \ge 16$ is shown in figure 37.2-A. The equivalent list with i a multiple of 8 is given in [FXT: data/structured-primes-2k2j1.txt]. The primes allow radix-2 number theoretic transforms up to a length of x^j.

Structured primes that are evaluations of cyclotomic polynomials are given in section 37.11.4.7 on page 768. The reduction rule for the 640-bit prime $M = Y_{50}(2^{32})$ is shown in figure 37.2-B. There is a choice for the 'granularity' of the rule: the modulus also equals $Y_{10}(2^{5 \cdot 32})$, so we can obtain the reduction rule for groups of 5 machine words

```
? M=x^4-x^3+x^2-x^1+1;
  [--snip--]
? for(i=0,n-1,print("  ",eval(Str("r_"i))," = ",polcoeff(R,i)))
  r_0 = p_0 + (-p_4 - p_5)
  r_1 = p_1 + (p_4 - p_6)
  r_2 = p_2 + (-p_4 - p_7)
  r_3 = p_3 + p_4
```

However, the rule in terms of single words seems to be more appropriate as it allows for easier code generation.

37.3 The sieve of Eratosthenes

Several number theoretic algorithms can take advantage of a precomputed list of primes. A simple and quite efficient algorithm, called the *sieve of Eratosthenes* computes all primes up to a given limit. It uses a tag-array where all entries ≥ 2 are initially marked as potential primes. The algorithm proceeds by searching for the next marked entry and deleting all multiples of it.

An implementation that uses the `bitarray` class (see section 4.6 on page 152) is given in [FXT: mod/eratosthenes-demo.cc]:

```
void
eratosthenes(bitarray &ba)
{
    ba.set_all();
    ba.clear(0);
    ba.clear(1);
    ulong n = ba.n_;
    ulong k = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        for (ulong j=2, i=j*k;  i<n;  ++j, i=j*k)  ba.clear(i);
    }
}
```

The program prints the resulting list of primes (code slightly simplified):

```
int
main(int argc, char **argv)
{
    ulong n = 100;
    NXARG(n, "Upper limit for prime search");

    bitarray ba(n);
    eratosthenes(ba);

    ulong k = 0;
    ulong ct = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        ++ct;
        cout << " " << k;
    }
    cout << endl;

    cout << "Found " << ct << " primes below " << n << "." << endl;

    return 0;
}
```

The output for the default ($n = 100$) is:

```
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Found 25 primes below 100.
```

A little thought leads to a faster variant: when deleting the multiples $k \cdot p$ of the prime p from the list we only need to care about the values of k that are greater than all primes found so far. Further, values $k \cdot p$ containing only prime factors smaller than p have already been deleted. That is, we only need to delete the values $\{p^2, p^2 + p, p^2 + 2p, p^2 + 3p, \ldots\}$. If we further extract the loop for the prime 2 then for the odd primes, we need to delete only the values $\{p^2, p^2 + 2p, p^2 + 4p, \ldots\}$.

The implementation is

```
void
eratosthenes_opt(bitarray &ba)
{
    ba.set_all();
    ba.clear(0);
    ba.clear(1);
    ulong n = ba.n_;
    for (ulong k=4; k<n; k+=2)  ba.clear(k);
    ulong r = isqrt(n);
    ulong k = 0;
    while ( (k=ba.next_set(k+1)) < n )
    {
        if ( k > r )  break;
```

```
        for (ulong j=k*k; j<n; j+=k*2)  ba.clear(j);
    }
}
```

The routine is included in the demo, the second argument chooses whether the optimized routine is used.

When computing the primes up to a limit N then about N/p values are deleted after finding the prime p. If we slightly overestimate the computational work W by

$$W \approx N \sum_{p<N,\, prime} \frac{1}{p} \tag{37.3-1}$$

then $W \approx N \log(\log(N))$ which is almost linear. Practically, much of the time used with larger values of N is lost waiting for memory access. Thereby, further improvements should rather address machines specific optimizations than additional algorithmic refinements.

The described algorithmic improvement can be deduced from the series acceleration of the Lambert series

$$\sum_{k>0} d(k)\, x^k = \sum_{k>0} \frac{x^k}{1-x^k} = \sum_{k>0}\sum_{j>0} x^{kj} = \sum_{k>0} \frac{1+x^k}{1-x^k}\, x^{k^2} \tag{37.3-2}$$

where $d(k)$ is the sum of the divisors of k. We note a relation from [156, p.644, ex.27]:

$$\sum_{k>0} \frac{x^k}{1-x^k} = \sum_{k>0} k\, x^k \left(1-x^{k+1}\right)\left(1-x^{k+2}\right)\left(1-x^{k+3}\right)\ldots \tag{37.3-3}$$

One can save half of the space by recording only the odd primes. A C++ implementation of the modified algorithm is [FXT: `make_oddprime_bitarray()` in mod/eratosthenes.cc]. The corresponding table is created upon startup of programs linking the FXT-library. The data can be used to verify the primality of small numbers [FXT: `is_small_prime()` in mod/primes.cc]. The function `next_small_prime()` in the same file uses the data to return the next prime greater or equal to its argument or zero if the argument is too big.

37.4 The order of an element

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 <--= exponent
-----------------------------------------  [order]
 0   1  0  0  0  0  0  0  0  0  0  0  0  0   [ --]
 1   1  1  1  1  1  1  1  1  1  1  1  1  1   [  1]
 2   1  2  4  8  3  6 12 11  9  5 10  7  1   [ 12]
 3   1  3  9  1  3  9  1  3  9  1  3  9  1   [  3]
 4   1  4  3 12  9 10  1  4  3 12  9 10  1   [  6]
 5   1  5 12  8  1  5 12  8  1  5 12  8  1   [  4]
 6   1  6 10  8  9  2 12  7  3  5  4 11  1   [ 12]
 7   1  7 10  5  9 11 12  6  3  8  4  2  1   [ 12]
 8   1  8 12  5  1  8 12  5  1  8 12  5  1   [  4]
 9   1  9  3  1  9  3  1  9  3  1  9  3  1   [  3]
10   1 10  9 12  3  4  1 10  9 12  3  4  1   [  6]
11   1 11  4  5  3  7 12  2  9  8 10  6  1   [ 12]
12   1 12  1 12  1 12  1 12  1 12  1 12  1   [  2]
```

Figure 37.4-A: Powers and orders modulo 13, the maximal order is $R(13) = 12 = \varphi(13)$.

The (multiplicative) *order* $r = \mathrm{ord}(a)$ of an element a is the smallest positive exponent so that $a^r = 1$. For elements that are not invertible ($\gcd(a, m) \neq 1$) the order is not defined. Figure 37.4-A shows the

	0	1	2	3	4	5	6	7	8	<--= exponent
										[order]
0	1	0	0	0	0	0	0	0	0	[--]
1	1	1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	7	5	1	2	4	[6]
3	1	3	0	0	0	0	0	0	0	[--]
4	1	4	7	1	4	7	1	4	7	[3]
5	1	5	7	8	4	2	1	5	7	[6]
6	1	6	0	0	0	0	0	0	0	[--]
7	1	7	4	1	7	4	1	7	4	[3]
8	1	8	1	8	1	8	1	8	1	[2]

	0	1	2	3	4	5	6	<--= exponent
								[order]
1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	7	5	1	[6]
4	1	4	7	1	4	7	1	[3]
5	1	5	7	8	4	2	1	[6]
7	1	7	4	1	7	4	1	[3]
8	1	8	1	8	1	8	1	[2]

Figure 37.4-B: Powers and orders modulo 9 (left), the maximal order is $R(9) = 6 = \varphi(9)$. The order modulo m is defined only for elements a where $\gcd(a, m) = 1$. The table of powers for the group of units $(\mathbb{Z}/9\mathbb{Z})^*$ (right) is obtained by dropping all elements for which the order is undefined.

powers of all elements modulo the prime 13. The rightmost column gives the order of those elements that are invertible.

An element a whose r-th power equals one is called an *r-th root of unity*: $a^r = 1$. Modulo 9 both elements 2 and 4 are a 6th roots of unity, see figure 37.4-B.

If $a^r = 1$ but $a^x \neq 1$ for all $x < r$ then a is called a *primitive r-th root of unity*. Modulo 9 the element 2 is a primitive 6th root of unity; the element 4 is not, it is a primitive 3rd root of unity. An element of order r is an r-th primitive root of unity.

The *maximal order* $R(m)$ is the maximum of the orders of all elements for a fixed modulus m:

$$R(m) \quad := \quad \max_{a \in \mathbb{Z}/m\mathbb{Z}^*} (\mathrm{ord}(a)) \tag{37.4-1}$$

For prime modulus p the maximal order equals $R(p) = p - 1$. When it cannot cause confusion we omit the argument to the maximal order in what follows.

An element of maximal order is a R-th primitive root of unity. Roots of unity of an order different from R are available only for the divisors d_i of R: if g is an element of maximal order R then g^{R/d_i} has order d_i (is a primitive d_i-th root of unity):

$$\mathrm{ord}\left(g^{R/d_i}\right) \quad = \quad d_i \tag{37.4-2}$$

This is because $(g^{R/d_i})^{d_i} = g^R = 1$ and $(g^{R/d_i})^k \neq 1$ for $k < d_i$.

The factor by which the order of an element falls short of the maximal order is sometimes called the *index* of the element. Let i be the index and r the order, then $i \cdot r = R$.

The concept of the order comes from group theory. The invertible elements modulo m with multiplication form a group. The neutral element is one. The order defined above is the order in this group, it tells us how often one has to multiply the element a to one to obtain one. We restrict orders to positive values, else every element would have order zero.

With addition things are simpler: all elements with addition form a group with zero as neutral element. The order of an element a in this group tells us how often one has to add a to zero to obtain zero. The order here is simply $m/\gcd(a, m)$. All elements coprime to m (and especially 1 and -1) are generators of the additive group.

The maximal order R of all elements of a group is sometimes called the *exponent of the group*. Elements of maximal order are also called *primitive elements* of the group, *primitive roots* of the group, or *generators* of the group.

37.5 Prime modulus: the field $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = \mathrm{GF}(p)$

If the modulus is a prime p then $\mathbb{Z}/p\mathbb{Z}$ is the field $\mathbb{F}_p = \mathrm{GF}(p)$: all elements except 0 have inverses and thereby division is possible in $\mathrm{GF}(p)$. The maximal order R equals $p-1$. Elements of order R are called *primitive roots modulo p*.

If g is a generator, then every element in $\mathrm{GF}(p)$ different from 0 is equal to some power g^e $(1 \le e < p)$ of g and its order is R/e. To test whether g is a primitive n-th root of unity in $\mathrm{GF}(p)$ one does not need to check whether $g^k \neq 1$ for all $k < n$. It suffices to do the check for exponents k that are prime factors of n. This is because the order of any element divides the maximal order.

To find a primitive root in $\mathrm{GF}(p)$ proceed as indicated by the following pseudo code:

```
function primroot(p)
{
    if p==2 then  return 1
    f[] := distinct_prime_factors(p-1)
    for r:=2 to p-1
    {
        x := TRUE
        foreach q in f[]
        {
            if r**((p-1)/q)==1 then x:=FALSE
        }
        if x==TRUE then  return r
    }
    error("no primitive root found")  // p cannot be prime !
}
```

The algorithm is a simple search and might seem ineffective. In practice the root is found after only a few tries. Note that the factorization of $p-1$ must be known. An element of order n in $\mathrm{GF}(p)$ is returned by the following function:

```
function element_of_order(n, p)
{
    R := p-1  // maxorder
    if (R/n)*n != R then  error("order n must divide maxorder p-1")
    r := primroot(p)
    x := r**(R/n)
    return x
}
```

37.6 Composite modulus: the ring $\mathbb{Z}/m\mathbb{Z}$

n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$	n: $\varphi(n)$
1: 1	13: 12	25: 20	37: 36	49: 42	61: 60	73: 72	85: 64
2: 1	14: 6	26: 12	38: 18	50: 20	62: 30	74: 36	86: 42
3: 2	15: 8	27: 18	39: 24	51: 32	63: 36	75: 40	87: 56
4: 2	16: 8	28: 12	40: 16	52: 24	64: 32	76: 36	88: 40
5: 4	17: 16	29: 28	41: 40	53: 52	65: 48	77: 60	89: 88
6: 2	18: 6	30: 8	42: 12	54: 18	66: 20	78: 24	90: 24
7: 6	19: 18	31: 30	43: 42	55: 40	67: 66	79: 78	91: 72
8: 4	20: 8	32: 16	44: 20	56: 24	68: 32	80: 32	92: 44
9: 6	21: 12	33: 20	45: 24	57: 36	69: 44	81: 54	93: 60
10: 4	22: 10	34: 16	46: 22	58: 28	70: 24	82: 40	94: 46
11: 10	23: 22	35: 24	47: 46	59: 58	71: 70	83: 82	95: 72
12: 4	24: 8	36: 12	48: 16	60: 16	72: 24	84: 24	96: 32

Figure 37.6-A: Values of $\varphi(n)$, the number of integers less than n and coprime to n, for $n \le 96$.

In what follows we will need the function φ, the *totient function* (or *Euler's totient function*). The function $\varphi(m)$ counts the number of integers coprime to and less than m:

$$\varphi(m) \quad := \quad \sum_{\substack{1 \le k < m \\ \gcd(k,m)=1}} 1 \tag{37.6-1}$$

The sequence of values $\varphi(n)$ is entry A000010 in [214]. The values of $\varphi(n)$ for $n \le 96$ are shown in figure 37.6-A. For $m = p$ prime one has $\varphi(p) = p - 1$. For m composite $\varphi(m)$ is always less than $m - 1$. For $m = p^k$ a prime power

$$\varphi(p^k) \quad = \quad p^k - p^{k-1} \; = \; p^{k-1}\,(p-1) \tag{37.6-2}$$

The totient function is a *multiplicative* function: one has $\varphi(p_1\,p_2) = \varphi(p_1)\,\varphi(p_2)$ for coprime p_1, p_2: $\gcd(p_1, p_2) = 1$ but p_1, p_2 are not necessarily prime. Thereby, if the factorization of n into distinct prime powers is $n = \prod_i p_i^{e_i}$, then

$$\varphi(n) \quad = \quad \prod_i \varphi(p_i^{e_i}) \tag{37.6-3}$$

An alternative expression for $\varphi(n)$ is

$$\varphi(n) \quad = \quad n \prod_{p_i} \left(1 - \frac{1}{p_i}\right) \qquad \text{where} \quad n = \prod_i p_k^{e_i} \tag{37.6-4}$$

We note a generalization: the number of s-element sets of numbers $\le n$ whose greatest common divisor is coprime to n equals

$$\varphi_s(n) \quad = \quad n^s \prod_{p_i} \left(1 - \frac{1}{p_i^s}\right) \qquad \text{where} \quad n = \prod_i p_i^{e_i} \tag{37.6-5}$$

Pseudo code to compute $\varphi(m)$ for arbitrary m:

```
function euler_phi(m)
{
    {n, p[], x[]} := factorization(m)  // m==product(i=0..n-1, p[i]**x[i])
    ph := 1
    for i:=0 to n-1
    {
        k = := x[i]  // exponent
        ph := ph * (p[i]**(k-1)) * (p[i]-1)  // ==ph * euler_phi(p[i]**x[i])
    }
}
```

The multiplicative group (consisting of the invertible elements, or *units*) is denoted by $(\mathbb{Z}/m\mathbb{Z})^*$. The size of the group $(\mathbb{Z}/m\mathbb{Z})^*$ equals the number of units:

$$\left|(\mathbb{Z}/m\mathbb{Z})^*\right| \quad = \quad \varphi(m) \tag{37.6-6}$$

If m factorizes as $m = 2^{e_0} \cdot p_1^{e_1} \cdot \ldots \cdot p_q^{e_q}$ where p_i are pairwise distinct primes then

$$\left|(\mathbb{Z}/m\mathbb{Z})^*\right| \quad = \quad \varphi(2^{e_0}) \cdot \varphi(p_1^{e_1}) \cdot \ldots \cdot \varphi(p_q^{e_q}) \tag{37.6-7}$$

Further, the group $(\mathbb{Z}/m\mathbb{Z})^*$ is isomorphic to the direct product of the multiplicative groups modulo the prime powers:

$$(\mathbb{Z}/m\mathbb{Z})^* \quad \sim \quad (\mathbb{Z}/2^{e_0}\mathbb{Z}) \times (\mathbb{Z}/p_1^{e_1}\mathbb{Z}) \times \cdots \times \left(\mathbb{Z}/p_q^{e_q}\mathbb{Z}\right) \tag{37.6-8}$$

That is, instead of working modulo m we can do all computations modulo all prime powers in parallel. The Chinese remainder theorem (section 37.7 on page 747) tells us how to find the element modulo m given the results modulo the prime powers. The other direction is simply modular reduction.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	[--]
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	[1]
2	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	[4]
3	1	3	9	12	6	3	9	12	6	3	9	12	6	3	9	[--]
4	1	4	1	4	1	4	1	4	1	4	1	4	1	4	1	[2]
5	1	5	10	5	10	5	10	5	10	5	10	5	10	5	10	[--]
6	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	[--]
7	1	7	4	13	1	7	4	13	1	7	4	13	1	7	4	[4]
8	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	[4]
9	1	9	6	9	6	9	6	9	6	9	6	9	6	9	6	[--]
10	1	10	10	10	10	10	10	10	10	10	10	10	10	10	10	[--]
11	1	11	1	11	1	11	1	11	1	11	1	11	1	11	1	[2]
12	1	12	9	3	6	12	9	3	6	12	9	3	6	12	9	[--]
13	1	13	4	7	1	13	4	7	1	13	4	7	1	13	4	[4]
14	1	14	1	14	1	14	1	14	1	14	1	14	1	14	1	[2]

	0	1	2	3	4	
1	1	1	1	1	1	[1]
2	1	2	4	8	1	[4]
4	1	4	1	4	1	[2]
7	1	7	4	13	1	[4]
8	1	8	4	2	1	[4]
11	1	11	1	11	1	[2]
13	1	13	4	7	1	[4]
14	1	14	1	14	1	[2]

Figure 37.6-B: Powers and orders modulo 15 (left). The ring $\mathbb{Z}/15\mathbb{Z}$ is noncyclic: there are $\varphi(15) = 8$ invertible elements but no element generates all of them as the maximal order is $R(15) = 4 < \varphi(15)$. The table of powers for the group of units $(\mathbb{Z}/15\mathbb{Z})^*$ (right) is obtained by dropping all elements that are not invertible.

37.6.1 Cyclic and noncyclic rings

If the maximal order R if equal to $|(\mathbb{Z}/m\mathbb{Z})^*| = \varphi(m)$ then the ring $(\mathbb{Z}/m\mathbb{Z})^*$ is then called a *cyclic ring*, else we call the ring *noncyclic*. The term cyclic reflects that in those rings the powers of any element of maximal order 'cycle through' all elements of $(\mathbb{Z}/m\mathbb{Z})^*$. An element of maximal order in cyclic rings is also called a *generator* as its powers 'generate' all elements. Strictly spoken, the terms cyclic and noncyclic refer to the group of invertible elements (with multiplication as group operation). As the additive group (of all elements, with addition as group operation) is always cyclic (one is always a generator) there is no ambiguity when speaking of a cyclic ring.

Figure 37.6-B shows the powers and orders of the noncyclic ring $\mathbb{Z}/15\mathbb{Z}$ where no element generates all units. The rings $\mathbb{Z}/13\mathbb{Z}$ and $\mathbb{Z}/9\mathbb{Z}$ are cyclic, see figure 37.4-A on page 739, and figure 37.4-B on page 740.

For prime modulus $m = p$ the group $(\mathbb{Z}/m\mathbb{Z})^*$ contains all nonzero elements and any element of maximal order is a generator of the group.

For m a power p^k of an odd prime p the maximal order R in $(\mathbb{Z}/m\mathbb{Z})^*$ is

$$R(p^k) \quad = \quad \varphi(p^k) \tag{37.6-9}$$

For m a power of two a tiny irregularity occurs:

$$R(2^k) \quad = \quad \begin{cases} 1 & \text{for } k = 1 \\ 2 & \text{for } k = 2 \\ 2^{k-2} & \text{for } k \geq 3 \end{cases} \tag{37.6-10}$$

That is, for powers of two greater than 4 the maximal order deviates from $\varphi(2^k) = 2^{k-1}$ by a factor of 2. For the general modulus $m = 2^{k_0} \cdot p_1^{k_1} \cdot \ldots \cdot p_q^{k_q}$ the maximal order is

$$R(m) \quad = \quad \text{lcm}\left(R(2^{k_0}),\, R(p_1^{k_1}),\, \ldots,\, R(p_q^{k_q})\right) \tag{37.6-11}$$

where lcm denotes the least common multiple. The maximal order $R(m)$ of an element in $\mathbb{Z}/m\mathbb{Z}$ can be computed as:

```
function maxorder(m)
{
    {n, p[], k[]} := factorization(m)  // m==product(i=0..n-1,p[i]**k[i])
    R := 1
    for i:=0 to n-1
    {
        t := euler_phi_pp(p[i], k[i])  // ==euler_phi(p[i]**k[i])
        if p[i]==2 AND k[i]>=3 then  t := t / 2
        R := lcm(R, t)
    }
    return R
}
```

Now we can see for which moduli m the ring $(\mathbb{Z}/m\mathbb{Z})^*$ will be cyclic (the maximal order equals the number of units $\varphi(m)$):

$$(\mathbb{Z}/m\mathbb{Z})^* \quad \text{is cyclic for} \quad m = 2,\ 4,\ p^k,\ 2 \cdot p^k \qquad \text{where } p \text{ is an odd prime} \tag{37.6-12}$$

If the factorization of m contains two different odd primes p_a and p_b then

$$R(m) \quad = \quad \text{lcm}(\dots, \varphi(p_a), \dots, \varphi(p_b), \dots)$$

is at least by a factor of two smaller than

$$\varphi(m) \quad = \quad \dots \cdot \varphi(p_a) \cdot \dots \cdot \varphi(p_b) \cdot \dots$$

because both $\varphi(p_a)$ and $\varphi(p_b)$ are even. Thereby $(\mathbb{Z}/m\mathbb{Z})^*$ cannot be cyclic in that case. The same argument holds for $m = 2^{k_0} \cdot p^k$ if $k_0 > 1$. For $m = 2^k$ the ring $(\mathbb{Z}/m\mathbb{Z})^*$ is cyclic only for $k = 1$ and $k = 2$ because of the mentioned irregularity of powers of two (relation 37.6-10 on the preceding page).

Pseudo code for a function that returns the order of a given element x in $(\mathbb{Z}/m\mathbb{Z})^*$:

```
function order(x, m)
{
    if gcd(x,m)!=1 then  return 0  // x not a unit
    h := euler_phi(m)  // number of elements of ring of units
    e := h
    {n, p[], k[]} := factorization(h)  // h==product(i=0..n-1,p[i]**k[i])
    for i:=0 to n-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := x**e mod m
        while g1!=1
        {
            g1 := g1**p[i] mod m
            e := e * p[i]
        }
    }
    return e
}
```

Pseudo code for a function that returns an element x in $(\mathbb{Z}/m\mathbb{Z})^*$ of maximal order:

```
function maxorder_element(m)
{
    R := maxorder(m)
    for x:=1 to m-1
    {
        if order(x, m)==R then  return x
    }
    // never reached
}
```

Again, while the function does a simple search it is efficient in practice. For prime m the function returns a primitive root. A C++ implementation is [FXT: `maxorder_element_mod()` in mod/maxorder.cc]. Note that for noncyclic rings the returned element does not necessarily have maximal order modulo all factors of the modulus. We list all elements of $(\mathbb{Z}/15\mathbb{Z})^*$ together with their orders modulo 15, 3, and 5:

```
 1:  r=1  r3=1  r5=1
 2:  r=4  r3=2  r5=4
 4:  r=2  r3=1  r5=2
 7:  r=4  r3=1  r5=4  <--=
 8:  r=4  r3=2  r5=4
11:  r=2  r3=2  r5=1
13:  r=4  r3=1  r5=4  <--=
14:  r=2  r3=2  r5=2
```

The two elements marked with an arrow have maximal order modulo 15 but not modulo 3. An element of maximal order modulo all factors of a composite modulus (equivalently, maximal order in all subgroups) can be found by computing a generator for all cyclic subgroups and applying the Chinese remainder algorithm given in section 37.7.

37.6.2 Generators in cyclic rings

Let G be the set of all generators in a cyclic ring modulo n. Then the number of generators is given by

$$|G| \quad = \quad \varphi(\varphi(n)) \tag{37.6-13}$$

Let g be a generator, the g^k is a generator exactly if $\gcd(k, \varphi(n)) = 1$ as there are $\varphi(\varphi(n))$ numbers k that are coprime to $\varphi(n)$.

Let g be a generator modulo a prime p. Then g is a generator modulo $2\,p^k$ for all $k \geq 1$ if g is odd. If g is even then $g + p^k$ is a generator modulo $2\,p^k$.

Further, g is a generator modulo p^k if $g^{p-1} \bmod p^2 \neq 1$. The only primes below $2^{36} \approx 68 \cdot 10^9$ for which the smallest primitive root is not the a generator modulo p^2 are 2, 40487 and 6692367337. Such primes are called *non-generous primes*, see entry A055578 of [214].

The only known primes p below $32 \cdot 10^{12}$ where $2^{p-1} = 1$ modulo p^2 are 1093 and 3511 (such primes are called *Wieferich primes*, see entry A001220 of [214]). As 2 is not a generator modulo either of the two we see that whenever 2 is a generator modulo $p < 32 \cdot 10^{12}$ then it is also a generator modulo p^k.

37.6.3 Generators in noncyclic rings

When the ring is cyclic an element of maximal order generates all invertible elements. With noncyclic rings one needs more than one generator. Pari/gp's function `znstar()` gives the complete information about the multiplicative group of units. The help text reads:

```
znstar(n): 3-component vector v, giving the structure of (Z/nZ)^*.
 v[1] is the order (i.e. eulerphi(n)),
 v[2] is a vector of cyclic components, and
 v[3] is a vector giving the corresponding  generators.
```

Its output for $2 \leq n \leq 25$ is shown in figure 37.6-C.

The ring is cyclic when there is just one generator. In general, when `znstar(n)` returns

$$[\varphi,\ [r_1, r_2, \ldots, r_k],\ [g_1, g_2, \ldots, g_k]] \tag{37.6-14}$$

then the φ invertible elements u are of the form

$$u \quad = \quad g_1^{e_1}\, g_2^{e_2}\, \ldots\, g_k^{e_k} \tag{37.6-15}$$

where $0 \leq e_i < r_i$ for $1 \leq i \leq k$. For example, with $n = 15$:

```
? for(n=2,25,print(n," ",znstar(n)))
     2 [1,  [],   []]    /* read:  [1, [1], [Mod(1,2)]] */
     3 [2,  [2],  [Mod(2, 3)]]
     4 [2,  [2],  [Mod(3, 4)]]
     5 [4,  [4],  [Mod(2, 5)]]
     6 [2,  [2],  [Mod(5, 6)]]
     7 [6,  [6],  [Mod(3, 7)]]
     8 [4,  [2, 2],  [Mod(5, 8), Mod(3, 8)]]
     9 [6,  [6],  [Mod(2, 9)]]
    10 [4,  [4],  [Mod(7, 10)]]
    11 [10, [10],  [Mod(2, 11)]]
    12 [4,  [2, 2],  [Mod(7, 12), Mod(5, 12)]]
    13 [12, [12],  [Mod(2, 13)]]
    14 [6,  [6],  [Mod(3, 14)]]
    15 [8,  [4, 2],  [Mod(8, 15), Mod(11, 15)]]
    16 [8,  [4, 2],  [Mod(5, 16), Mod(7, 16)]]
    17 [16, [16],  [Mod(3, 17)]]
    18 [6,  [6],  [Mod(11, 18)]]
    19 [18, [18],  [Mod(2, 19)]]
    20 [8,  [4, 2],  [Mod(3, 20), Mod(11, 20)]]
    21 [12, [6, 2],  [Mod(5, 21), Mod(8, 21)]]
    22 [10, [10],  [Mod(13, 22)]]
    23 [22, [22],  [Mod(5, 23)]]
    24 [8,  [2, 2, 2],  [Mod(13, 24), Mod(19, 24), Mod(17, 24)]]
    25 [20, [20],  [Mod(2, 25)]]
```

Figure 37.6-C: Structure of the multiplicative group modulo n for $2 \leq n \leq 25$, as reported by pari/gp's function `znstar()`.

```
? znstar(15)
  [8, [4, 2], [Mod(8, 15), Mod(11, 15)]]
? g1=Mod(8, 15); g2=Mod(11,15);
? for(e1=0,4-1,for(e2=0,2-1,print(e1," ",e2,"  ",g1^e1*g2^e2)))
  0 0  Mod(1, 15)
  0 1  Mod(11, 15)
  1 0  Mod(8, 15)
  1 1  Mod(13, 15)
  2 0  Mod(4, 15)
  2 1  Mod(14, 15)
  3 0  Mod(2, 15)
  3 1  Mod(7, 15)
```

Generators modulo $n = 2^k$

The ring modulo $n = 2^k$ is cyclic only for $k \leq 2$:

```
? for(i=1,6,print(i,": ",znstar(2^i)))
   1: [1, [], []]
   2: [2, [2], [Mod(3, 4)]]
   3: [4, [2, 2], [Mod(5, 8), Mod(3, 8)]]
   4: [8, [4, 2], [Mod(5, 16), Mod(7, 16)]]
   5: [16, [8, 2], [Mod(5, 32), Mod(15, 32)]]
   6: [32, [16, 2], [Mod(5, 64), Mod(31, 64)]]
```

For $k \geq 3$ the multiplicative group is generated by the two elements 5 and -1.

37.6.4 Inversion by exponentiation

For a unit u of order $r = \mathrm{ord}(u)$ one has $u^r = 1$. As r divides the maximal order R also $u^R = 1$ holds and thereby $u^{R-1} \cdot u = 1$. That is, the inverse of any invertible element u equals u to the $(R-1)$-st power:

$$u^{-1} \quad = \quad u^{R-1} \tag{37.6-16}$$

In fact, one has also $u^{-1} = u^{\varphi(m)-1}$ which may involve slightly more work if the ring is noncyclic.

37.7 The Chinese Remainder Theorem (CRT)

Let m_1, m_2, ..., m_f be pairwise coprime (that is, $\gcd(m_i, m_j) = 1$ for all $i \neq j$). If $x \equiv x_i \pmod{m_i}$ for $i = 1, 2, \ldots, f$ then x is unique modulo the product $M = m_1 \cdot m_2 \cdots m_f$. This is the *Chinese remainder theorem* (CRT). Note that it is not assumed that any of the m_i is prime.

The theorem tells us that a computation modulo a composite number M can be split into separate computations modulo the coprime factors of M. To evaluate a function $y := f(x) \bmod M$ where $M = m_1 \cdot m_2$ (with $\gcd(m_1, m_2) = 1$), proceed as follows

1. Splitting: compute $x_1 = x \bmod m_1$ and $x_2 = x \bmod m_2$.
2. Separate computations: compute $y_1 := f(x_1) \bmod m_1$ and $y_2 := f(x_2) \bmod m_2$
3. Recombination: compute y from y_1 and y_2 using the CRT

For example, when computing the exact convolution of a long sequence via number theoretic transforms (see section 25.3 on page 514) the largest term of the result must be smaller than the modulus. Assume that (efficient) modular arithmetic is available for moduli of at most word size. Now choose several coprime moduli whose product M is greater than the largest element of the result, compute the transforms separately and only at the very end compute the elements modulo M.

Efficient computation

For two moduli m_1, m_2 compute x with $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$ as suggested by the following pseudo code:

```
function crt2(x1, m1, x2, m2)
{
    c := m1**(-1) mod m2    // inverse of m1 modulo m2
    s := ((x2-x1)*c) mod m2
    return  x1 + s * m1
}
```

For repeated CRT calculations with the same moduli one will use precomputed values $c = m_1^{-1} \bmod m_2$. With more than two moduli use the above algorithm repeatedly. Pseudo code to perform the CRT for several moduli:

```
function crt(x[0,...,f-1], m[0,...,f-1], f)
{
    x1 := x[0]
    m1 := m[0]
    i := 1
    do
    {
        x2 := x[i]
        m2 := m[i]
        x1 := crt2(x1, m1, x2, m2)
        m1 := m1 * m2
        i := i + 1
    }
    while i < f
    return x1
}
```

A C++ implementation is given in [FXT: mod/chinese.cc]:

```
umod_t
chinese(const umod_t *x, const factorization &f)
// Return R modulo M where:
//   f[] is the factorization of M,
//   x[] := R modulo the prime powers of f[].
{
    const int n = f.nprimes();
    // (omitted test that gcd(m_0,...,m_{n-1})=1 )
    const umod_t M = f.product();
    umod_t R = 0;
    for (int i=0; i<n; ++i)
    {
```

```
        // Ti = prod(mk)  (where k!=i);  Ti==M/mi:
        const umod_t Ti = M / f.primepow(i);  // exact division
        // ci = 1 / Ti:
        umod_t ci = inv_modpp(Ti, f.prime(i), f.exponent(i));
        // here:  0 <= ci < mi
        // Xi = x[i] * ci * Ti:
        umod_t Xi = ci * Ti;  // 0 <= Xi < M
        Xi = mul_mod(Xi, x[i], M);
        // add Xi to result:
        R = add_mod(R, Xi, M);
    }
    return R;
}
```

The underlying construction

We derive the algorithm for CRT recombination from a construction for k coprime moduli. Define T_i as

$$T_i \;:=\; \prod_{k\neq i} m_k \tag{37.7-1}$$

and c_i as

$$c_i \;:=\; T_i^{-1} \mod m_i \tag{37.7-2}$$

Then for X_i defined as

$$X_i \;:=\; x_i\, c_i\, T_i \tag{37.7-3}$$

one has

$$X_i \mod m_j \;=\; \begin{cases} x_i & \text{for} \quad j=i \\ 0 & \text{else} \end{cases} \tag{37.7-4}$$

Therefore

$$x \;:=\; \sum_k X_k \;=\; x_i \mod m_i \tag{37.7-5}$$

For the special case of two moduli m_1, m_2 one has

$$T_1 \;=\; m_2, \qquad T_2 \;=\; m_1 \tag{37.7-6a}$$

$$c_1 \;=\; m_2^{-1} \mod m_1, \qquad c_2 \;=\; m_1^{-1} \mod m_2 \tag{37.7-6b}$$

The quantities are related by

$$c_1\, m_2 + c_2\, m_1 \;=\; 1 \tag{37.7-7}$$

and

$$x \;=\; \sum_k X_k \;=\; x_1\, c_1\, T_1 + x_2\, c_2\, T_2 \tag{37.7-8a}$$

$$\;=\; x_1\, c_1\, m_2 + x_2\, c_2\, m_1 \tag{37.7-8b}$$

$$\;=\; x_1\,(1 - c_2\, m_1) + x_2\, c_2\, m_1 \tag{37.7-8c}$$

$$\;=\; x_1 + (x_2 - x_1)\,(m_1^{-1} \mod m_2)\, m_1 \tag{37.7-8d}$$

The last equality is used in the code.

37.8 Quadratic residues

Let p be a prime. The *quadratic residues modulo p* are those values a so that the equation

$$x^2 \equiv a \pmod{p} \tag{37.8-1}$$

has a solution. If the equation has no solution then a is called a *quadratic non-residue modulo p*. A quadratic residue is a square (modulo p) of some number, so we can safely just call it a *square modulo p*.

Let g be a primitive root (the particular choice does not matter), then every nonzero element x can uniquely be written as $x = g^e$ where $0 < e < p$. Rewriting equation 37.8-1 as $x^2 = (g^e)^2 = g^{2e} = a$ it is apparent that the quadratic residues are the even powers of g. The non-residues are the odd powers of g. All generators are non-residues.

Let us compute $f(x) := x^{(p-1)/2}$ for both residues and non-residues: With a quadratic residue g^{2e} we get $f(g^{2e}) = g^{2e(p-1)/2} = 1^e = 1$ where we used $g^{p-1} = 1$. With a non-residue $a = g^k$, k odd, we get $f(a) = f(g^k) = g^{k(p-1)/2} = -1$ where we used $g^{(p-1)/2} = -1$ (the only square root of 1 apart from 1 is -1) and $-1^k = -1$ for k odd.

Apparently we just found a function that can tell residues from non-residues. In fact, we rediscovered the so-called *Legendre symbol* usually written as $\left(\frac{a}{p}\right)$. A surprising result (proved by Gauss) about the Legendre symbol is the *law of quadratic reciprocity*: let p and q be distinct odd primes, then

$$\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\,\frac{q-1}{2}} \left(\frac{q}{p}\right) \tag{37.8-2}$$

Moreover,

$$\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}} = \begin{cases} +1 & \text{if} \quad p \equiv 1 \pmod 4 \\ -1 & \text{if} \quad p \equiv 3 \pmod 4 \end{cases} \tag{37.8-3}$$

and

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} +1 & \text{if} \quad p \equiv \pm 1 \pmod 8 \\ -1 & \text{if} \quad p \equiv \pm 3 \pmod 8 \end{cases} \tag{37.8-4}$$

Further,

$$\left(\frac{3}{p}\right) = 1 \quad \Longleftrightarrow \quad p \equiv \pm 1 \bmod 12 \tag{37.8-5}$$

and

$$\left(\frac{-3}{p}\right) = 1 \quad \Longleftrightarrow \quad p = 2,\ p = 3,\ \text{or}\ p \equiv 1 \bmod 3 \tag{37.8-6}$$

If a is a quadratic residue modulo p then the polynomial $x^2 - a$ factors as $(x - r_1)(x - r_2)$ where $r_1^2 \equiv a$ and $r_2^2 \equiv a$. Modulo $41 = 4 \cdot 10 + 1$ minus one is a quadratic residue and we have $x^2 + 1 = (x - 9)(x - 32)$. The polynomial $x^2 + 1$ with coefficients modulo $43 = 4 \cdot 10 + 3$ is irreducible.

The relation between the Legendre symbols of positive and negative arguments is

$$\left(\frac{-a}{p}\right) = (-1)^{\frac{p-1}{2}} \left(\frac{a}{p}\right) = \begin{cases} +\left(\frac{a}{p}\right) & \text{if} \quad p = 4k + 1 \\ -\left(\frac{a}{p}\right) & \text{if} \quad p = 4k + 3 \end{cases} \tag{37.8-7}$$

Modulo a prime $p = 4\,k + 3$, if $+a$ is a square then $-a$ is not a square. The orders of any two elements $+a$ and $-a$ differ by a factor of two. Non-residues can easily be found: the number $-(b^2)$ is a non-residue for all b.

Modulo a prime $p = 4\,k + 1$, if $+a$ is a square then $-a$ is also a square. The orders of two non-residues $+a$ and $-a$ are identical. The orders of two residues $+a$ and $-a$ can be identical or differ by a factor of two.

A special case are primes of the form $p = 2^x + 1$, the *Fermat primes*. Only five Fermat primes are known today: $2^1 + 1 = 3$, $2^2 + 1 = 5$, $2^4 + 1 = 17$, $2^8 + 1 = 257$ and $2^{16} + 1 = 65537$. To be prime the exponent x must actually be a power of two. The primitive roots are exactly the non-residues: the maximal order equals $R = \varphi(p) = 2^x$. There are $\varphi(\varphi(p)) = 2^{x-1}$ primitive roots. There are $(p-1)/2 = 2^{x-1}$ squares which all have order at most $R/2$. Remain 2^{x-1} non-residues which must all be primitive roots.

b\a	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
0:	0	+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1:	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
2:	0	+	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0	+	0	−	0
3:	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0	+	−	0
4:	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0
5:	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+	−	−	+	0	+
6:	0	+	0	0	0	+	0	+	0	0	0	+	0	−	0	0	0	−	0	+	0	0	0	+	0	+	0	0	0	+	0	+	0
7:	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+	+	−	+	−	−	0	+	−	+	−
8:	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	+	0	+	0	−	0	−	0	−	0
9:	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	+	0	+	0
10:	0	+	0	+	0	0	0	−	0	+	0	−	0	+	0	0	0	−	0	−	0	−	0	0	0	+	0	−	0	+	0	−	0	0	0
11:	0	+	−	+	+	+	−	−	−	+	−	0	+	−	+	+	+	−	−	−	+	−	0	+	−	+	+	+	−	−	−	+	−	0	+	−	+
12:	0	+	0	0	0	−	0	+	0	0	0	0	−	0	+	0	0	0	−	0	+	0	0	0	−	0	+	0	0	0	−	0
13:	0	+	−	+	+	−	−	−	−	+	+	−	+	0	+	−	+	+	−	−	−	−	+	+	−	+	+	+	−	−	−	−	+	+
14:	0	+	0	+	0	+	0	0	0	+	0	−	0	+	0	+	0	−	0	+	0	0	0	+	0	+	0	+	0	−	0	−	0	−	0	0	0
15:	0	+	+	0	+	0	0	−	+	0	0	−	0	−	−	0	+	+	0	+	0	0	−	+	0	0	−	0	−	−	0	+	+	0	+	0	0
16:	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0	+	0
17:	0	+	+	−	+	−	−	−	+	+	−	−	−	+	−	+	+	0	+	+	−	+	−	−	−	+	+	−	−	−	+	−	+	+	0	+	+
18:	0	+	0	0	0	−	0	+	0	0	0	−	0	−	0	0	0	+	0	−	0	0	0	+	0	+	0	0	0	−	0	+	0	0	0	−	0
19:	0	+	−	−	+	+	+	−	−	−	−	+	+	−	0	+	−	−	+	+	+	+	−	+	−	+	−	−	−	−	+	+
20:	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0	−	0	+	0	+	0	−	0	0	0

Figure 37.8-A: Kronecker symbols $\left(\frac{a}{b}\right)$ for small positive a and b.

We will not pursue the issue, but it should be noted that there are more efficient ways than powering to compute f and the concept of quadratic residues can be carried over to composite moduli: the so-called *Kronecker symbol* generalizes the Legendre symbol. An efficient implementation for its computation (following [83, p.29]) is given in [FXT: `kronecker()` in mod/kronecker.cc]:

```
int
kronecker(umod_t a, umod_t b)
// Return Kronecker symbol (a/b).
// Equal to Legendre symbol (a/b) if b is an odd prime.
{
    static const int  tab2[] = {0, 1, 0, -1, 0, -1, 0, 1};
    // tab2[ a & 7 ] := (-1)^((a^2-1)/8)

    if ( 0==b )  return (1==a);
    if ( 0==((a|b)&1) )  return 0;  // a and b both even ?

    int v = 0;
    while ( 0==(b&1) )  { ++v;  b>>=1; }

    int k;
    if ( 0==(v&1) )  k = 1;
    else             k = tab2[ a & 7 ];

    while ( 1 )
    {
        if ( 0==a )  return ( b>1 ? 0 : k );

        v = 0;
        while ( 0==(a&1) )  { ++v;  a>>=1; }

        if ( 1==(v&1) )  k *= tab2[ b & 7 ];  // k *= (-1)**((b*b-1)/8)

        if ( a & b & 2 )  k = -k;  // k = k*(-1)**((a-1)*(b-1)/4)

        umod_t r = a; // signed:  r = abs(a)
        a = b % r;
        b = r;
    }
}
```

A table of Kronecker symbols $\left(\frac{a}{b}\right)$ for small a and b is shown in figure 37.8-A. It was created with the

program [FXT: mod/kronecker-demo.cc].

Whether a given number a is a square modulo 2^x can be determined via the simple routine [FXT: `is_quadratic_residue_2ex()` in mod/quadresidue.cc]:

```
bool is_quadratic_residue_2ex(umod_t a, ulong x)
// Return whether a is quadratic residue mod 2**x
{
    if ( x==1 )  return true;
    if ( (x>=3 ) && (1==(a&7)) )  return true;
    if ( (x==2 ) && (1==(a&3)) )  return true;
    return false;
}
```

A curious observation regarding quadratic residues is that exactly for the 29 moduli

```
2, 3, 4, 5, 8, 12, 15, 16, 24, 28, 40, 48, 56, 60, 72, 88, 112, 120,
168, 232, 240, 280, 312, 408, 520, 760, 840, 1320, 1848
```

all quadratic residues are non-prime. This sequence is entry A065428 of [214]. It can be generated using the program [FXT: mod/mod-residues-demo.cc].

See any textbook on number theory for the details of the theory of quadratic residues, [160] and [83] for the corresponding algorithms. A method for watermarking that uses quadratic residues is discussed in [21].

37.9 Computation of a square root modulo m

37.9.1 Prime modulus

The square root of a square a modulo a prime $p = 4k + 3$ can be computed as

$$\sqrt{a} \quad = \quad \pm a^{(p+1)/4} \tag{37.9-1}$$

Write $(a^{(p+1)/4})^2 = a^{(p+1)/2} = a^{(p-1)/2+1} = \pm 1 \cdot a = \pm a$, if a is not a square then the square root of $-a$ is obtained. Similar expressions for square root modulo p are developed in [3]. An algorithm for the computation of a square root modulo a prime p (without restriction on the form of p) is given in [83, p.32]. We just give a C++ implementation [FXT: `sqrt_modp()` in mod/sqrtmod.cc]:

```
umod_t
sqrt_modp(umod_t a, umod_t p)
// Return x so that x*x==a (mod p)
// p must be an odd prime.
// If a is not a quadratic residue mod p then return 0.
{
    if ( 1!=kronecker(a,p) )  return 0;  // not a quadratic residue

    // initialize q,t so that  p == q * 2^t + 1
    umod_t q;  int t;
    n2qt(p, q, t);

    umod_t z = 0,  n = 0;
    for (n=1; n<p; ++n)
    {
        if ( -1==kronecker(n, p) )
        {
            z = pow_mod(n, q, p);
            break;
        }
    }

    if ( n>=p )  return 0;

    umod_t y = z;
    uint r = t;
    umod_t x = pow_mod(a, (q-1)/2, p);
    umod_t b = x;
    x = mul_mod(x, a, p);
```

```
    b = mul_mod(b, x, p);

    while ( 1 )
    {
        if ( 1==b )  return x;

        uint m;
        for (m=1; m<r; ++m)
        {
            if ( 1==pow_mod(b, 1ULL<<m, p) )  break;
        }

        if ( m==r )  return  0;  // a is not a quadratic residue mod p

        umod_t v = pow_mod(y, 1ULL<<(r-m-1), p);
        y = mul_mod(v, v, p);
        r = m;
        x = mul_mod(x, v, p);
        b = mul_mod(b, y, p);
    }
}
```

37.9.2 Prime power modulus

For the computation of a square root modulo a prime power p^x the Newton iteration can be used (see section 28.1.5 on page 543). The case $p = 2$ has to be treated separately [FXT: `sqrt_modpp()` in mod/sqrtmod.cc]:

```
umod_t
sqrt_modpp(umod_t a, umod_t p, long ex)
// Return r so that r^2 == a (mod p^ex)
// return 0 if there is no such r
{
    umod_t r;

    if ( 2==p )  // case p==2
    {
        if ( false==is_quadratic_residue_2ex(a, ex) )  return 0;  // no sqrt exists
        else  r = 1;  // (1/r)^2  = a mod 2
    }
    else        // case p odd
    {
        umod_t z = a % p;
        r = sqrt_modp(z, p);
        if ( r==0 )  return  0;  // no sqrt exists
    }
    // here r^2 == a (mod p)

    if ( 1==ex )  return  r;

    umod_t m = ipow(p, ex);
    if ( 2==p )  // case p==2
    {
        long x = 1;
        while ( x<ex )  // Newton iteration for inverse sqrt, 2-adic case
        {
            umod_t z = a;
            z = mul_mod(z, r, m);  // a*r
            z = mul_mod(z, r, m);  // a*r*r
            z = sub_mod(3, z, m);  // 3 - a*r*r
            r = mul_mod(r, z/2, m);  // r*(3 - a*r*r)/2 = r*(1 + (1-a*r*r)/2)
            x *= 2;  //  (1/r)^2 == a mod 2^x
        }
        r = mul_mod(r, a, m);
    }
    else   // case p odd
    {
        umod_t h = inv_modpp(2, p, ex);  // 1/2
        long x = 1;
        while ( x<ex )  // Newton iteration
        {
            umod_t ri = inv_modpp(r, p, ex);    // 1/r
            umod_t ar = mul_mod(a, ri, m);      // a/r
            r = add_mod(r, ar, m);              // r+a/r
            r = mul_mod(r, h, m);               // (r+a/r)/2
```

```
            x *= 2;  //  r^2 == a mod p^x
        }
    }
    return  r;
}
```

37.9.3 Arbitrary modulus

The square root modulo an arbitrary number can be computed from the square roots of its prime power factors using the Chinese remainder theorem (see section 37.7 on page 747) [FXT: `sqrt_modf()` in mod/sqrtmod.cc]:

```
umod_t
sqrt_modf(umod_t a, const factorization &mf)
// Return sqrt(a) mod m, given the factorization mf of m
{
    ALLOCA(umod_t, x, mf.nprimes() );
    for (int i=0; i<mf.nprimes(); ++i)
    {
        // x[i]=sqrt(a) modulo i-th prime power:
        x[i] = sqrt_modpp( a, mf.prime(i), mf.exponent(i) );
        if ( x[i]==0 )  return 0;  // no sqrt exists
    }
    return  chinese(x, mf);  // combine via CRT
}
```

37.10 The Rabin-Miller test for compositeness

We describe a probabilistic method to prove compositeness of an integer.

37.10.1 Pseudoprimes and strong pseudoprimes

For a prime p the maximal order of an element equals $p-1$. That is, for all $a \neq 0$

$$a^{p-1} \equiv 1 \mod p \tag{37.10-1}$$

If for a given number n one succeeds to find an $a > 1$ so that $a^{n-1} \neq 1 \bmod p$ the compositeness of n has been proven. Composite numbers n for which $a^{n-1} = 1 \bmod n$ are called *pseudoprime to base a* (or a-pseudoprime). For example, for $n = 15$ one finds

a :	2	3	4	5	6	7	8	9	10	11	12	13	14
a^{14}:	4	9	1	10	6	4	4	6	10	1	9	4	1

We found that 15 is pseudoprime to the bases 4, 11 and 14 which we also could have read off the rightmost column of figure 37.6-B on page 743.

The bad news is that some composite numbers are pseudoprime to very many bases. The smallest such example is number 561 which is pseudoprime to all bases a with $\gcd(a, n) = 1$. Numbers with that property are called *Carmichael numbers*. The first few are 561, 1105, 1729, 2465, 2821, 6601, 8911, ..., this is sequence A002997 of [214]. There are infinitely many Carmichael numbers as proved in [7]. Finding a base that proves a Carmichael number composite is as difficult as finding a factor.

A significantly better algorithm can be found by a rather simple variation. We use the numbers q and t so that $n - 1 =: q \cdot 2^t$ and examine the values $b := a^q$, b^2, b^4, ..., $b^{2^{t-1}} = a^{(n-1)/2}$. We say that n is a *strong pseudoprime to base a* if either $b \equiv 1$ or $b^{2^e} \equiv -1 \equiv n-1$ for some e where $0 \leq e < t$. We abbreviate strong pseudoprime as SPP. If neither of the conditions holds then n is proven composite. Then n is either not a pseudoprime to base a or we found a square root of 1 that is not equal to $n-1$.

With two different square roots s_1, s_2 modulo n of some number z (here $z = 1$) one has

$$s_1^2 - z \equiv 0 \mod n \tag{37.10-2a}$$
$$s_2^2 - z \equiv 0 \mod n \tag{37.10-2b}$$
$$s_1^2 - s_2^2 = (s_1 + s_2)\,(s_1 - s_2) \equiv 0 \mod n \tag{37.10-2c}$$

So both $s_1 + s_2$ and $s_1 - s_2$ are nontrivial factors of n if $s_1 \neq n - s_2$. Thereby a square root $s \neq -1$ of 1 proves compositeness because both $\gcd(s+1, n)$ and $\gcd(s-1, n)$ are nontrivial factors of n.

Let $B = \left[b, b^2, b^4, \ldots, b^{2^t}\right]$, then for n prime the sequence B must have one of the following forms: either

$$B = [1, 1, 1, \ldots, 1] \quad \text{or} \tag{37.10-3a}$$
$$B = [*, \ldots, *, -1, 1, \ldots, 1] \tag{37.10-3b}$$

where an asterisk denotes any number not equal to $\pm 1 \bmod n$ (notation as in [160]). For n composite the sequence B can also be of the form

$$B = [*, \ldots, *] \quad (a^{n-1} \neq 1) \quad \text{or} \tag{37.10-4a}$$
$$B = [*, \ldots, *, 1, \ldots, 1] \quad \text{(found square root of 1 not equal to } -1) \tag{37.10-4b}$$

If one of the latter two forms is encountered then n must be composite.

With our example $n = 15$ we have $n - 1 = 7 \cdot 2^1$, thereby $q = 7$ and $t = 1$. We only have to examine the value of b. Values of a for which b is not equal to either $+1$ or -1 prove the compositeness of 15.

a:	2	3	4	5	6	7	8	9	10	11	12	13	14
b:	8	12	4	5	6	13	2	9	10	11	3	7	-1

In our example all bases $\neq 14$ prove 15 composite. As n is always a SPP to base $a = n - 1 \equiv -1$ we restrict our attention to values $2 \leq a \leq n - 2$.

A pari/gp implementation of the test whether n is a SPP to base a:

```
sppq(n, a)=
{ /* Return whether n is a strong pseudoprime to base a */
    local(q, t, b, e);
    q = n-1;
    t = 0;
    while ( 0==bitand(q,1), q/=2; t+=1 );
    /* here  n==2^t*q+1 */
    b = Mod(a, n)^q;
    if ( 1==b, return(1) );
    e = 1;
    while ( e<t,
        if( (b==1) || (b==n-1), break(); );
        b *= b;
        e++;
    );
    return( if ( b!=(n-1), 0, 1 ) );
}
```

The Carmichael number 561 ($561 - 1 = 35 \cdot 2^4$, so $q = 35$ and $t = 4$) is a SPP to only 8 out of the 558 interesting bases, and not a SPP for any $2 \leq a \leq 20$ as shown in figure 37.10-A. Note that with $a = 4$ we found $s = 67$ where $s^2 \equiv 1 \bmod 561$ and thereby the factors $\gcd(67+1, 561) = 17$ and $\gcd(67-1, 561) = 33$ of 561.

37.10.2 The Rabin-Miller test

The *Rabin-Miller test* is an algorithm to prove compositeness of a number n by testing strong pseudo-primality with several bases:

```
    a=2:      b=263  b^2=166  b^4= 67
    a=3:      b= 78  b^2=474  b^4=276
    a=4:      b=166  b^2= 67  b^4=  1          all SPP basis:
    a=5:      b= 23  b^2=529  b^4=463        a=50:      b=560
    a=6:      b=318  b^2=144  b^4=540        a=101:     b=560
    a=7:      b=241  b^2=298  b^4=166        a=103:     b=  1
    a=8:      b=461  b^2=463  b^4= 67        a=256:     b=  1
    a=9:      b=474  b^2=276  b^4=441        a=305:     b=560
    a=10:     b=439  b^2=298  b^4=166        a=458:     b=560
    a=11:     b=209  b^2=484  b^4=319        a=460:     b=  1
    a=12:     b= 45  b^2=342  b^4=276        a=511:     b=  1
    a=13:     b=208  b^2= 67  b^4=  1
    a=14:     b=551  b^2=100  b^4=463
    a=15:     b=111  b^2=540  b^4=441
    a=16:     b= 67  b^2=  1
    a=17:     b=527  b^2= 34  b^4= 34
    a=18:     b=120  b^2=375  b^4=375
    a=19:     b= 76  b^2=166  b^4= 67
    a=20:     b=452  b^2=100  b^4=463
```

Figure 37.10-A: The Carmichael number $561 = 35 \cdot 2^4 + 1$ is a strong pseudoprime to 8 out of 558 bases a (right) and no basis $2 \le a \le 20$ (left).

```
rm(n, na=20)=
{ /* Rabin Miller test */
    local(a);
    for (a=2, na+2,
        if ( a>n-2,  break() );
        if ( 0==sppq(n, a), return(0) ); /* proven composite */
    );
    return(1); /* composite with probability less than 0.25^na */
}
```

It can be shown that for a composite number the probability to be a SPP to a 'random' base is at most 1/4. Thereby the compositeness of number can in practice quickly be proven. While the algorithm does *not* prove primality, it can be used to rule out compositeness with a very high probability.

```
        Bases tested:    2   3  5  6  7 10 11 12 13 14 15 17
              91:  [3]                  10    12          17
             133:  [2]                     11 12
             145:  [2]                        12          17
             276:  [2]                     11    13
             286:  [2]       3                             17
             703:  [2]       3        7
             742:  [2]                                 15 17
             781:  [2]          5                          17
             946:  [2]                7                  15
            1111:  [2]             6                       17
            1729:  [2]                10    12
            2047:  [2]    2              11
            2806:  [2]          5                13
            2821:  [2]                     12             17
            3277:  [3]    2                          14 15
            4033:  [2]    2                             17
            4187:  [2]                10                 17
            5662:  [2]          5                       17
            5713:  [2]             6                14
            6533:  [2]             6    10
            6541:  [2]                                14 15
            7171:  [2]                                14    17
            8401:  [2]       3           10
            8911:  [3]       3                 12 13
            9073:  [2]                         12    14
```

Figure 37.10-B: All numbers $\le 10,000$ that are strong pseudoprimes to more than one base $a \le 17$ (omitting bases that are perfect prime powers).

A list (created with the program [FXT: mod/rabinmiller-demo.cc]) of composites $n \le 10,000$ that are SPP to more than one base $a \le 17$ is shown in figure 37.10-B. The table indicates how effective the Rabin-Miller algorithm actually is: it does not contain a single number pseudoprime to both 2 and 3. The first few odd composite numbers that are SPP to both bases $a = 2$ and $a = 3$ are shown in figure 37.10-C. There are 104 such composite $n < 2^{32}$, given in [FXT: data/pseudo-spp23.txt]. This sequence of numbers

```
    1,373,653  ==  1 + 2^2 * 3^3 * 7 * 23 * 79
               ==  829 * 1657  == (1 + 2^2 *3^2 *23) * (1 + 2^3 *3^2 *23)

    1,530,787  ==  1 + 2 * 3 * 103 * 2477
               ==  619 * 2473  == (1 + 2 *3 *103)    * (1 + 2^3 *3 *103)

    1,987,021  ==  1 + 2^2 * 3^2 * 5 * 7 * 19 * 83
               ==  997 * 1993  == (1 + 2^2 *3 *83)   * (1 + 2^3 *3 *83)

    2,284,453  ==  1 + 2^2 * 3^2 * 23 * 31 * 89
               == 1069 * 2137  == (1 + 2^2 *3 *89)   * (1 + 2^3 *3 *89)

    3,116,107  ==  1 + 2 * 3^2 * 7^2 * 3533
               ==  883 * 3529  == (1 + 2 *3^2 *7^2)  * (1 + 2^3 *3^2 *7^2)

    5,173,601  ==  1 + 2^5 * 5^2 * 29 * 223
               ==  929 * 5569  == (1 + 2^5 *29)      * (1 + 2^6 *3 *29)

    6,787,327  ==  1 + 2 * 3 * 7 * 13 * 31 * 401
               == 1303 * 5209  == (1 + 2 *3 *7 *31)  * (1 + 2^3 *3 *7 *31)

   11,541,307  ==  1 + 2 * 3 * 7 * 283 * 971
               == 1699 * 6793  == (1 + 2 *3 *283)    * (1 + 2^3 *3 *283)

   13,694,761  ==  1 + 2^3 * 3^2 * 5 * 109 * 349
               == 2617 * 5233  == (1 + 2^3 *3 *109)  * (1 + 2^4 *3 *109)
```

Figure 37.10-C: The first composite numbers that are SPP to both bases 2 and 3.

```
      25,326,001  ==  1 + 2^4 * 3^3 * 5^3 * 7 * 67
                  == 2251 * 11251  == (1 + 2 *3^2 *5^3)  * (1 + 2 *3^2 *5^4)

     161,304,001  ==  1 + 2^6 * 3 * 5^3 * 11 * 13 * 47
                  == 7333 * 21997  == (1 + 2^2 *3 *13 *47)  * (1 + 2^2 *3^2 *13 *47)

     960,946,321  ==  1 + 2^4 * 3 * 5 * 29 * 101 * 1367
                  == 11717 * 82013  == (1 + 2^2 *29 *101)  * (1 + 2^2 *7 *29 *101)

   1,157,839,381  ==  1 + 2^2 * 3^3 * 5 * 401 * 5347
                  == 24061 * 48121  == (1 + 2^2 *3 *5 *401)  * (1 + 2^3 *3 *5 *401)

   3,215,031,751  ==  1 + 2 * 3^4 * 5^3 * 7 * 37 * 613
                  == 151 * 751 * 28351
                  == (1 + 2 *3 *5^2)  * (1 + 2 *3 *5^3)  * (1 + 2 *3^4 *5^2 *7)

   3,697,278,427  ==  1 + 2 * 3^3 * 31 * 563 * 3923
                  == 30403 * 121609  == (1 + 2 *3^3 *563)  * (1 + 2^3 *3^3 *563)
```

Figure 37.10-D: The first composite numbers that are SPP to all bases 2, 3 and 5.

is entry A072276 of [214], entry A001262 gives the base-2 SPPs, and entry A020229 the base-3 SPPs. We note the uneven distribution modulo 12:

```
(n%12: num)     (1: 75)     (5: 9)     (7: 18)    (11: 2)
```

Composites that are SPP to the three bases 2, 3 and 5 are quite rare, figure 37.10-D shows all 6 such composite numbers smaller than 2^{32} (values taken from [192] which lists all such numbers $< 25 \cdot 10^9$). Thereby one can speed up the Rabin-Miller test for small values of n (say, $n < 2^{32}$) by only testing the bases $a = 2, 3, 5$ and, if n is a SPP to these bases, look up the composites in the table. The smallest odd composites that are SPP to the first k prime bases up to $k = 8$ are determined in [140], they are given as sequence A006945 of [214].

```
    composite       SPP to base
         2047       2
      1373653       2, 3
     25326001       2, 3, 5
   3215031751       2, 3, 5, 7
2152302898747       2, 3, 5, 7, 11
3474749660383       2, 3, 5, 7, 11, 13
341550071728321     2, 3, 5, 7, 11, 13, 17 [and 19]
341550071728321     2, 3, 5, 7, 11, 13, 17, 19
```

Note that if the probability of a base not proving compositeness was exactly 1/4 we would find *much* more entries in figure 37.10-D. Slightly overestimating the number of composites below N as N, there should be about $(1/4)^3 N = N/64$ entries, that is $2^{26} \approx 6 \cdot 10^7$ for $N = 2^{32}$, but we have only 6 entries. Thereby the Rabin-Miller test is in practice significantly more efficient than one may initially assume. Let $p_{k,t}$ be the probability that a k-bit composite 'survives' t passes of the Rabin-Miller test. Then we

have, as shown in [95],

$$p_{k,1} \quad < \quad k^2\, 4^{2-\sqrt{k}} \qquad \text{for} \quad k > 2 \tag{37.10-5}$$

For large numbers, the probability bound on the left hand side is much smaller than $1/4$: for example, $p_{1000,1} < 2^{-39}$. Other bounds given in the cited paper are

$$p_{100,10} \quad < \quad 2^{-44} \tag{37.10-6a}$$
$$p_{300,5} \quad < \quad 2^{-60} \tag{37.10-6b}$$
$$p_{600,1} \quad < \quad 2^{-75} \tag{37.10-6c}$$

The last bound is stronger than that of relation 37.10-5. Even stronger bounds are given in [70], especially the relation $p_{k,t} < 4^{-t}$ for all $k \geq 2$ and $t \geq 1$.

```
       Bases tested:  2   3  5  6  7 10 11 12 13 14 15 17
              11476:      3  5                       15 17
              88831:               7    11 12        15 17
             188191:      3        7 10                17
             597871:                  10 11 12 13
             736291:               7 10    12 13
             765703:                  10 11 12    14
            1024651:      3  5     7          12       15
            1056331:               7 10       12 13
            1152271:      3        7    11 12 13
            1314631:         5     7    11 12 13
            1373653:   2  3     6              12             17
            1530787:   2  3     6              12
            1627921:      3  5                          14 15
            1857241:         5                 11 12    14
            1987021:   2  3     6              12             17
            2030341:                           11 12    14 15
            2284453:   2  3     6  7           11 12
            2741311:                           11 12    14 15
            3116107:   2  3     6              12
            4181921:   2     5        10           13
            4224533:                6          11       14 15 17  <--=
            5122133:                6  7 10 11
            5173601:   2  3     6              12
            5481451:      3  5                 12          15
            6594901:         5                 12 13 14
            6787327:   2  3     6              12 13
            8086231:         5                 12    14    17
            9504191:                        11 12 13    15
            9863461:   2     5  6     10                      17
```

Figure 37.10-E: Composites $\leq 10^7$ that are SPP to at least four bases.

The composites $\leq 10^7$ that are SPP to four or more bases $a < 17$ are shown in figure 37.10-E. We omit values of a that are perfect powers because if n is a base-a SPP then it is also a base-a^k SPP for all $k > 1$. The entry for $n = 4224533$ (marked with an arrow) shows that a number that is not a SPP to two bases a_1 and a_2 may still be a SPP to the base $a_1 \cdot a_2$ (here $a_1 = 2$, $a_2 = 3$). This indicates that one might want to restrict the tested bases to primes. All odd composite numbers $\leq 10^7$ that are SPP to four or more prime bases $a \leq 17$ are

```
 Bases tested:   2  3  5  7 11 13 17
1152271:  [4]       3     7 11 13
1314631:  [4]          5  7 11 13
2284453:  [4]    2  3     7 11
```

Note that a number that is a SPP to bases a_1 and a_2 is *not* necessarily SPP to the base $a_1 \cdot a_2$. An example is $n = 9,006,401$ which is a SPP to bases 2 and 5 but not to base 10:

```
9006401:   2     4  5        8                       16    18
```

All composites $\leq 10^7$ that are SPP to bases 2 and 3 are also SPP to base 6, same for bases 2 and 5. Out of six composites $\leq 10^7$ that are SPP to bases 2 and 7 three are not SPP to base 14:

```
 314821:   2     4     6  7  8  9                       16    18
2269093:   2     4        7  8              14          16
2284453:   2  3  4     6  7  8  9    11 12              16    18
3539101:   2     4        7  8                 13       16
5489641:   2     4        7  8                    14    16
6386993:   2     4        7  8                    14    16
```

37.10.3 Implementation of the Rabin-Miller test

A C++ implementation of the test for pseudoprimality is given in [FXT: mod/rabinmiller.cc]:

```
bool
is_strong_pseudo_prime(const umod_t n, const umod_t a, const umod_t q, const int t)
// Return whether n is a strong pseudoprime to base a.
// q and t must be set so that  n == q * 2^t + 1
{
    umod_t b = pow_mod(a, q, n);

    if ( 1==b )  return true;  // passed
    // if ( n-1==b )  return true;  // passed

    int e = 1;
    while ( (b!=1) && (b!=(n-1)) && (e<t) )
    {
        b = mul_mod(b, b, n);
        e++;
    }

    if ( b!=(n-1) )  return false;  // =--> composite

    return  true;  // passed
}
```

It uses the routine

```
void
n2qt(const umod_t n, umod_t &q, int &t)
// Set q,t so that  n == q * 2^t + 1
// n must not equal 1, else routine loops.
{
    q = n - 1;  t = 0;
    while ( 0==(q & 1) )  { q >>= 1; ++t; }
}
```

Now the Rabin-Miller test can be implemented as

```
bool
rabin_miller(umod_t n, uint cm/*=0*/)
// Rabin-Miller compositeness test.
// Return true of none of the bases <=cm prove compositeness.
// If false is returned then n is proven composite (also for n=1 or n=0).
// If true is returned the probability
//   that n is composite is less than (1/4)^cm
{
    if ( n<=1 )  return false;
    if ( n < small_prime_limit )  return  is_small_prime( (ulong)n );

    umod_t q;
    int t;
    n2qt(n, q, t);

    if ( 0==cm )  cm = 20;  // default
    uint c = 0;
    while ( ++c<=cm )
    {
        umod_t a = c + 1;

        // if n is a c-SPP then it also is a c**k (k>1) SPP.
        // That is, powers of a non-witness are non-witnesses.
        // So we skip perfect powers:
        if ( is_small_perfpow(a) )  continue;

        if ( a >= n )  return  true;
        if ( !is_strong_pseudo_prime(n, a, q, t) )  return false;  // proven composite
    }

    return true;  // strong pseudoprime for all tested bases
}
```

The function `is_small_perfpow()` [FXT: mod/perfpow.cc] returns `true` is its argument is a (small) perfect power. It uses a lookup in a precomputed bit-array.

A generalization of the Rabin-Miller test applicable when more factors (apart from 2) of $n-1$ are known is given in [40]. Another generalization (named extended quadratic Frobenius primality test) is suggested in [96].

37.11 Proving primality

We describe several methods to prove primality. Only the first, Pratt's certificate of primality, is applicable for numbers of arbitrary form but not practical in general because it relies on the factorization of $n-1$. The Pocklington-Lehmer test only needs a partial factorization of $n-1$. We give further tests applicable only for numbers of a special forms: Pepin's test, the Lucas-Lehmer test, and the Lucas test.

As already said, the Rabin-Miller test can only prove compositeness. Even if a candidate 'survives' many passes, we only know that it is prime with a high probability.

37.11.1 Pratt's certificate of primality

Only with a prime modulus p the maximal order equals $R = p-1$. To determine the order of an element modulo p one needs the factorization of $p-1$. Now if the factorization of $p-1$ is known and we can find a primitive root then we do know that p is prime. Thereby it is quite easy to prove primality for numbers of certain special forms. For example, let $p := 2 \cdot 3^{30} + 1 = 411,782,264,189,299$. One finds that 3 is a primitive root and so we know that p is prime.

```
[314159311, [3], [2, 3, 5, 199, 1949]]
    [2, "--"]
    [3, [2], [2]]
        [2, "--"]
    [5, [2], [2]]
        [2, "--"]
    [199, [3], [2, 3, 11]]
        [2, "--"]
        [3, [2], [2]]
            [2, "--"]
        [11, [2], [2, 5]]
            [2, "--"]
            [5, [2], [2]]
                [2, "--"]
    [1949, [2], [2, 487]]
        [2, "--"]
        [487, [3], [2, 3]]
            [2, "--"]
            [3, [2], [2]]
                [2, "--"]
```

Figure 37.11-A: A certificate for the primality of $p = 314,159,311$.

In general, the factorization of $p-1$ can contain large factors whose primality needs to be proven. Recursion leads to a primality certificate in the form of a tree which is called *Pratt's certificate of primality*.

A certificate for the primality of $p = 314,159,311$ is shown in figure 37.11-A. The first line says that 3 is a primitive root of $p = 314159311$ and $p-1$ has the prime factors $2, 3, 5, 199, 1949$ (actually, $p-1 = 2 \cdot 3^4 \cdot 5 \cdot 199 \cdot 1949$ but we can ignore exponents). In the second level (that is, indented by 4 characters) appear the prime factors just determined together with their primality certificates: the prime 2 is trivially accepted, all other primes are followed by their (further indented) certificates.

The certificate was produced with the following pari/gp code:

```
indprint(x, ind)=
{ /* print x, indented by ind characters */
    for (k=1, ind, print1(" ") );
    print(x);
}

pratt(p, ind=0)=
{
    local( a, p1, f, nf, t );
```

```
    [3141592653589793238462643383279502884197169399375 31, [3], \
        [2, 3, 5, 67, 89, 151, 39829177707048956693, 29200179492960384562 1939]]
        [151, [6], [2, 3, 5]]
        [39829177707048956693, [2], [2, 9957294426762239173]]
            [9957294426762239173, [6], [2, 3, 7, 11, 14153, 385109, 1977139]]
                [14153, [3], [2, 29, 61]]
                [385109, [2], [2, 43, 2239]]
                    [2239, [3], [2, 3, 373]]
                        [373, [2], [2, 3, 31]]
                [1977139, [3], [2, 3, 109841]]
                    [109841, [3], [2, 5, 1373]]
                        [1373, [2], [2, 7]]
        [29200179492960384562 1939, [2], [2, 13, 3157127, 3557296955910619]]
            [3157127, [7], [2, 7, 225509]]
                [225509, [2], [2, 56377]]
                    [56377, [5], [2, 3, 29]]
            [3557296955910619, [3], [2, 3, 47, 673, 6247908971]]
                [673, [5], [2, 3, 7]]
                [6247908971, [2], [2, 5, 624790897]]
                    [624790897, [5], [2, 3, 13016477]]
                        [13016477, [2], [2, 11, 29, 101]]
                            [101, [2], [2, 5]]
```

Figure 37.11-B: A shortened certificate for the primality of first prime greater than $\pi \cdot 10^{50}$. Here all primes smaller than 100 are considered trivially verifiable and not listed.

```
    if ( p<=2,   \\ 2 is trivially prime
        indprint([p, "--"], ind);
        return();
    );
    \\ p-1 is factored here:
    a = component(znprimroot(p),2);
    \\ but we cannot access the factorization, so we do it "manually":
    p1 = p-1;
    f = factor(p1);
    nf = matsize(f)[1];
    t = vector(nf,j, f[j,1] );  f = t;  \\ prime factors only
    indprint([p, [a], t], ind);
    \\ recurse on prime factors of p-1:
    for (k=1, nf, pratt(f[k], ind+4));
    return();
}
p=nextprime(Pi*10^8)
pratt(p)
```

The routine has to be taken with a grain of salt as we rely on the fact that `znprimroot(p)` will fail for composite p:

```
? pratt(1000)
  ***   primitive root does not exist in gener
```

The routine has an additional parameter `ind` determining the indentation used with printing. This parameter is incremented with the recursion level, resulting in the tree-like structure of the output. This little trick is often useful with recursive procedures.

With a precomputed table of small primes (see section 37.3 on page 738) the line

```
    if ( p<=2,   \\ 2 is trivially prime
```

can be changed to something like

```
    if ( (p<=ptable_max) && (ptable[p]==1),   \\ trivial to verify
```

which will shorten the certificate significantly. A certificate for the smallest prime p greater than $\pi \cdot 10^{50}$ and `ptable_max`=100 is shown in figure 37.11-B, the output of 'trivial' primes is suppressed. We note that $p = \lceil \pi \cdot 10^{50} \rceil + 20$.

Once a certificate is computed it can be verified extremely quickly. Note that this type of primality certificate needs the factorization of $p-1$ and thus may not be feasible for large values of p.

37.11.2 The Pocklington-Lehmer test

Let $p-1 = F \cdot U$ where $F > U$ and all prime factors of F are known. If, for each prime factor q of F, we can find a_q so that $a_q^{p-1} \equiv 1 \bmod p$ and $\gcd\left(a_q^{(p-1)/q} - 1, p\right) = 1$, then p is prime.

The following implementation removes entries from the list of prime factors q of F until the list is empty:

```
pocklington_lehmer(F, u, c=10000)=
{ /* Pocklington-Lehmer test for the primality of p=f*u+1.
   * Return last successful base, else zero.
   * F must be the factorization of f.
   * Test bases a=2...c
   * Must have u<f.
   */
    local(n, f, C, p, t, ct);
    n = matsize(F)[1];
    f = prod(j=1, n, F[j,1]^F[j,2]);
    if ( f<=u, return(0) );
    p = f*u + 1;
    C = vector(n, j, (p-1)/F[j,1]);
    ct = n;  \\ number remaining prime divisors of f
    for (a=2, c,
        if ( 1==Mod(a,p)^(p-1),
            for (j=1, n,
                if ( C[j]!=0,  \\ skip entries already removed
                    t = component( Mod(a,p)^C[j], 2);
                    if( 1==gcd(t-1, p),
                        C[j] = 0;  \\ remove entry
                        ct -= 1;   \\ number of remaining entries
                    );
                );
            );
            if ( ct==0, return(a) );
        );
    );
    return( 0 );
}
```

We search all primes of the form $p = F \cdot U + 1$ where $F = 100!$, $U = F - d$, and d lies in the range $1, \ldots, 1000$. Only candidates that are strong pseudoprimes to both bases 2 and 3 are tested:

```
f=100!;
F=factor(f);
{ for (d=1, 1000,
    u = f - d;
    p = f*u+1;
    if ( sppq(p, 2) && sppq(p,3),
        q2 = pocklington_lehmer(F, u);
        print1(d, ":  ");
        print1("  ", q2);
        print();
    );
 ) }
```

We find five such primes $\approx 8.70978248908948 \cdot 10^{315}$ (in about 10 seconds):

```
     d:   last a
    45:    103
   778:    101
   818:    101
   880:    101
   884:    103
```

The returned value a_q is the one that did lead to the removal of the last entry in `C[]`. The value is smaller with less prime factors of F. Using $F = 2^{500}$ we find primes ($\approx 1.07150860718626 \cdot 10^{301}$) of the form $p = F \cdot U + 1$ where $U = F - d$ and $1 \le d \le 3000$ for the following d and maximal a_q:

```
     d:   last a        d:   last a
   214:    5         1383:    3
   294:    3         1801:   13
  1023:    3         2041:   11
```

```
1114:    5          2481:    3
1321:   17
```

The search takes about 20 seconds. Discarding candidates that have small prime factors ($p < 1,000$) gives a four-fold speedup. The prime $2^{3340}\,(2^{3340} - 1633) + 1 \approx 7.59225935 \cdot 10^{2010}$ is found within 5 minutes.

A further refinement of the test is given in [83], see also [67].

37.11.3 Tests for $n = k\,2^t + 1$

37.11.3.1 Proth's theorem and Pepin's test

For numbers of the form $p = q \cdot 2^t + 1$ with q odd and $2^t > q$ primality can be proven as follows: If there is an integer a so that $a^{(p-1)/2} \equiv -1$ then p must be prime. This is *Proth's theorem.* The 'FFT-primes' are natural candidates for Proth's theorem. For example, with $p := 2^{57} \cdot 29 + 1 = 4,179,340,454,199,820,289$ one finds that $a^{(p-1)/2} \equiv -1$ for $a = 3$ so p must be prime.

Note that Proth's theorem is the special case $F = 2^t > k = U$ of the Pocklington-Lehmer test.

Numbers of the form 2^t+1 are composite unless t is a power of two. The candidates are therefore restricted to the so-called *Fermat numbers* $F_n := 2^{2^n} + 1$. Here it suffices to test whether $3^x \equiv -1 \bmod F_n$ where $x = 2^{t-1}$:

```
pepin(tx)=
{
    local(t, F, x);
    t = 2^tx;
    F = 2^t+1;
    x = 2^(t-1);
    return(  (-1==Mod(3,F)^x) );
}
```

This test is known as *Pepin's test.* As shown in section 37.8 on page 750 all non-residues are primitive roots modulo prime F_n. Three is just the smallest non-residue.

```
for (tx=1,12, print(tx,"  ", pepin(tx)))
   1  1  \\ F_1 = 5
   2  1  \\ F_2 = 17
   3  1  \\ F_3 = 257
   4  1  \\ F_4 = 65537
   5  0
   6  0
  12  0
```

No Fermat prime greater than $F_4 = 65537$ is known today and all F_n where $5 \le n \le 32$ are known to be composite.

Note that F_{n+1} has (about) twice as many bits as F_n. Further the number of squarings involved in the test $(t-1)$ is (about) doubled. If we underestimate the cost of multiplying N-bit numbers with N operations we get a lower bound for the ratio of the costs of testing F_{n+1} and F_n of four. Assuming the computer power doubles every 18 month and Pepin's test of F_n is just feasible today we'd have to wait three years (36 month) before we can test F_{n+1}. The computation that proved F_{24} composite is described in [91].

37.11.3.2 What to consider before doing the Pepin's test

As $2^t \equiv -1 \bmod F_n = 2^t + 1$ we see that the order of two equals $2\,t = 2^{n+1}$. The same is true for factors of composite Fermat numbers. When searching factors of F_n one only needs to consider candidates of the form $1 + k\,2^{n+2}$. A routine that searches for small factors of f_n can be implemented as:

```
ord2pow2(p)=
\\ Return the base-2 logarithm of the order of two modulo p
\\ Must have: ord(2)==2^k for some k
{
    local(m, rx);
```

```
    rx = 0;
    m = Mod(2,p);
    while ( m!=1, m*=m; rx++; );
    return( rx );
}

ftrialx(n, mm=10^5, brn=0)=
\\ Try to find small factors of the Fermat number F_n=2^(2^n)+1
\\ Try factors 1+ps, 1+2*ps, ... , 1+mm*ps  where ps=2^(n+2)
\\ Stop if brn factors were found (zero: do not stop)
{
    local(p,ps,ttx,fct);
    ps = 2^(n+2);  \\ factors are of the form 1+k*ps
    p = ps+1;      \\ trial factor
    ttx = 2^(n+1); \\ will test whether Mod(2,p)^ttx==1
    fct = 0;  \\ how many factors were found so far
    for (ct=1, mm,
        if ( (Mod(2,p)^(ttx)==1)  \\ order condition
            && ( (rx=ord2pow2(p)) == n+1 ) \\ avoid factors of smaller Fermat numbers
        , /* then */
            print1(n, ":  ");
            print1(p);
            print1("    p-1=",factor(p-1));
            print();
            fct++;
            if ( fct==brn,  break() );
        );
        p += ps;
    );
    return(fct);
}
```

We create a list of small prime factors of F_n for $5 \le n \le 32$ where the search is restricted to factors $f \le 1 + 10^5\, 2^{n+2}$ and stopped when a factor was found:

```
for(n=5,32, ftrialx(n, 10^5, 1); );
   5:  641    p-1=[2, 7; 5, 1]
   6:  274177    p-1=[2, 8; 3, 2; 7, 1; 17, 1]
   9:  2424833    p-1=[2, 16; 37, 1]
  10:  45592577    p-1=[2, 12; 11131, 1]
  11:  319489    p-1=[2, 13; 3, 1; 13, 1]
  12:  114689    p-1=[2, 14; 7, 1]
  15:  1214251009    p-1=[2, 21; 3, 1; 193, 1]
  16:  825753601    p-1=[2, 19; 3, 2; 5, 2; 7, 1]
  18:  13631489    p-1=[2, 20; 13, 1]
  19:  70525124609    p-1=[2, 21; 33629, 1]
  23:  167772161    p-1=[2, 25; 5, 1]
  32:  25409026523137    p-1=[2, 34; 3, 1; 17, 1; 29, 1]
```

A list for $5 \le n \le 300$ is given in [FXT: data/small-fermat-factors.txt]. Note that an entry

```
 201:  1245698371909569261600129013982869249475211760780421005925626675 21 \
    p-1=[2, 204; 3, 1; 5, 1; 17, 1; 19, 1]
```

asserts the compositeness of the number F_{201} where Pepin's test is out of reach by far. Indeed, its binary representation could not be stored in all existing computer memory combined: F_{201} is a $\log_2(F_{201}) \approx 2^{201} = 3.2138 \cdot 10^{60}$ -bit number.

The status of the (partial) factorizations of Fermat numbers is given in [149].

37.11.4 Tests for $n = k\, 2^t - 1$

37.11.4.1 The Lucas-Lehmer test for Mersenne numbers

Define the sequence H by $H_0 = 1$, $H_1 := 2$ and $H_i = 4\, H_{i-1} - H_{i-2}$. The Mersenne number $n = 2^e - 1$ is prime exactly if $H_{2^{e-2}} \equiv 0 \bmod n$. The first few terms of the sequence H are

```
    k:  0  1  2   3   4   5     6     7     8      9      10      11      12
  H_k:  1  2  7  26  97  362  1351  5042  18817  70226  262087  978122  3650401 ...
```

The numbers H_k can be computed efficiently via the index doubling formula $H_{2k} = 2\,H_k^2 - 1$. Starting with the value $H_1 = 2$ and computing modulo n the implementation is a s simple as

```
LL(e)=
{
    local(n, h);
    n = 2^e-1;
    h = Mod(2,n);
    for (k=1, e-2, h=2*h*h-1);
    return( 0==h );
}
? LL(521)
1  \\  2^521-1 is prime
? LL(239)
0  \\  2^239-1 is composite
? LL(9941)
1 \\ 2^9941-1 is prime
? ##
  ***   last result computed in 4,296 ms.
```

The algorithm is called the *Lucas-Lehmer test*.

Note that most sources use the sequence $V = 2, 4, 14, 52, 194, 724, 2702, \ldots$ that satisfies the same recurrence relation. We have $H_k = \frac{1}{2}\,V_k$ (H is *half* of V). The index doubling relation becomes $V_{2k} = V_k^2 - 2$.

37.11.4.2 What to consider before doing the Lucas-Lehmer test

The exponent e of a Mersenne prime must be prime, else n factors algebraically as

$$2^e - 1 \quad = \quad \prod_{d \backslash e} Y_d(2) \tag{37.11-1}$$

where $Y_k(x)$ is the k-th cyclotomic polynomial. For example, for $2^{21} - 1 = 2097151$:

```
? m=1; fordiv(21,d,y2=subst(polcyclo(d,x),x,2);m*=y2;print(d,": ",y2)); m
  1: 1
  3: 7
  7: 127
  21: 2359
  2097151
```

The factors found are not necessarily prime: here 2359 factors further into $7 \cdot 337$. More information on the multiplicative structure of $b^e \pm 1$ can be found in [68].

Before doing the Lucas-Lehmer test one should do a special version of trial division based on the following observation: any factor f of $m = 2^e - 1$ has the property that $2^e \equiv 1 \bmod f$. That is $2^e - 1 \equiv 0 \bmod f$, so $m \equiv 0 \bmod f$ and f divides m. We further use the fact that factors are of the form $2\,e\,k + 1$ and $f \equiv \pm 1 \bmod 8$. The following routine does not try assert the primality of a candidate factor f as this would render the computation considerably slower.

```
mers_trial(e, mct=10^7, bnf=0)=
\\  try to discover small factors of the Mersenne number 2^e-1
\\  e : exponent of the Mersenne number
\\  mct : how many factors are tried
\\  pfq : stop with the factor found (zero: do not stop)
{
    local(f, fi, ct, fct);
    print("exponent e=",e);
    print("trying up to ", mct, " factors");
    fi=2*e; \\ factors are of the form 2*e*k+1
    f=1;
    ct=0;
    fct=0;  \\ how many factors where found so far
    while (ct < mct,
        f += fi;
        m8 = bitand(f, 7);  \\ factor modulo 8
        if ( (1!=m8) && (7!=m8), next(); );  \\ must equal +1 or -1
        if ( Mod(2, f)^e == Mod(1, f),
            print(f, "  ", isprime(f));  \\ give factor and tell whether it is prime
            fct++;
```

```
            if ( fct==bnf , break(); );
        );
        ct++;
    );
}
```

For $m = 2^{10007} - 1$ (3013 decimal digits) we find three factors of which all are prime:

```
? e=10007; mers_trial(e,,3);
exponent e=10007
trying up to 10000000 factors
  240169  1
  60282169  1
  136255313  1
? ##
  ***   last result computed in 44 ms.
? ceil((e*log(2.0)/log(10.0)))
  3013 \\ m=2^e-1 has 3,013 decimal digits
```

Sometimes one is lucky with truly huge numbers:

```
? e=2^31-1; mers_trial(e,,1);
exponent e=2147483647
trying up to 10000000 factors
  295257526626031  1
? ##
  ***   last result computed in 583 ms.
? ceil((e*log(2.0)/log(10.0)))
  646456993  \\  m=2^e-1 has 646,456,993 decimal digits
```

Note that we found that $m = 2^e - 1$ is prime if and only if there is no prime $f < m$ where the order of two equals e. A special case is sometimes given as follows: if both $p = 4\,k + 3$ and $q = 2\,p + 1$ are prime then q divides $2^p - 1$ (because the order of 2 modulo q equals p).

By the way, if both $p = 4\,k + 1$ and $q = 2\,p + 1$ are prime then q divides $2^p + 1$ (because the order of 2 modulo q equals $2\,p$ and $2^{2\,p} - 1 = (2^p + 1)\,(2^p - 1)$).

37.11.4.3 Lucas-Lehmer test with floats *

Using the formula $\cosh(2\,x) = 2\,\cosh(x)^2 - 1$ we can give a criterion equivalent to the Lucas-Lehmer condition as follows:

$$\cosh\left(2^{m-2}\,\log\left(2+\sqrt{3}\right)\right) \quad \equiv \quad 0 \mod M_m \qquad \Longrightarrow \qquad M_m \text{ is prime} \tag{37.11-2}$$

The relation is computationally useless because the quantity to be computed grows doubly-exponentially with m: the number of digits grows exponentially with m. Already for $m = 17$ the calculation has to be carried out with more than $18,741$ decimal digits:

```
? cosh(2^(17-2)*log(2+sqrt(3)))
 1.88889395811398377260975 38478056602 E18741
```

The program [hfloat: examples/ex8.cc] does the computations in the obvious (insane) way. Using a precision of 32,768 decimal digits we obtain:

```
 cosh(...)=
  +.18888939581139837726097538478056602859465844315551 \
    [... about 18,000 digits ...]
     ... 55797500398006802841700000000000000 ...
                                ^[decimal point after 7]
    00000000000000000000000000000000000000000000000000 \
    [...]
    00000000000000001549695720446140150427588985400185472*10^18742
                    [nonzero due to numerical imprecision]
```

After rounding and computing the modulus the program declares $M_{17} = 2^{17} - 1$ prime. All this using just 4 MB of memory and computations equivalent to about 35 FFTs of length 1 million, taking about 4 seconds. This is many many million times the work needed by the original (sane) version of the test. Even trial division would have been significantly faster.

The number M_{31} would need a bigger machine as the computations needs a precision of more than 300 million digits:

```
? (2^(31-2)*log(2+sqrt(3)))/log(10)  /* approx decimal digits */
 307062001.46039800926268312190009204
```

Apart from being insane the computation can be used to test high precision floating point libraries.

37.11.4.4 The Lucas test

The Lucas-Lehmer test can be generalized for a less restricted set of candidates. The *Lucas test* can be stated as follows (taken from [200, p.131]):

Let $n = k\,2^t - 1$ were k is odd, $2^t > k$, $n \neq 0 \bmod 3$ and $k \neq 0 \bmod 3$ (so we must have $n \equiv 1 \bmod 3$). Then n is prime if and only if $H_{(n+1)/4} \equiv 0 \bmod n$ where H is as given above.

To turn this into an efficient algorithm use the relation $(n+1)/4 = k\,2^{t-2}$. Firstly, compute H_k as described in section 34.1.1 on page 635:

$$[H_k,\ H_{k+1}] \quad = \quad [H_0,\ H_1] \begin{bmatrix} 0 & -1 \\ 1 & 4 \end{bmatrix}^k \tag{37.11-3}$$

This is a one-liner in pari/gp:

```
? H(k)= return( ([1,2] * [0, -1; 1, 4]^k)[1] );
? for(k=0,10,print(k,": ",H(k),"  = 1/2 * ",2*H(k)))
  0: 1  = 1/2 * 2
  1: 2  = 1/2 * 4
  2: 7  = 1/2 * 14
  3: 26  = 1/2 * 52
  4: 97  = 1/2 * 194      /*  = 2*7^2-1 = (14^2-2)/2 */
  5: 362  = 1/2 * 724
  6: 1351  = 1/2 * 2702   /*  = 2*26^2-1 = (52^2-2)/2 */
  7: 5042  = 1/2 * 10084
  8: 18817  = 1/2 * 37634
  9: 70226  = 1/2 * 140452
  10: 262087  = 1/2 * 524174
```

To compute $H_{k\,2^{t-2}}$ from H_k use ($t-2$ times) the index doubling relation $H_{2k} = 2\,H_k^2 - 1$. The test can be implemented as

```
H(k, n)= return( (Mod([1,2],n) * Mod([0,-1; 1,4], n)^k)[1] );

lucas(k, t)=
{
    local(n, h);
    /* check preconditions: */
    if ( 0==bitand(k,1), return(0) );  \\ k must be odd
    if ( k>=2^t, return(0) );
    n = k*2^t-1;
    if ( n%3!=1, return(0) );  \\ gcd(3,k)!=0 && gcd(3,n)!=0

    /* main loop: */
    h = H(k, n);
    for (j=1,t-2, h*=h; h+=h; h-=1; );  \\ index doubling
    return ( 0==h );
}
```

Note that the routine returns 'false' even for primes if the preconditions are not met. With $n = 5{\cdot}2^{12}-1 = 20479$ we obtain

```
    n=20479  k=5  t=12
     j  H_{k*2^j} modulo n
     0      362
     1    16339
     2    17832
     3     5581
     4    18482
     5     9686
     6     8593
     7     5228
     8     5516
     9     9402
    10        0
```

which shows that 20479 is prime. Proving $n = 5 \cdot 2^{1340} - 1$ prime takes about 10 milliseconds. The following code finds the first value $t \geq 2500$ so that $n = 5 \cdot 2^t - 1$ is prime:

```
k=5;  t=2500;  while ( 0==lucas(k,t), t+=1; );  t
```

Within 1 second we obtain the result $t = 2548$.

37.11.4.5 Numbers of the form $n = 24\,j + 7$ and $n = 24\,j + 19$

```
          n:   SPP bases a<100,000 (max 5 given)
    1037623:  67191 67192                                  [--snip--]
    2211631:  6333 7260 8160 16793 21219 21580          2946282799:
    4196191:  9104 26498 93477                          3075304399:
    7076623:                                            3145717759:
    9100783:                                            3299597407:
   11418991:  44936                                     3554502799:
   15219559:                                            3554889199:
   21148399:                                            4091977039:
    [--snip--]                                          4207009999:
   829577839:
   887557999:  4899 33982 46674 62180
   961315183:
  1192222639:
    [--snip--]
```

Figure 37.11-C: Composite numbers $n < 2^{32}$ of the form $n = 24\,j + 7$ that pass the Lucas type test. Five of them are strong pseudoprime to some base smaller than $100,000$.

Numbers of the form $n = 24\,j + 7$ satisfy the preconditions of the Lucas test *except* for the condition that $2^t > k$ where $n = k\,2^t - 1$. We test whether $H_{(n+1)/4} \equiv 0 \bmod n$, as in the Lucas test. Note that $H_n = T_n(2)$ where $T_n(x)$ is the n-th Chebyshev polynomial of the first kind. We use the fast algorithm for its computation described in section 34.2.3 on page 650 for the test routine:

```
bool test_7mod24(ulong n)
{
    ulong nu = (n+1) >> 2;
    umod_t t = chebyT2(nu, n);   // == chebyT(nu, 2, n);
    return  (0==u1);
}
```

The function `chebyT2()` is given in [FXT: mod/chebyshev1.cc]. Figure 37.11-C gives composite numbers $n < 2^{32}$ that pass the test. The complete list of such numbers is given in [FXT: data/pseudo-7mod24.txt], there are just 64 entries. There are only five entries that are strong pseudoprimes to any basis $a < 100,000$, all shown in figure 37.11-C.

The data suggests that composites of the form $n = 24\,j + 7$ that pass the test and are pseudoprime to a small base are extremely rare. The implied test would cover $1/8$ of all candidates (that are not divisible by 2 or 3), as eight numbers (1, 5, 7, 11, 13, 17, 19 and 23) are prime to 24.

```
          n:   SPP bases a<100,000
      30739:                                            [--snip--]
     153931:                                           97917619:
     249331:                                          100079611:  4820
    1575859:                                          124134067:
    1960243:                                           [--snip--]
    2557627:  36814 49266 49267 86080                 2946282799:
    3444403:                                          3075304399:
    3767347:  26452 79860 94736                       3145717759:
    3881179:  47489 67676 72825 73841 84995 87856     3299597407:
    3882283:                                          3554502799:
   14324491:                                          3554889199:
   14970499:                                          4091977039:
   15894163:                                          4207009999:
```

Figure 37.11-D: Composite numbers $n < 2^{32}$ of the form $n = 24\,k + 19$ that pass the Lucas type test. Four of them are strong pseudoprime to some base smaller than $100,000$.

For numbers of the form $n = 24\,j+19$ we use a different test: here we check whether $U_{(n+1)/4-1} \equiv 0 \bmod n$ where $U_0 = 0$, $U_1 = 1$ and $U_k = 4\,U_{k-1} - U_{k-2}$ (the Chebyshev polynomial of the second kind, $U_n(x)$, evaluated at $x = 2$). The function for testing is

```
bool test_19mod24(ulong n)
{
    ulong nu = ((n+1) >> 2) - 1;
    umod_t t = chebyU2(nu, n);  // == chebyU(nu, 2, n);
    return  (0==t);
}
```

were the function `chebyU2()` is given in [FXT: mod/chebyshev2.cc]. The list [FXT: data/pseudo-19mod24.txt] contains all (155) composites $n < 2^{32}$ that pass the test. An extract is shown in figure 37.11-D. Just four numbers $n < 2^{32}$ are also strong pseudoprimes to any base $a < 100,000$.

The application of second order recurrent sequences to primality testing is described in [27]: define the sequence W_k by

$$W_k \quad = \quad P\,W_{k-1} - Q\,W_{k-2}, \qquad W_0 = 0, \quad W_1 = 1 \tag{37.11-4}$$

Then n is a *Lucas pseudoprime* (with parameters P and Q) if $W_{n\pm1} \equiv 0 \bmod n$, where the sign depends on whether $D = P^2 - 4\,Q$ is a square modulo n. For both cases considered here we have $n = 12\,j + 7$, $D = 16 - 4 = 12 = 4 \cdot 3$, and 3 is not a square modulo n. The test would be (note that $W_n = U_{n-1}(2)$)

```
bool lucas_7mod12(ulong n)
{
    ulong nu = n;
    umod_t t = chebyU2(nu, n);
    return  (0==t);
}
```

This test is passed by far more composites than the two tests considered here. A primality test combining a Lucas-type test and a test for strong pseudoprimality has been suggested in [192]. No composite that passes the test has been found so far.

37.11.4.6 An observation about Mersenne numbers

An interesting observation about Mersenne numbers is that the following *seems* to be true:

$$M = 2^e - 1 \quad \text{prime} \quad \Longleftrightarrow \quad 3^{2^{e-1}} \equiv -3 \bmod M \tag{37.11-5}$$

Note that for odd e the condition is equivalent to $3^{(M-1)/2} \equiv -1 \bmod n$ and 3 is a non-residue. For prime exponents e it can be seen that it is very unlikely to find a composite M_e where $3^{(M-1)/2} \equiv -1 \bmod n$: the number m is a strong pseudoprime (SPP) to base 2 by construction and the right hand side of condition 37.11-5 says that m is a SPP to base 3. Given the rarity of composites that are SPP to both bases (see section 37.10.2 on page 754) the chance to find such a number among the exponentially growing Mersenne numbers is very small. Tony Reix, who observed the statement of relation 37.11-5 independently verified it for prime exponents up to 132499.

37.11.4.7 Primes that are evaluations of cyclotomic polynomials

The Mersenne numbers and Fermat numbers are actually special cases of numbers obtained as cyclotomic polynomials $Y_n(x)$ (see section 38.7 on page 826) evaluated at $x = 2$ (sequence A019320 of [214]). The first such numbers are shown in figure 37.11-E. The sequence of values n such that $Y_n(2)$ is prime is entry A072226 of [214]:

```
2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 19, 22, 24, 26, 27, 30,
31, 32, 33, 34, 38, 40, 42, 46, 49, 56, 61, 62, 65, 69, 77, 78, 80, 85, 86,
89, 90, 93, 98, 107, 120, 122, 126, 127, 129, 133, 145, 150, 158, 165, 170,
174, 184, 192, 195, 202, 208, 234, 254, 261, ...
```

Now set $N := Y_n(2)$, testing whether $3^{(N-1)/2} \equiv -1 \bmod N$ seems to prove primality for all values of n. Note that for n a power of two the test is Pepin's test. Information about the primality of $Y_n(2)$ is given in [114]. Theorems about factorizations of $Y_n(x)$ where x is an integer are given in [120], see also [63] and [61].

```
 n:  s=Yn(2)
 2:  3
 3:  7
 4:  5
 5:  31
 6:  3
 7:  127
 8:  17
 9:  73
10:  11
11:  23 * 89   <--= SPP [11]
12:  13
13:  8191
14:  43
15:  151
16:  257
17:  131071
18:  3 * 19
19:  524287
20:  5 * 41
21:  7 * 337
22:  683
23:  47 * 178481
24:  241
25:  601 * 1801   <--= SPP [29]
26:  2731
27:  262657
28:  29 * 113
29:  233 * 1103 * 2089
30:  331
31:  2147483647
32:  65537
33:  599479
34:  43691
35:  71 * 122921
36:  37 * 109  <--= SPP [17, 19, 23]
37:  223 * 616318177
38:  174763
39:  79 * 121369
40:  61681
41:  13367 * 164511353
42:  5419
43:  431 * 9719 * 2099863
44:  397 * 2113
45:  631 * 23311      <--= SPP [5]
46:  2796203
47:  2351 * 4513 * 13264529
48:  97 * 673
49:  4432676798593
50:  251 * 4051
51:  103 * 2143 * 11119
52:  53 * 157 * 1613
53:  6361 * 69431 * 20394401
54:  3 * 87211
55:  881 * 3191 * 201961
56:  15790321
57:  32377 * 1212847
58:  59 * 3033169
59:  179951 * 3203431780337
60:  61 * 1321
61:  2305843009213693951
62:  715827883
63:  92737 * 649657
64:  641 * 6700417
65:  145295143558111
```

```
 n:  s=Yn(3)
 2:  2 * 2
 3:  13
 4:  2 * 5
 5:  11 * 11
 6:  7
 7:  1093
 8:  2 * 41
 9:  757
10:  61
11:  23 * 3851
12:  73
13:  797161
14:  547
15:  4561
16:  2 * 17 * 193
17:  1871 * 34511
18:  19 * 37          <--= SPP [7]
19:  1597 * 363889
20:  5 * 1181
21:  368089
22:  67 * 661
23:  47 * 1001523179
24:  6481
25:  8951 * 391151
26:  398581
27:  109 * 433 * 8209
28:  29 * 16493
29:  59 * 28537 * 20381027
30:  31 * 271         <--= SPP [29]
31:  683 * 102673 * 4404047
32:  2 * 21523361
33:  2413941289
34:  103 * 307 * 1021
35:  71 * 2664097031
36:  530713
37:  13097927 * 17189128703
38:  2851 * 101917
39:  13 * 313 * 6553 * 7333
40:  42521761
41:  83 * 2526913 * 86950696619
42:  7 * 43 * 2269
43:  431 * 380808546861411923
44:  5501 * 570461
45:  181 * 1621 * 927001
46:  23535794707
47:  1223 * 21997 * 5112661 * 96656723
48:  97 * 577 * 769
49:  491 * 4019 * 8233 * 51157 * 131713
50:  151 * 22996651
51:  12853 * 99810171997
52:  53 * 4795973261
53:  107 * 24169 * 3747607031112307667
54:  19441 * 19927
55:  11 * 1321 * 560088668384411
56:  430697 * 647753
57:  229 * 248749 * 1824179209
58:  523 * 6091 * 5385997
59:  14425532687 * 489769993189671059
60:  47763361
61:  603901 * 105293313660391861035901
62:  6883 * 22434744889
63:  1445429185285300809
64:  2 * 926510094425921
65:  131 * 3701101 * 110133112994711
```

Figure 37.11-E: Evaluations s of the first cyclotomic polynomials at two (left). Entries at prime n are Mersenne numbers M_n, entries at $n = 2^k$ are Fermat numbers F_k. Composites that are strong pseudoprimes to prime bases other than two are marked with 'SPP'. The right columns show the corresponding data for evaluations at three.

The primes $Y_n(2)$ are also of interest for number theoretic transforms (see section 25.1 on page 507) because of their special structure allowing for very efficient modular reduction (see section 37.2 on page 735). A prominent example is $Y_{192}(2) = 2^{64} - 2^{32} + 1$. Note that the order of 2 modulo $Y_n(2)$ equals n.

The structure of the primes becomes (in base 10) visible if we check evaluations at 10, the first primes $Y_n(10)$ are

```
  n:   Yn(10)
  2:   11
  4:   101
 10:   9091
 12:   9901
 14:   909091
 19:   1111111111111111111
 23:   11111111111111111111111
 24:   99990001
 36:   999999000001
 38:   909090909090909091
 39:   900900900900990990990991
 48:   9999999900000001
```

Finally, we do a silly thing: the factors of $Y_{2^7-1}(x)$ over $GF(2)$ are the irreducible binary polynomials of degree 7. If we evaluate them as polynomials over $\mathbb{Z}$ at $x = 10$ and select the prime numbers we obtain 8-digit primes consisting of only zero and ones:

```
10011101    10111001    11100101    11110111    11111101
```

The same procedure, with $Y_{3^5-1}(x)$ and factoring over $GF(3)$ gives the primes

```
101221    102101    111121    111211    112111    120011    122021
```

The list is created via

```
n=3^5-1;  f=lift(factor(polcyclo(n)*Mod(1,3)));  f=f[,1];
for(k=1, #f, v=subst(f[k],x,10); if(isprime(v), print(v)));
```

Further reading

Excellent introductions into topics related to prime numbers and methods of factorization are [200], [248], and [92]. Primality tests and factorization algorithms are also described in [119]. Some of the newer factorization algorithms can be found in [83], readable surveys are [167] and [182]. Tables of factorizations of numbers of the form $b^e \pm 1$ are given in [68] which also contains much historical information.

It remains to say that a deterministic polynomial-time algorithm for proving primality was published by Agrawal, Kayal and Saxena in August 2002 [4]. While this a major breakthrough in mathematics it does not render the Rabin-Miller test worthless. Practically, 'industrial grade' primes are still produced with it, see [66]. Good introductions into the ideas behind the so-called *AKS algorithm* and its improvements are [127] and [92, p.200ff].

37.12 Complex moduli: GF(p^2)

Recall that with real numbers the equation $x^2 = -1$ has no solution, there is no real square root of minus one. The construction of complex numbers proceeds by taking pairs of real numbers $(a, b) = a + i\,b$ together with (component wise) addition $(a, b) + (c, d) = (a + c, b + d)$ and multiplication defined by $(a, b)\,(c, d) = (ac - bd, ad + bc)$. Indeed the pairs of real numbers together with addition and multiplication as given constitute a field.

We will now rephrase the construction in a way that shows how to construct an *extension field* from a given *ground field* (or *base field*). In the example above the ground field are the real numbers and the extension field are the complex numbers.

37.12.1 The construction of complex numbers

Equivalently to saying that there is no real square root of minus one we could have said that the polynomial x^2+1 has no linear factor. The construction of the complex numbers proceeds by taking numbers of the form $a+b\,i$ where i was boldly defined to be a root of the polynomial x^2+1. Now observe that if we identify $a+b\,i = b\,i+a$ with the polynomial $b\,x+a$ and use polynomial addition and multiplication modulo the polynomial x^2+1 we obtain the arithmetic of complex numbers. Addition is component wise, no modular reduction occurs. The multiplication rule can be obtained with polynomial arithmetic:

$$\begin{aligned}(b\,x+a)\;(d\,x+c) &= (bd)\,x^2+(ad+bc)\,x+(ac) && \text{(37.12-1a)}\\ &\equiv (ad+bc)\,x+(ac-bd) \qquad (\mathrm{mod}\ x^2+1) && \text{(37.12-1b)}\end{aligned}$$

We used the relation $x^2=-1$, so $u\,x^2 \equiv -u \pmod{x^2+1}$. Identify x with i in the relations and observe that computations with a root (that is not in the ground field) of a polynomial is equivalent to computations with polynomials (whose coefficients are in the ground field) modulo the polynomial x^2+1.

When the ground field is the real numbers the story comes to an end: every polynomial of arbitrary degree n with complex coefficients has exactly n complex roots (including multiplicity). That is, we cannot use the given construction to extend the field $\mathbb{C}$ of complex numbers as every polynomial $p(x)$ with coefficients in $\mathbb{C}$ is a product of linear factors: $p(x)=(x-r_1)\,(x-r_2)\ldots,(x-r_n)$ with $r_k\in\mathbb{C}$. The field $\mathbb{C}$ is algebraically closed.

If we choose the ground field to be $\mathbb{F}_p = \mathrm{GF}(p)$, the integers modulo a prime p, and an irreducible polynomial $c(x)$ of degree n whose coefficients are in $\mathbb{F}_p$ we get an extension field $\mathbb{F}_{p^n}=\mathrm{GF}(p^n)$, a *finite field* with p^n elements. The special case of the *binary finite fields* $\mathrm{GF}(2^n)$ is treated in chapter 40 on page 851.

37.12.2 Complex finite fields

With primes of the form $p=4k+3$ it is possible to construct a field of complex numbers as minus one is a quadratic non-residue and so the polynomial x^2+1 is irreducible. The field is denoted by $\mathrm{GF}(p^2)$.

The rules for complex addition, subtraction and multiplication are the 'usual' ones. The field has p^2 elements of which $R=p^2-1$ are invertible. The maximal order equals R, so the inverse of and element a can be computed as $a^{-1}=a^{R-1}=a^{p^2-2}$.

For example, the powers of $a=1+x=1+i$ modulo $c=x^2+1=0=3+3i$ are

```
a= 1+1*i

a^1 =  1+1*i
a^2 =  0+2*i  // (1+x)*(1+x)   = x^2+2*x+1   == 2*x+1 - 1 = 2*x+0 == 0+2*x
a^3 =  1+2*i  // (1+x)*(0+2*x) = 2*x^2+2*x   == 2*x   - 2 = 2*x-2 == 1+2*x
a^4 =  2+0*i  // (1+x)*(1+2*x) = 2*x^2+3*x+1 == 3*x+1 - 2 = 3*x-1 == 2+0*x
a^5 =  2+2*i  //                              ^^                    ^^
a^6 =  0+1*i  //                          mod(x^2+1)            mod(3)
a^7 =  2+1*i == a^(-1)= 2+1*i
a^8 =  1+0*i == one
a^9 =  1+1*i

R=maxord==8 == Mat([2, 3])
r=ord(a)==8
R/r=1
```

Note that the modular reduction happens with both the polynomial x^2+1 and $p=3$. The polynomial reduction uses $x^2=-1$.

With primes of the form $p=4k+1$ it also possible to construct a field $\mathrm{GF}(p^2)$. But we have to use a different polynomial as x^2+1 is reducible modulo p and thereby the multiplication rules is different. For example, with $p=5$ we find that x^2+x+1 is irreducible:

```
? p=5;  m=Mod(1,p)*(1+x+x^2);  polisirreducible(m)
  1
a=Mod(1,p)*(1+3*x)
for(k=1,p^2-1,print("a^",k," = ",component(Mod(a,m)^k,2)))
  a^1 = Mod(3, 5)*x + Mod(1, 5)
  a^2 = Mod(2, 5)*x + Mod(2, 5)
  a^3 = Mod(2, 5)*x + Mod(1, 5)
  [...]
```

The output, slightly beautified:

```
    a^1  = 1 + 3*x
    a^2  = 2 + 2*x  // (1+3*x)*(1+3*x) = 9*x^2+6*x+1 == 6*x+1 - (9*x+9) =-3*x-8 == 2+2*x
    a^3  = 1 + 2*x  // (1+3*x)*(2+2*x) = 6*x^2+8*x+2 == 8*x+2 - (6*x+6) = 2*x-4 == 1+2*x
    a^4  = 0 + 4*x  // (1+3*x)*(1+2*x) = 6*x^2+5*x+1 == 5*x+1 - (6*x+6) =  -x-5 == 0+4*x
    a^5  = 3 + 2*x  //                                  ^^                    ^^
    a^6  = 2 + 0*x  //                           mod(x^2+x+1)                mod(5)
    a^7  = 2 + 1*x
    a^8  = 4 + 4*x
    a^9  = 2 + 4*x
    a^10 = 0 + 3*x
    a^11 = 1 + 4*x
    a^12 = 4 + 0*x
    a^13 = 4 + 2*x
    a^14 = 3 + 3*x
    a^15 = 4 + 3*x
    a^16 = 0 + 1*x
    a^17 = 2 + 3*x
    a^18 = 3 + 0*x
    a^19 = 3 + 4*x
    a^20 = 1 + 1*x
    a^21 = 3 + 1*x
    a^22 = 0 + 2*x
    a^23 = 4 + 1*x
    a^24 = 1 + 0*x
```

We see that $a = 1 + 3\,x$ has the maximal order (24), it is a primitive root. The polynomial reduction uses the relation $x^2 = -(x + 1)$.

The values of the powers of the primitive root can be used to 'randomly' fill a $p \times p$ array. With $a^k = u + x\,v$ we mark the entry at row v, column u with k:

```
[ 4 11  9 19  8]
[10  1 17 14 15]
[22  3  2  5 13]
[16 20  7 21 23]
[-- 24  6 18 12]
```

The position $0, 0$ (lower left) is not visited. Note row zero is the lowest row.

As described, the procedure fills an $p \times p$ array where p is a prime. Working with an irreducible polynomial of degree n we can fill a $p^e \times p^f \times p^g \times \ldots \times p^k$ array if $e + f + g + \ldots + k = n$: for the exponents that are one just choose one polynomial coefficient. For the exponents greater one (say, h) combine h polynomial coefficients $c_0,\ c_1,\ \ldots,\ c_{h-1}$ and use $z_h = c_0 + c_1\,p + c_2\,p^2 + \ldots c_{h-1}\,p^{h-1}$.

37.12.3 Efficient reduction modulo certain quadratic polynomials

The polynomial $C = x^2 + 1$ is irreducible for primes of the form $4\,k + 3$ (-1 is not a square)

$$\begin{aligned}
(a\,x + b)\,(A\,x + B) &= (a\,A)\,x^2 + (a\,B + b\,A)\,x + (b\,B) && \text{(37.12-2a)}\\
&\equiv (a\,B + b\,A)\,x + (-a\,A + b\,B) \mod x^2 + 1 && \text{(37.12-2b)}\\
&= ((a + b)\,(A + B) - a\,A - b\,B)\ x + (-a\,A + b\,B) && \text{(37.12-2c)}
\end{aligned}$$

The last equality shows how to multiply two complex numbers at the cost of three real multiplications and five real additions instead of four multiplications and two additions.

The polynomial $C = x^2 + d$ is irreducible if $-d$ is not a square. We have

$$\begin{aligned}
(a\,x + b)\,(A\,x + B) &\equiv (a\,B + b\,A)\,x + (-d\,a\,A + b\,B) \mod x^2 + d && \text{(37.12-3a)}\\
&= ((a + b)\,(A + B) - a\,A - b\,B)\ x + (-d\,a\,A + b\,B) && \text{(37.12-3b)}
\end{aligned}$$

If the multiplication by d is cheap (for example, if $d = 2$) the implied technique can be a gain.

The polynomial $C := x^2 + x + 1$ has the roots $\left(-1 \pm \sqrt{-3}\right)/2$ so it is irreducible modulo p if -3 is not a square modulo p. The first few such primes p are

```
2 5 11 17 23 29 41 47 53 59 71 83 89 101 107 113 131 137 149 167 173 179 191 197 227 233 239
```

Multiplication modulo C costs only three scalar multiplications:

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &= (a\,A)\,x^2 + (a\,B+b\,A)\,x + (b\,B) && (37.12\text{-}4a)\\ &\equiv (-a\,A+a\,B+b\,A)\,x + (-a\,A+b\,B) \mod x^2+x+1 && (37.12\text{-}4b)\\ &= ((a-b)\,(B-A)+b\,B)\;x + (-a\,A+b\,B) && (37.12\text{-}4c)\end{aligned}$$

For the polynomial $C = x^2 + x + d$ use

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &\equiv (-a\,A+a\,B+b\,A)\,x + (-d\,a\,A+b\,B) \mod x^2+x+d && (37.12\text{-}5a)\\ &= ((a-b)\,(B-A)+b\,B)\;x + (-d\,a\,A+b\,B) && (37.12\text{-}5b)\end{aligned}$$

The polynomial $C = x^2 - x - 1$ has the roots $\left(1 \pm \sqrt{5}\right)/2$, so it is irreducible modulo p if 5 is not a square modulo p. The first few such primes are:

```
2 3 7 13 17 23 37 43 47 53 67 73 83 97 103 107 113 127 137 157 163 167 173 193 197 223 227 233
```

Multiplication modulo C again costs only three scalar multiplications:

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &= (a\,A)\,x^2 + (a\,B+b\,A)\,x + (b\,B) && (37.12\text{-}6a)\\ &\equiv (a\,A+a\,B+b\,A)\,x + (a\,A+b\,B) \mod x^2-x-1 && (37.12\text{-}6b)\\ &= ((a+b)\,(A+B)-b\,B)\;x + (a\,A+b\,B) && (37.12\text{-}6c)\end{aligned}$$

With the polynomial $C = x^2 - x - d$ use

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &\equiv (a\,A+a\,B+b\,A)\,x + (d\,a\,A+b\,B) \mod x^2-x-d && (37.12\text{-}7a)\\ &= ((a+b)\,(A+B)-b\,B)\;x + (d\,a\,A+b\,B) && (37.12\text{-}7b)\end{aligned}$$

For polynomials of the form $C = x^2 - e\,x - d$ one has

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &= (a\,A)\,x^2 + (a\,B+b\,A)\,x + (b\,B) && (37.12\text{-}8a)\\ &\equiv (e\,a\,A+a\,B+b\,A)\,x + (d\,a\,A+b\,B) \mod x^2-e\,x-d && (37.12\text{-}8b)\\ &= ((a+b)\,(A+B)-[e-1]\,a\,A-b\,B)\;x + (d\,a\,A+b\,B) && (37.12\text{-}8c)\end{aligned}$$

If the multiplications by $e-1$ and d are cheap then the last equality can be useful. For example, with the polynomial $C = x^2 - 2\,x - 1$ use

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &\equiv (2\,a\,A+a\,B+b\,A)\,x + (a\,A+b\,B) \mod x^2-2\,x-1 && (37.12\text{-}9a)\\ &= ((a+b)\,(A+B)-b\,B+a\,A)\;x + (a\,A+b\,B) && (37.12\text{-}9b)\end{aligned}$$

With $C = x^2 - 3\,x \pm 1$ use

$$\begin{aligned}(a\,x+b)\,(A\,x+B) &\equiv (3\,a\,A+a\,B+b\,A)\,x + (\mp a\,A+b\,B) \mod x^2-3\,x\pm 1 && (37.12\text{-}10a)\\ &= ((a+b)\,(A+B)-b\,B+2\,a\,A)\;x + (\mp a\,A+b\,B) && (37.12\text{-}10b)\end{aligned}$$

37.12.4 An algorithm for primitive 2^j-th roots

For primes with the lowest k bits set ($p \equiv (2^k - 1) \pmod{2^k}$) the largest power of two dividing the maximal order in GF(p^2) (with field polynomial $x^2 + 1$) equals $N = 2^{k+1}$: $p = j\,2^k - 1$ with j odd, so $p + 1 = j\,2^k$ and $p - 1 = j\,2^k - 2 = 2\,(j\,2^{k-1} - 1)$, thereby $p^2 - 1 = 2^{k+1}\,[j\,(j\,2^k - 1)]$ where the term in square brackets is odd.

An algorithm for the construction of primitive 2^j-th roots in GF(p^2) for $j = 2, 3, \ldots, a$ where 2^a is the largest power of two dividing $p^2 - 1$ is given in [115] (and also in [42]):

Let $u_2 := 0$ and for $j > 2$ define

$$u_j \;:=\; \begin{cases} \big((u_{j-1}+1)/2\big)^{(p+1)/4} & \text{if} \quad j < a \\ \big((u_{j-1}-1)/2\big)^{(p+1)/4} & \text{if} \quad j = a \end{cases} \tag{37.12-11}$$

and (for $j = 2, 3, \ldots, a$)

$$v_j \;:=\; \begin{cases} \big(+1-u_j^2\big)^{(p+1)/4} & \text{if} \quad j < a \\ \big(-1-u_j^2\big)^{(p+1)/4} & \text{if} \quad j = a \end{cases} \tag{37.12-12}$$

where all operations are modulo p. Then $u_j + i\,v_j$ is a primitive 2^j-th root of unity in GF(p^2).

For example, with $p = 127$ we get

```
j:       u_j  v_j
2:    ord(0 + i*1) = 4
3:    ord(8 + i*8) = 8
4:    ord(103 + i*21) = 16
5:    ord(68 + i*87) = 32
6:    ord(15 + i*41) = 64
7:    ord(32 + i*82) = 128
8:    ord(98 + i*38) = 256
```

For Mersenne primes $p = 2^e - 1$ one has $p^2 - 1 = (p+1)\,(p-1) = 2^e\,(2^e - 2) = 2^{e+1}\,(2^{e-1} - 1) =: 2^a\,k$ where k is odd. The highest power of two for which a primitive root exists is $a = 2^{e+1}$ which checks with our example where $p = 127 = 2^7 - 1$.

37.12.5 Primitive 2^j-th roots with Mersenne primes

```
p =  131071  = 2^17-1
r3 =   43811     r3^2 =  -3

k:    h + i*w =    z^(2^k)            =   h + i*r3*  u
0:    h + i*w =    +256 + i* -56490   =   h + i*r3*    +256    ord(.) = 2^18
1:    h + i*w =      +2 + i* +43811   =   h + i*r3*      +1    ord(.) = 2^17
2:    h + i*w =      +7 + i* +44173   =   h + i*r3*      +4    ord(.) = 2^16
3:    h + i*w =     +97 + i* -36933   =   h + i*r3*     +56    ord(.) = 2^15
4:    h + i*w =  +18817 + i* +43903   =   h + i*r3*  +10864    ord(.) = 2^14
5:    h + i*w =  -17636 + i* -35524   =   h + i*r3*  +45327    ord(.) = 2^13
6:    h + i*w =   -5975 + i* -36232   =   h + i*r3*  +30114    ord(.) = 2^12
7:    h + i*w =  -32446 + i* +44887   =   h + i*r3*  +58666    ord(.) = 2^11
8:    h + i*w =  -38713 + i* -16371   =   h + i*r3*   +3123    ord(.) = 2^10
9:    h + i*w =  +61109 + i* -46595   =   h + i*r3*  +24597    ord(.) = 2^9
10:   h + i*w =  +63110 + i* +25098   =   h + i*r3*  -48310    ord(.) = 2^8
11:   h + i*w =  +35245 + i* +14561   =   h + i*r3*   -3138    ord(.) = 2^7
12:   h + i*w =  -30756 + i* -12111   =   h + i*r3*  +50228    ord(.) = 2^6
13:   h + i*w =  -15743 + i* -35732   =   h + i*r3*  -19124    ord(.) = 2^5
14:   h + i*w =  -26425 + i* -55712   =   h + i*r3*   -1910    ord(.) = 2^4
15:   h + i*w =    -256 + i*   +256   =   h + i*r3*  +18830    ord(.) = 2^3
16:   h + i*w =       0 + i*     -1   =   h + i*r3*  +58294    ord(.) = 2^2
17:   h + i*w =      -1 + i*      0   =   h + i*r3*       0    ord(.) = 2^1
18:   h + i*w =      +1 + i*      0   =   h + i*r3*       0    ord(.) = 2^0
```

Figure 37.12-A: Elements of order 2^j in GF(p^2) where $p = 2^{17} - 1$ is a Mersenne prime.

For Mersenne primes $p = 2^e - 1$ a primitive root of order 2^{e+1} (in GF(p^2) with field polynomial $x^2 + 1$) can be constructed more directly: first compute $\sqrt{-3} = 3^{(p+1)/4} = 3^{2^{e-2}}$ by squaring $e - 2$ times, then compute $1/\sqrt{2} = 2^{(e-1)/2}$ which does not require modular reduction. Now

$$z \;:=\; \frac{1}{\sqrt{2}}\left(1 + \sqrt{3}\right) \;=\; \frac{1}{\sqrt{2}}\left(1 + i\,\sqrt{-3}\right) \tag{37.12-13}$$

is an element of order 2^{e+1}. The number z is sometimes called the Creutzburg-Tasche primitive root as the construction was given in [93, p.200]. We have $z^2 = 2+\sqrt{3} = 2+i\sqrt{-3} = H_1 + i\sqrt{-3}\,U_1$ and, more general

$$z^{2^k} \quad = \quad H_{2^{k-1}} + i\sqrt{-3}\,U_{2^{k-1}} \qquad (1 \le k) \tag{37.12-14}$$

Figure 37.12-A shows the values of the successive 2^k-th powers of z in GF(p^2) where $p = 2^{17}-1$.

The sequences $H = 2,\ 7,\ 97,\ 18817,\ \ldots$ and $U = 1,\ 4,\ 56,\ 10864,\ \ldots$ are those which appear in the Lucas-Lehmer test. The order of z^{2^k} is 2^{e+1-k}.

We have $H_{2j} = 2\,H_j^2 - 1$ and $H_j^2 - 3\,U_j^2 = 1$, thereby

$$H_{2j} \quad = \quad H_j^2 + 3\,U_j^2 \tag{37.12-15a}$$
$$U_{2j} \quad = \quad 2\,H_j\,U_j \tag{37.12-15b}$$

These are the index doubling formulas for the convergents of the continued fraction of $\sqrt{3}$.

37.12.6 Cosine and sine in GF(p^2)

Let z be an element of order n in GF(p^2), we would like to identify z with $\exp(2\,i\,\pi/n)$ and determine the values equivalent to $\cos(2\,\pi/n)$ and $\sin(2\,\pi/n)$ with complex numbers over $\mathbb{R}$. One can set

$$\cos\frac{2\,\pi}{n} \quad \equiv \quad \frac{z^2+1}{2\,z} \tag{37.12-16a}$$
$$i\,\sin\frac{2\,\pi}{n} \quad \equiv \quad \frac{z^2-1}{2\,z} \tag{37.12-16b}$$

For the choice of sin and cos both relations $\exp(x) = \cos(x) + i\,\sin(x)$ and $\sin(x)^2 + \cos(x)^2 = 1$ should hold. The first check is trivial: $\frac{z^2+1}{2\,z} + \frac{z^2-1}{2\,z} = z$. The second is also easy if we write i for some element that is the square root of -1: $(\frac{z^2+1}{2\,z})^2 + (\frac{z^2-1}{2\,z\,i})^2 = \frac{(z^2+1)^2-(z^2-1)^2}{4\,z^2} = 1$.

For the 2^j-th roots in GF(127^2) we obtain

```
0:    u+i*v=   1 + i* 0          cos=   1    i*sin=  0
1:    u+i*v= 126 + i* 0          cos= 126    i*sin=  0
2:    u+i*v=   0 + i* 1          cos=   0    i*sin=  1*i
3:    u+i*v=   8 + i* 8          cos=   8    i*sin=  8*i
4:    u+i*v= 103 + i*21          cos= 103    i*sin= 21*i
5:    u+i*v=  68 + i*87          cos=  68    i*sin= 87*i
6:    u+i*v=  15 + i*41          cos=  15    i*sin= 41*i
7:    u+i*v=  32 + i*82          cos=  32    i*sin= 82*i
8:    u+i*v=  98 + i*38          cos=  38*i  i*sin= 98
```

Note how the i swaps side with the element of highest order 2^a.

The simple construction allows us to mechanically convert fast Fourier (or Hartley) transforms with explicit trigonometric constants into the corresponding number theoretic transforms. The idea of expressing cosines and sines in terms of primitive roots was taken from [228].

37.12.7 Cosine and sine in GF(p)

What about primes of the form $p = 4k+1$ that are used anyway for NTTs? The same construction works. The polynomial x^2+1 is reducible modulo $p = 4k+1$, equivalently, -1 is a quadratic residue so its square root lies in GF(p). We could say: i is real modulo p if p is of the form $4k+1$.

In the implementation [FXT: `class mod` in mod/mod.h] the cosine and sine values are computed from the primitive roots of order 2^j. The program [FXT: mod/modsincos-demo.cc] generates the list of 2^j-th roots and inverse roots shown in figure 37.12-B.

```
modulus= 257 == 0x101
modulus is cyclic
modulus is prime
bits(modulus)= 8.0056245  == 9 - 0.99437545
euler_phi(modulus)= 256 == 0x100 == 2^8
maxorder= 256 == 0x100
maxordelem= 3 == 0x3
max2pow= 8   (max FFT length = 2**8 == 256)
root2pow(max2pow)=3   root2pow(-max2pow)=86
sqrt(-1) =: i = 241

 8:    z=   3  = ( 173   +  87)  =  ( 173   + 107*i)
 7:    z=   9  = ( 233   +  33)  =  ( 233   +  14*i)
 6:    z=  81  = ( 123   + 215)  =  ( 123   +  99*i)
 5:    z= 136  = ( 188   + 205)  =  ( 188   + 196*i)
 4:    z= 249  = (  12   + 237)  =  (  12   + 194*i)
 3:    z=  64  = (  30   +  34)  =  (  30   +  30*i)
 2:    z= 241  = (   0   + 241)  =  (   0   +   1*i)
 1:    z= 256  = ( 256   +   0)  =  ( 256   +   0*i)
 0:    z=   1  = (   1   +   0)  =  (   1   +   0*i)
-1:    z= 256  = ( 256   +   0)  =  ( 256   +   0*i)
-2:    z=  16  = (   0   +  16)  =  (   0   + 256*i)
-3:    z= 253  = (  30   + 223)  =  (  30   + 227*i)
-4:    z=  32  = (  12   +  20)  =  (  12   +  63*i)
-5:    z= 240  = ( 188   +  52)  =  ( 188   +  61*i)
-6:    z= 165  = ( 123   +  42)  =  ( 123   + 158*i)
-7:    z= 200  = ( 233   + 224)  =  ( 233   + 243*i)
-8:    z=  86  = ( 173   + 170)  =  ( 173   + 150*i)
```

Figure 37.12-B: Roots of order 2^j modulo $p = 257 = 2^8 + 1$.

Again we can translate a given FFT implementation in a mechanical way.

An element modulo a prime $p = k \cdot 2^t + 1$ whose order equals 2^t can be found by the following algorithm even if the factorization of k is not known:
Choose random a where $1 < a < p-1$ and compute $s = a^k$, if $-1 = s^{2^{t-1}}$ then return s, else try another a.

The algorithm will terminate when the first element a is encountered whose order has the factor 2^t. Demonstration in pari/gp:

```
el2(k, t)=
{
    local(p, s);
    p=k*2^t+1;
    for(a=2, p-2,
        s=Mod(a,p)^k;
        if( Mod(-1,p)==s^(2^(t-1)), return( s ); );
    );
}
```

With $p = 314151729239163 \cdot 2^{26} + 1$ the algorithm terminates after testing $a = 5$ (of order $(p-1)/3$) and returning $s = 18583781386455525528042$ whose order is indeed 2^{26}.

In general, if $p = u \cdot f + 1$, $\gcd(f, u) = 1$ and $f = \prod p_i^{e_i}$ is fully factored then an element of order f can be determined by testing random values a:

1. Take a random a and set $s = a^u$.

2. If $s^{f/p_i} \neq 1$ for all prime factors p_i of f, then return s (an element of order f).

3. Go to step 1.

37.12.8 Decomposing $p = 4\,k + 1$ as sum of two squares

37.12.8.1 Direct computation

The direct way to determine u and v with $n = u^2 + v^2$ is to check (for $v = 1, 2, \ldots, \lfloor\sqrt{n}\rfloor$) whether $n - v^2$ is a perfect square. If so, return $u = \sqrt{n - v^2}$ and v:

```
sumofsquares_naive(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(n, w);
    n = k*2^t+1;
    for (v=1, sqrtint(n),  \\ search until n-v^2 is a square
        w = n-v^2;
        if ( issquare(w), return( [sqrtint(w), v] ) );
    );
    return ( 0 );  \\ not the sum of two squares
}
```

The routine needs at most $\lfloor\sqrt{n}\rfloor$ steps which renders it rather useless for n large: With the prime $n = 314151729239163 \cdot 2^{26} + 1 \approx 2 \cdot 10^{22}$ we have $n = u^2 + v^2$ where $u = 132599472793$ and $v = 59158646772$ and the routine would need v steps to find the solution. The method described next finds the solution immediately.

37.12.8.2 Computation using continued fractions

The square root i of -1 can be used to find the representation of a prime $p = 4\,k + 1$ as a sum of two squares, $p = u^2 + v^2$, as follows:

1. Determine i where $i^2 = -1$ modulo p. If $i \geq p/2$ then set $i = p - i$.
2. Compute the continued fraction of p/i, it has the form $[a_0, a_1, \ldots, a_n, a_n, \ldots, a_1, a_0]$.
3. Compute the numerators of the $(n-1)$-st and the n-th convergent, P_{n-1} and P_n. Return $u = P_{n-1}$ and $v = P_n$.

Assume that $p = k \cdot 2^t + 1$ where $t \geq 2$. Use an element of order 2^t to find a square root of -1:

```
imag4k1(k, t)=
{ /* determine x so that x^2=-1 modulo p=k*2^t+1 */
    local(s);
    s = el2(k, t);
    s = s^(2^(t-2));
    return( s );
}
```

Then

```
sumofsquares(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(i, s, p, cf, q, u, v);
    i = component( imag4k1(k, t), 2);
    p = k*2^t+1;
    if ( i>=p/2, i = p-i );
    cf = contfrac(p/i);
    cf = vector(length(cf)/2, j, cf[j]);
    q = contfracpnqn(cf);
    u = q[1, 1];  v = q[1, 2];
    return( [u, v] );
}
```

An example, for $p = 2281$ we obtain

```
   i = 1571  \\ square root of -1
   i =  710  \\ choose smaller square root
   cf = [3, 4, 1, 2, 2, 1, 4, 3]  \\ == contfrac(2281/710)
   cf = [3, 4, 1, 2]  \\ first half on contfrac
  q = contfracpnqn(cf) =
    [45 16]  \\ == [P_4, P_3]
```

```
    [14 5]   \\ == [Q_4, Q_3] (unused)
    u=45;  v=16;  \\ u^2 + v^2 = 2025 + 256 = 2281
```

37.12.8.3 A memory saving version

An algorithm that avoids storing the continued fraction comes from the observation that u and v appear in the calculation of $\gcd(p, i)$. With

```
gcd_print(p, i)=
{
    local( t, s );
    if ( p<i,  t=p; p=i; i=t; );
    s = sqrtint(p);
    while ( i,
        print ("  ", p, "  ", i);
        t = p % i;  p = i;  i = t;
    );
}
```

and $p = 2281$, $i = 710$ we obtain

```
      u    v
   2281  710
    710  151
    151  106
    106   45
     45   16 <--=
     16   13
     13    3
      3    1
```

The marked pair is the first where $u^2 < p$. The resulting routine is

```
sumofsquares_gcd(k, t)=
{ /* return [u,v] so that u^2+v^2==p =k*2^t+1 */
    local(s, p, i, w);
    i = component( imag4k1(k, t), 2);
    p = k*2^t+1;
    if ( i>=p/2, i = p-i );
    w = sqrtint(p);
    while ( i,
        if ( p<=w, return( [p,i] ) );
        t = p % i;  p = i;  i = t;
    );
    return( [0,0] ); \\ failure
}
```

We note that by the relation $a^2+b^2 = (a+i\,b)\,(a-i\,b)$ we can use the decomposition into two squares to obtain the factorization of a number over the complex integers. For example, we have $3141592653 = 3 \cdot 107 \cdot 9786893$ (over $\mathbb{Z}$) where the greatest prime factor is of the form $4k+1$. Using $9786893 = 2317^2+2102^2$ one can obtain $3141592653 = -i \cdot 3 \cdot 107 \cdot (2317 + 2102i) \cdot (2102 + 2317i)$. Pari/gp has a builtin routine for this task:

```
? factor(3141592653+0*I)
   [-I 1]   [3 1]   [107 1]   [2317 + 2102*I 1]   [2102 + 2317*I 1]
```

37.13 Solving the Pell equation

Simple continued fractions can be used to find integer solutions of the equations

$$x^2 - d\,y^2 = +1 \tag{37.13-1a}$$

$$x^2 - d\,y^2 = -1 \tag{37.13-1b}$$

Equation 37.13-1a is usually called the *Pell equation*. The name *Bhaskara equation* (or *Brahmagupta-Bhaskara equation*, used in [160]) seems more appropriate as Brahmagupta (ca. 600 AD) and Bhaskara (ca. 1100 AD) were the first to study and solve this equation.

37.13.1 Solution via continued fractions

The convergents P_k/Q_k of the continued fraction of $\sqrt{d}$ are close to $\sqrt{d}$: $(P_k/Q_k)^2 \approx d$. If we define $e_k := P_k^2 - d\,Q_k^2$ then solutions of relation 37.13-1a correspond to $e_k = +1$, solutions of 37.13-1b to $e_k = -1$.

As an example we set $d = 53$. The continued fraction of $\sqrt{53}$ is

$$\begin{aligned} \mathrm{CF}(\sqrt{53}) &= [7,\ 3,1,1,3,14,\ 3,1,1,3,14,\ 3,1,1,3,14,\ \ldots] && \text{(37.13-2a)} \\ &= [7,\ \overline{3,1,1,3,14}] && \text{(37.13-2b)} \end{aligned}$$

We observe that the sequence is periodic after the initial term and the last term of the period is twice the initial term. Moreover, disregarding the term 14, the terms in the period form a palindrome. These properties actually hold for all simple continued fractions of square roots $\sqrt{d}$ with d not a perfect square, for the proofs the reader is referred to either [187] or [160]. For the computation of the continued fraction of a square root a specialized version of the algorithm from section 35.3.1.2 on page 681 will be most efficient.

k :	a_k	P_k	Q_k	$e_k := P_k^2 - d\,Q_k^2$
−1:	−	0	1	+1
0:	7	7	1	−4
1:	3	22	3	+7
2:	1	29	4	−7
3:	1	51	7	+4
4:	3	182	25	−1
5:	14	2599	357	+4
6:	3	7979	1096	−7
7:	1	10578	1453	+7
8:	1	18557	2549	−4
9:	3	66249	9100	+1
10:	14	946043	129949	−4
11:	3	2904378	398947	+7
12:	1	3850421	528896	−7
13:	1	6754799	927843	+4
14:	3	24114818	3312425	−1
15:	14	344362251	47301793	+4

Figure 37.13-A: The first convergents P_k/Q_k of the continued fraction of $\sqrt{53}$.

The table shown in figure 37.13-A gives the first convergents P_k/Q_k together with $e_k := P_k^2 - d\,Q_k^2$ ($d = 53$). The entry for $k = 4$ corresponds to the smallest solution (x, y) of $x^2 - 53\,y^2 = -1$: $182^2 - 53 \cdot 25^2 = -1$. Entry $k = 9$ corresponds to $66249^2 - 53 \cdot 9100^2 = +1$, the smallest nontrivial solution to $x^2 - 53\,y^2 = +1$ (the trivial solution is $(P_{-1}, Q_{-1}) = (1, 0)$).

With $d = 19$ we obtain the continued fraction

$$\begin{aligned} \mathrm{CF}(\sqrt{19}) &= [4,\ 2,1,3,1,2,8,\ 2,1,3,1,2,8,\ 2,1,3,1,2,8,\ \ldots] && \text{(37.13-3a)} \\ &= [4,\ \overline{2,1,3,1,2,8}] && \text{(37.13-3b)} \end{aligned}$$

with period length $l = 6$. The corresponding table (figure 37.13-B) contains solutions with $e_k = +1$ but none with $e_k = -1$:

Let e correspond the minimal nontrivial solution of $x^2 - d\,y^2 = \pm 1$, if $e = +1$ then no solution for $x^2 - d\,y^2 = -1$ exists. Nontrivial solutions with $e = +1$ always exist, solutions with $e = -1$ only exist when the period length l of the continued fraction of $\sqrt{d}$ is odd. The period length l is always odd for

$k:$	a_k	P_k	Q_k	$e_k := P_k^2 - d\,Q_k^2$
$-1:$	$-$	0	1	$+1$
0:	4	4	1	-3
1:	2	9	2	$+5$
2:	1	13	3	-2
3:	3	48	11	$+5$
4:	1	61	14	-3
5:	2	170	39	$+1$
6:	8	1421	326	-3
7:	2	3012	691	$+5$
8:	1	4433	1017	-2
9:	3	16311	3742	$+5$
10:	1	20744	4759	-3
11:	2	57799	13260	$+1$
12:	8	483136	110839	-3

Figure 37.13-B: The first convergents P_k/Q_k of the continued fraction of $\sqrt{19}$.

primes of the form $p = 4\,k+1$ and never for numbers of the form $n = 4\,k+3$ and $4\,k$. If any factor f_i of d is of the form $f_i = 4\,k+3$ then no solution with $e = -1$ exists, because this would imply $x^2 \equiv -1 \bmod f_i$ but -1 is never a quadratic residue modulo $f_i = 4\,k+3$ by relation 37.8-3 on page 749.

However, all prime factors being $f_i \equiv 1 \bmod 4$ does not guarantee that $e = -1$, the smallest examples are $205 = 5 \cdot 41$, $221 = 13 \cdot 17$, $305 = 5 \cdot 61$, and $377 = 13 \cdot 29$. The sequence of numbers d with no factor of the form $4\,k+3$ so that $x^2 - d\,y^2 = -1$ has no solution is entry A031399 of [214]:

```
4, 8, 16, 20, 25, 32, 34, 40, 52, 64, 68, 80,
100, 104, 116, 128, 136, 146, 148, 160, 164, 169, 178, 194,
200, 205, 208, 212, 221, 232, 244, 256, 260, 272, 289, 292, 296, ...
```

37.13.2 Powering solutions

Let $(x, y) = (P_m, Q_m)$ be a solution of relation 37.13-1b ($e_m = -1$) then $(x, y) := (P_p, Q_p)$ where

$$P_p := P_m^2 + d\,Q_m^2 \tag{37.13-4a}$$
$$Q_p := 2\,P_m\,Q_m \tag{37.13-4b}$$

is a solution of 37.13-1a ($e_p = +1$). If (P_m, Q_m) is the smallest solution with $e := e_m = -1$ then (P_p, Q_p) is the smallest solution with $e_p = e^2 = +1$. If (P_m, Q_m) is the smallest solution with $e = +1$ then (P_p, Q_p) is the second smallest solution with $e_p = +1$.

With our example from figure 37.13-A: $(P_m, Q_m) = (P_4, Q_4) = (182, 25)$ and

$$P_p := 182^2 + 53 \cdot 25^2 = 66249 = P_9 \tag{37.13-5a}$$
$$Q_p := 2 \cdot 182 \cdot 25 = 9100 = Q_9 \tag{37.13-5b}$$

If the period length of the continued fractions equals l then $m = l - 1$ and $p = 2\,l - 1$. Alternative expressions for P_p are

$$P_p = 2\,P_m^2 - e = 2\,d\,Q_m^2 + e \tag{37.13-6}$$

This can be seen as follows: $P_p + 0 = P_p + e - e = (P_m^2 + d\,Q_m) + (P_m^2 - d\,Q_m) - e$ and (second relation) $P_m^2 = d\,Q_m^2 + e$.

Further (P_t, Q_t) where

$$P_t := P_m \left(P_m^2 + 3\,d\,Q_m^2\right) \quad (37.13\text{-}7a)$$
$$Q_t := Q_m \left(3\,P_m^2 + d\,Q_m^2\right) \quad (37.13\text{-}7b)$$

is the second smallest solution with $e_t = e^3 = -1$. With our example $P_t = 24114818$ and $Q_t = 3312425$ (and $t = 14 = 3\,l - 1$). Alternative forms for P_t and Q_t are

$$P_t := P_m \left(4\,P_m^2 - 3\,e\right) \quad (37.13\text{-}8a)$$
$$Q_t := Q_m \left(4\,P_m^2 - e\right) = Q_m \left(4\,d\,Q_m^2 + 3\,e\right) \quad (37.13\text{-}8b)$$

Note that relations 37.13-4a and 37.13-4b are just the numerator and denominator of the second order iteration for $\sqrt{d}$ (relation 28.2-2a on page 544). Relations 37.13-7a and 37.13-7b correspond to the third order iteration (relation 28.2-2b on page 544).

If the pair (x, y) is a solution of relation 37.13-1a, then $(T_n(x), yU_{n-1}(x))$ is also a solution, where T_n and U_n are the Chebyshev polynomials of the first and second kind:

$$T_n^2(x) - d\,(y\,U_{n-1}(x))^2 = T_n^2(x) - d\,y^2\,U_{n-1}(x)^2 = \quad (37.13\text{-}9a)$$
$$T_n^2(x) - (x^2 - 1)\,U_{n-1}(x)^2 = 1 \quad (37.13\text{-}9b)$$

The last equality is relation 34.2-25 on page 652. Similarly, if (x, y) is a solution of 37.13-1b, then $(T_n^+(x), yU_{n-1}^+(x))$ is also a solution, by equation 34.2-30 on page 653.

37.14 Multigrades *

Let a, $a+1$ be any two successive numbers, then $a^0 = (a+1)^0$ (wow!). Let a, $a+1$, $a+2$, $a+3$ any four successive numbers, then $a^1 + (a+3)^1 = (a+1)^1 + (a+2)^1$ (still not really exciting). We write the relations as 'prototypes':

$$0^0 = 1^0 \quad (37.14\text{-}1a)$$
$$0^1 + 3^1 = 1^1 + 2^1 \quad (37.14\text{-}1b)$$

We also have

$$0^2 + 3^2 + 5^2 + 6^2 = 1^2 + 2^2 + 4^2 + 7^2 \quad (37.14\text{-}1c)$$

and

$$0^3 + 3^3 + 5^3 + 6^3 + 9^3 + 10^3 + 12^3 + 15^3 = 1^3 + 2^3 + 4^3 + 7^3 + 8^3 + 11^3 + 13^3 + 14^3 \quad (37.14\text{-}1d)$$

$$0^4 + 3^4 + 5^4 + 6^4 + 9^4 + 10^4 + 12^4 + 15^4 + 17^4 + 18^4 + 20^4 + 23^4 + 24^4 + 27^4 + 29^4 + 30^4 = \quad (37.14\text{-}1e)$$
$$= 1^4 + 2^4 + 4^4 + 7^4 + 8^4 + 11^4 + 13^4 + 14^4 + 16^4 + 19^4 + 21^4 + 22^4 + 25^4 + 26^4 + 28^4 + 31^4$$

In general, for a fixed exponent $k > 0$, let $s = 2^{k+1}$ and partition the set $\{0, 1, 2, 3, \ldots, s-1\}$ into two sets S_0, S_1 so that S_0 contains all elements j $(0 \le j < s)$ with parity zero, S_1 the elements with parity one. Then for $d \in \mathbb{C}$,

$$\sum_{a \in S_0} (a+d)^k = \sum_{b \in S_1} (b+d)^k \quad (37.14\text{-}2)$$

An example, for $k = 2$ we have $S_0 = \{0, 3, 5, 6\}$, $S_1 = \{1, 2, 4, 7\}$ and so

$$(0+d)^2 + (3+d)^2 + (5+d)^2 + (6+d)^2 = (1+d)^2 + (2+d)^2 + (4+d)^2 + (7+d)^2 \quad (37.14\text{-}3)$$

Compare to relation 37.14-1c. The prototypes for the exponent k are also valid for all positive integer exponents less than k and the exponent zero.

The equations given are special solutions of the so-called *multigrades* or *Prouhet-Tarry-Escott* problem: Find two sets A and B such that for a given k

$$\sum_{a\in A} a^j \quad = \quad \sum_{b\in B} b^j \qquad j = 0, 1, 2, \ldots, k \tag{37.14-4}$$

Both sets have the same number n of elements. This is a (k, n)-multigrade. We found solutions of $(k, 2^k)$-multigrades with the special property that the union of both sets contains 2^k successive numbers. The shift invariance holds again, it is a property of all multigrade solutions.

Multiplying the symbolic entries in a prototype gives a valid prototype which again is shift invariant. This means that, for example, relation 37.14-1c can be interpreted as

$$\begin{aligned}(0\,m+d)^k + (3\,m+d)^k + (5\,m+d)^k + (6\,m+d)^k \;&= \\ = \; (1\,m+d)^k + (2\,m+d)^k + (4\,m+d)^k + (7\,m+d)^k&\end{aligned} \tag{37.14-5}$$

for $k \in \{0, 1, 2\}$, $d \in \mathbb{C}$ and $m \in \mathbb{C}$.

Adding and subtraction shifted versions of the prototypes gives more prototypes. We subtract from relation 37.14-1c the shifted by one version with swapped sides:

$$\begin{aligned} +\left(0^2 + 3^2 + 5^2 + 6^2 \right) &= 1^2 + 2^2 + 4^2 + 7^2\big) \\ -\left(2^2 + 3^2 + 5^2 + 8^2 \right) &= 1^2 + 4^2 + 6^2 + 7^2\big) \end{aligned}$$

Giving $2 \cdot 2^2 + 8^2 = 0^2 + 2 \cdot 6^2$. We can divide all symbolic entries by two and get:

$$2 \cdot 1^2 + 4^2 \quad = \quad 0^2 + 2 \cdot 3^2 \tag{37.14-6}$$

which really is

$$2 \cdot (1\,m+d)^k + (4\,m+d)^k \quad = \quad (0\,m+d)^k + 2 \cdot (3\,m+d)^k \tag{37.14-7}$$

for $k \in \{0, 1, 2\}$, $d \in \mathbb{C}$ and $m \in \mathbb{C}$.

37.15 Properties of the convergents of $\sqrt{2}$ *

We give some number theoretical properties of the convergents of the continued fraction of $\sqrt{2}$.

Define $P_0 := 1$, $P_1 := 1$, $P_n := 2\,P_{n-1} + P_{n-2}$, $Q_0 := 0$, $Q_1 := 1$ and $Q_n := 2\,Q_{n-1} + Q_{n-2}$. Then P_{n+1}/Q_{n+1} is the n-th convergent of $\sqrt{2}$:

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_n:	1	3	7	17	41	99	239	577	1393	3363	8119	19601	47321	114243	275807
Q_n:	1	2	5	12	29	70	169	408	985	2378	5741	13860	33461	80782	195025

37.15.1 Pythagorean triples with difference one

Pythagorean triples are solutions of the Diophantine equation $a^2 + b^2 = c^2$. All primitive (a, b, c coprime) solutions can be obtained by choosing coprime x and y and setting $a := 2\,x\,y$, $b := x^2 - y^2$, and $c := x^2 + y^2$. There are Pythagorean triples where a and b differ by one, the first few are given in figure 37.15-A. The list can be obtained using $a_n = (P_{2\,n-1} - 1)/2$, $b_n = a_n + 1$ and $c_n = Q_{2\,n-1}$:

```
     n :  a^2 + b^2 = c^2
     1 :  0^2 + 1^2 = 1^2
     2 :  3^2 + 4^2 = 5^2
     3 :  20^2 + 21^2 = 29^2
     4 :  119^2 + 120^2 = 169^2
     5 :  696^2 + 697^2 = 985^2
     6 :  4059^2 + 4060^2 = 5741^2
     7 :  23660^2 + 23661^2 = 33461^2
     8 :  137903^2 + 137904^2 = 195025^2
     9 :  803760^2 + 803761^2 = 1136689^2
    10 :  4684659^2 + 4684660^2 = 6625109^2
    11 :  27304196^2 + 27304197^2 = 38613965^2
    12 :  159140519^2 + 159140520^2 = 225058681^2
    13 :  927538920^2 + 927538921^2 = 1311738121^2
    14 :  5406093003^2 + 5406093004^2 = 7645370045^2
    15 :  31509019100^2 + 31509019101^2 = 44560482149^2
```

Figure 37.15-A: Solutions of $a^2 + b^2 = c^2$ with $b - a = 1$.

```
P(n)=( ([1,1]*[0,1;1,2]^n)[1] );
Q(n)=( ([0,1]*[0,1;1,2]^n)[1] );
pyth1(k)=
{ /* return [a,b,c] where a^2+b^2=c^2 and b=a+1 */
    local(p,q);
    p = (P(2*k-1)-1)/2;
    q = Q(2*k-1);
    return( [p, p+1, q] );
}
```

Or with the equivalent

```
pyth2(k)=
{ /* return [a,b,c] where a^2+b^2=c^2 and b=a+1 */
    local(x,y);
    x = P(k);  y=Q(k);
    return( [x^2-y^2, 2*x*y, x^2+y^2] );
}
```

37.15.2 Solutions of $x^4 \pm 4 \cdot z^4 = c^2 + 1$ and $x^4 - y^4 = c^2\, d$

```
     n :  x^4 + 4*z^4 = c^2 + 1
     1 :  1^4 + 4*1^4 = 2^2 + 1
     2 :  3^4 + 4*2^4 = 12^2 + 1
     3 :  7^4 + 4*5^4 = 70^2 + 1
     4 :  17^4 + 4*12^4 = 408^2 + 1
     5 :  41^4 + 4*29^4 = 2378^2 + 1
     6 :  99^4 + 4*70^4 = 13860^2 + 1
     7 :  239^4 + 4*169^4 = 80782^2 + 1
     8 :  577^4 + 4*408^4 = 470832^2 + 1
     9 :  1393^4 + 4*985^4 = 2744210^2 + 1
    10 :  3363^4 + 4*2378^4 = 15994428^2 + 1
    11 :  8119^4 + 4*5741^4 = 93222358^2 + 1
    12 :  19601^4 + 4*13860^4 = 543339720^2 + 1
    13 :  47321^4 + 4*33461^4 = 3166815962^2 + 1
    14 :  114243^4 + 4*80782^4 = 18457556052^2 + 1
    15 :  275807^4 + 4*195025^4 = 107578520350^2 + 1
```

Figure 37.15-B: Solutions of $x^4 + 4z^4 = c^2 + 1$.

If set $a_n = (P_{2\,n} - 1)/2$, $b_n = a_n + 1$ and $c_n = Q_{2\,n}$ then we get 'almost' Pythagorean triples:

```
    n :  a^2 + b^2 = c^2 + 1
    1 :  1^2 + 2^2 = 2^2 + 1
    2 :  8^2 + 9^2 = 12^2 + 1          // b=3^2  a=2*2^2
    3 :  49^2 + 50^2 = 70^2 + 1        // a=7^2  b=2*7^2
    4 :  288^2 + 289^2 = 408^2 + 1     // b=17^2 a=2*12^2
    5 :  1681^2 + 1682^2 = 2378^2 + 1  // a=41^2 b=2*41^2
```

As the comments suggest, all equations are of the form $x^4 + 4 \cdot z^4 = c^2 + 1$. We obtain solutions via the simple routine:

```
deq44(k)=
{ /* return [x,y,c] where x^4+4*z^4=c^2+1 */
```

```
     n :  x^4 - 4*z^4 = c^2 + 1
     1 :  3^4 - 4*2^4 = 4^2 + 1
     2 :  17^4 - 4*12^4 = 24^2 + 1
     3 :  99^4 - 4*70^4 = 140^2 + 1
     4 :  577^4 - 4*408^4 = 816^2 + 1
     5 :  3363^4 - 4*2378^4 = 4756^2 + 1
     6 :  19601^4 - 4*13860^4 = 27720^2 + 1
     7 :  114243^4 - 4*80782^4 = 161564^2 + 1
     8 :  665857^4 - 4*470832^4 = 941664^2 + 1
     9 :  3880899^4 - 4*2744210^4 = 5488420^2 + 1
    10 :  22619537^4 - 4*15994428^4 = 31988856^2 + 1
    11 :  131836323^4 - 4*93222358^4 = 186444716^2 + 1
    12 :  768398401^4 - 4*543339720^4 = 1086679440^2 + 1
    13 :  4478554083^4 - 4*3166815962^4 = 6333631924^2 + 1
    14 :  26102926097^4 - 4*18457556052^4 = 36915112104^2 + 1
    15 :  152139002499^4 - 4*107578520350^4 = 215157040700^2 + 1
```

Figure 37.15-C: Solutions of $x^4 - 4z^4 = c^2 + 1$.

```
    local(x,z,c);
    x = P(k);  z = Q(k);  c=2*x*z;
    return( [x, z, c] );
}
```

The output is shown in figure 37.15-B. There are no solutions to the equation $x^4 + 4\,z^4 = c^2$ [31] so this is as close as one can get. Also, $x^4 - 4\,z^4 = c^2$ has no solutions but solutions of $x^4 - 4\,z^4 = c^2 + 1$ are obtained via

```
deq44m(k)=
{ /* return [x,y,c] where x^4-4*z^4=c^2+1 */
    local(x,z,c);
    x = P(2*k);  z = Q(2*k);  c=2*z;
    return( [x, z, c] );
}
```

The first few solutions shown in figure 37.15-C.

```
     n :  x^4 - y^4 = c^2 * d
     1 :  1^4 - 0^4 = 1^2 * 1
     2 :  4^4 - 3^4 = 5^2 * 7
     3 :  21^4 - 20^4 = 29^2 * 41
     4 :  120^4 - 119^4 = 169^2 * 239
     5 :  697^4 - 696^4 = 985^2 * 1393
     6 :  4060^4 - 4059^4 = 5741^2 * 8119
     7 :  23661^4 - 23660^4 = 33461^2 * 47321
     8 :  137904^4 - 137903^4 = 195025^2 * 275807
     9 :  803761^4 - 803760^4 = 1136689^2 * 1607521
    10 :  4684660^4 - 4684659^4 = 6625109^2 * 9369319
    11 :  27304197^4 - 27304196^4 = 38613965^2 * 54608393
    12 :  159140520^4 - 159140519^4 = 225058681^2 * 318281039
    13 :  927538921^4 - 927538920^4 = 1311738121^2 * 1855077841
    14 :  5406093004^4 - 5406093003^4 = 7645370045^2 * 10812186007
    15 :  31509019101^4 - 31509019100^4 = 44560482149^2 * 63018038201
```

Figure 37.15-D: Solutions of $x^4 - y^4 = c^2\,d$.

Finally, solutions of the equation $x^4 - y^4 = c^2 \cdot d$ (where $y = x - 1$) can be computed by

```
deq4421(k)=
{ /* return [x,y,c,d] where x^4-y^4=c^2*d */
    local(x,y,c,d);
    y = (P(2*k-1)-1)/2;
    x = y+1;
    c = Q(2*k-1);  d = P(2*k-1);
    return( [x, y, c, d] );
}
```

The first few are shown in figure 37.15-D. Note that equation 4 is $120^4 - 119^4 = 13^4\,239$.

37.15.3 Triangular square numbers

A triangular number is a number of the form $n\,(n+1)/2$. There are triangular numbers that are also squares. Let z_n be the n-th triangular number that is a square. Then

$$z_n \;=\; (P_n\,Q_n)^2 \;=\; 1,\ 36,\ 1225,\ 41616,\ 1413721,\ \ldots \tag{37.15-1}$$

We can generate the z_n via

```
P(n)=(([1,1]*[0,1;1,2]^n)[1]);
Q(n)=(([0,1]*[0,1;1,2]^n)[1]);
zn(n)=(P(k)*Q(k))^2
```

The sequence of values z_n is entry A001110 in [214]. A recurrence for z_n is given by $z_0 = 1$, $z_1 = 36$, $z_n = 34\,z_{n-1} - z_{n-2} + 2$, a closed form expression is $z_n = \frac{1}{32}\left((3+\sqrt{8})^{2n} + (3-\sqrt{8})^{2n} - 2\right)$. A simpler recurrence gives the square roots of z_n: define $w_0 = 1$, $w_1 = 6$, and $w_n = 6\,w_{n-1} - w_{n-2}$. Then $w_n^2 = z_n$, the w_n-th square number equals z_n. The numbers w_n can be obtained as:

```
? wn(n)=return(([1,6]*[0,-1;1,6]^n)[1]);
? for(k=0,10,print1(" ",wn(k)))
 1 6 35 204 1189 6930 40391 235416 1372105 7997214 46611179
? for(k=0,10,print1(" ",wn(k)^2))
 1 36 1225 41616 1413721 48024900 1631432881 55420693056 1882672131025
```

Let $s_0 = 1$, $s_1 = 8$, $s_2 = 49$ and $s_n = 7\,s_{n-1} - 7\,s_{n-2} + s_{n-3}$. Then $s_n\,(s_n+1)/2 = z_n$, and the s_n-th triangular number equals z_n. The numbers s_n can be obtained as:

```
? sn(n)=return(([1,8,49]*[0,0,1;1,0,-7;0,1,7]^n)[1]);
? for(k=0,10,print1(" ",sn(k)))
 1 8 49 288 1681 9800 57121 332928 1940449 11309768 65918161
? tria(n)=n*(n+1)/2;
? for(k=0,10,print1(" ",tria(sn(k))))
 1 36 1225 41616 1413721 48024900 1631432881 55420693056 1882672131025
```

37.15.4 Solutions of the equations $s^3 + d^2 = w^4$

```
    n: s^3 + d^2 = w^4
    0: 1^3 + 0^2 = 1^4
    1: 8^3 + 28^2 = 6^4
    2: 49^3 + 1176^2 = 35^4
    3: 288^3 + 41328^2 = 204^4
    4: 1681^3 + 1412040^2 = 1189^4
    5: 9800^3 + 48015100^2 = 6930^4
    6: 57121^3 + 1631375760^2 = 40391^4
    7: 332928^3 + 55420360128^2 = 235416^4
    8: 1940449^3 + 1882670190576^2 = 1372105^4
    9: 11309768^3 + 63955420452028^2 = 7997214^4
   10: 65918161^3 + 2172601941851880^2 = 46611179^4
   11: 384199200^3 + 73804512448220400^2 = 271669860^4
   12: 2239277041^3 + 2507180832055219320^2 = 1583407981^4
   13: 13051463048^3 + 8517034384012899362 8^2 = 9228778026^4
   14: 76069501249^3 + 2893284510097771529376^2 = 53789260175^4
   15: 4433665544448^3 + 98286503001614049040128^2 = 313506783024^4
```

Figure 37.15-E: Solutions of $s^3 + d^2 = w^4$.

The sum of cubes up to n^3 is the square of the n-th triangular number:

$$\sum_{k=1}^{n} k^3 \;=\; \left(\frac{n\,(n+1)}{2}\right)^2 \tag{37.15-2}$$

So the difference of the squares of two consecutive triangular numbers is a cube:

$$\left(\frac{n\,(n+1)}{2}\right)^2 - \left(\frac{(n-1)\,n}{2}\right)^2 \;=\; n^3 \tag{37.15-3}$$

We write $T(n)$ for the n-th triangular number. Now, we have seen that $T(s_n)$, the s_n-th triangular number, is a square: $T(s_n) = w_n^2$. Squaring gives $T(s_n)^2 = w_n^4$. Subtracting $d_n := T(s_n - 1)$ from both sides gives $s_n^3 + d_n^2 = w_n^4$. That is, the triples $[s_n, d_n, w_n]$ are a solutions of the Diophantine equation

$$s^3 + d^2 \quad = \quad w^4 \tag{37.15-4}$$

which is a somewhat surprising observation. For the computation of d_n we use $d_n = w_n^2 - s_n = z_n - s_n$:

```
deq(n)=
{ /* Return [s,d,w] so that  s^3 + d^2 = w^4 */
    local(s, t, w, d);
    s = sn(n);
    w = wn(n);
    d = w^2-s;  \\ == zn(n)-s;
    return( [s, d, w] );
}
```

The first few solutions of equation 37.15-4 of the given form are shown in figure 37.15-E. The equations with odd index are of the form $8\,u^6+d^2 = w^4$. The equations with even index are of the form $u^6+d^2 = w^4$, corresponding to Pythagorean triples giving triangles with hypothenuse a square and one side a third power. For example (equation 2): $(7^3)^2 + (1176)^2 = (35^2)^2$ which is $7^2 + 24^2 = 5^4$ multiplied by 7^4.

The triangular square numbers were determined by Euler. That triangular square numbers yield solutions to $s^3 + d^2 = w^4$ was observed 1912 by H. B. Mathieu.

We note that there exist solutions that are not of the given form. A simple search reveals some of them, we use the routine

```
N=200;
{ for (s=1, N,  s3 = s^3;
    for (w=1, N,  w4 = w^4;
        t=w4-s3;
        if ( issquare(t) && (t!=0),
            d=sqrtint(t);
            g=gcd(s,gcd(d,w));
            print(s, "^3 + ", d, "^2 = ", w, "^4   [", g, "]");
        );
 ); ); }
```

It finds all solutions where $0 < s \le 200$ and $0 < w \le 200$:

```
   s^3 + d^2 = w^4    [gcd(s,d,w)]
    8^3 + 28^2 = 6^4   [2]
    18^3 + 27^2 = 9^4   [9]
    23^3 + 6083^2 = 78^4   [1]
    36^3 + 63^2 = 15^4   [3]
    49^3 + 1176^2 = 35^4   [7]
    108^3 + 648^2 = 36^4   [36]
    108^3 + 3807^2 = 63^4   [9]
    126^3 + 2925^2 = 57^4   [3]
    128^3 + 1792^2 = 48^4   [16]
    135^3 + 4941^2 = 72^4   [9]
    136^3 + 4785^2 = 71^4   [1]
    143^3 + 433^2 = 42^4   [1]
```

In his amazing note [84] Cohen gives for the Diophantine equation

$$a^4 + b^3 \quad = \quad c^2 \tag{37.15-5}$$

the parametric solution

$$a \quad = \quad 6 \cdot s \cdot t \cdot (4 \cdot s^4 + 3 \cdot t^4) \tag{37.15-6a}$$

$$b \quad = \quad 16 \cdot s^8 - 168 \cdot s^4 \cdot t^4 + 9 \cdot t^8 \tag{37.15-6b}$$

$$c \quad = \quad 64 \cdot s^{12} + 1584 \cdot s^8 \cdot t^4 - 1188 \cdot s^4 \cdot t^8 - 27 \cdot t^{12} \tag{37.15-6c}$$

where $s, t \in \mathbb{Z}$.

Regarding triangular numbers, we note a nice generalization of the relation

$$\left(\sum_{k=1}^{n} k\right)^2 \;=\; \sum_{k=1}^{n} k^3 \tag{37.15-7}$$

which is a consequence of

$$\sum_{k=1}^{n} k \;=\; \frac{n\,(n+1)}{2} \;=\; T(n) \tag{37.15-8a}$$

$$\sum_{k=1}^{n} k^3 \;=\; \left(\frac{n\,(n+1)}{2}\right)^2 \;=\; T(n)^2 \tag{37.15-8b}$$

Indeed,

$$\left(\sum_{d\backslash N} \nu(d)\right)^2 \;=\; \sum_{d\backslash N} \nu(d)^3 \tag{37.15-9}$$

where $\nu(d)$ counts the number of divisors of d. Relation 37.15-7 is the special case where $N = p^{n-1}$. The divisors of N are 1, p, p^2, ..., p^{n-1} whose numbers of divisors in turn are 1, 2, 3, ..., n.

Relation 37.15-7 can easily be proven for the case that N is a product of n distinct primes: $n = p_1 \cdot p_2 \cdot \cdot p_n$. There are $\binom{n}{j}$ divisors with exactly j prime factors, each having 2^j divisors. Thereby the sum on the left hand side of relation 37.15-9 is equal to

$$\left(\sum_{j=0}^{n} \binom{n}{j} 2^j\right)^2 \;=\; \left((1+2)^n\right)^2 \;=\; \sum_{j=0}^{n} \binom{n}{j} 2^{3j} \;=\; \sum_{j=0}^{n} \binom{n}{j} \left(2^j\right)^3 \tag{37.15-10a}$$

The quantity is always an even power of 3:

$$\;=\; \sum_{j=0}^{n} \binom{n}{j} 8^j \;=\; (1+8)^n \;=\; 9^n \tag{37.15-10b}$$

This holds only if N factors into mutually distinct primes.

We close with two relations taken from the marvelous book of A. H. Beiler [31, p.161-162]:

$$2\left(\sum_{k=1}^{n} T(k)\right)^2 \;=\; \sum_{k=1}^{n} k^5 + \sum_{k=1}^{n} k^7 \tag{37.15-11}$$

$$3\left(\sum_{k=1}^{n} T(k)\right)^3 \;=\; \sum_{k=1}^{n} T(k)^3 + 2\sum_{k=1}^{n} T(k)^4 \tag{37.15-12}$$

37.16 Multiplication of hypercomplex numbers *

An n-dimensional vector space (over a field) together with component-wise addition and a *multiplication table* that defines the product of any two (vector) components defines an *algebra*.

The product of two elements $x = \sum_k \alpha_k\, e_k$ and $y = \sum_j \beta_j\, e_j$ of the algebra is defined as

$$x \cdot y \;=\; \sum_{k,j=0}^{n-1} \left[(\alpha_k \cdot \beta_j)\, e_k\, e_j\right] \tag{37.16-1}$$

The quantities $e_k\, e_j$ are given in the multiplication table of the algebra. These can be arbitrary elements of the algebra, that is, linear combinations of the components. For example, a 2-dimensional algebra over the reals could have the following multiplication table:

```
           e0                    e1
e0:  (5*e1 + 3*e0)    (239*e0 + 3.1415*e1)
e1:      (0)          (17*e1 + 2.71828*e0)
```

Note that there is no neutral element of multiplication ('one'). Further, the algebra has *zero divisors*: the equation $x \cdot y = 0$ has a solution where neither element is zero, namely $x = e_1$, and $y = e_0$. As almost all randomly defined algebras, it is completely uninteresting.

In what follows we will only consider algebras over the reals where the product of two components equals ± 1 times another component. For example, the *complex* numbers are a two-dimensional algebra (over the real numbers) with the multiplication table

```
         e0   e1

e0:  +e0  +e1
e1:  +e1  -e0
```

Which is, using the symbols '`1`' and '`i`',

```
         1   i

1:   +1  +i
i:   +i  -1
```

We will denote the components of an n-dimensional algebra by the numbers $0, 1, \ldots, n-1$. The multiplication table for the complex numbers would thus be written as

```
         0   1

0:   +0  +1
1:   +1  -1
```

37.16.1 The Cayley-Dickson construction

The *Cayley-Dickson construction* recursively defines multiplication tables for certain algebras where the dimension is a power of two. Let a, A, b and B be elements of a 2^{n-1}-dimensional algebra U. Define the multiplication rule for an algebra V (of dimension 2^n) written as pairs of elements of U via

$$(a,\, b) \cdot (A,\, B) \quad := \quad (a \cdot A - B \cdot b^*,\; a^* \cdot B + A \cdot b) \tag{37.16-2}$$

where the *conjugate* C^* of an element $C = (a, b)$ is defined as

$$(a,\, b)^* \;:=\; (a^*,\, -b) \tag{37.16-3}$$

and the conjugate of a real number a equals a (unmodified). The construction leads to multiplications tables where the product of two units equals plus or minus one other unit only: $e_i \cdot e_j = \pm e_k \;\forall i, j$.

Figure 37.16-A gives the multiplication table for a 16-dimensional algebra, the so-called *sedenions*. The upper left (8×8) quarter gives the multiplication rule for the *octonions* (or *Cayley numbers*), the upper left 4×4 subtable gives the rule for the *quaternions* and the upper left 2×2 subtable corresponds to the complex numbers. Note that multiplication is in general neither commutative (only up to dimension 2) nor associative (only up to dimension 4).

The 2^n-dimensional algebras are (for $n > 1$) referred to as *hypercomplex numbers*. There is no generally accepted naming scheme for the algebras beyond dimension 16. We will use the names '2^n-ions'.

This form (relation 37.16-2) of the construction is given in [22], an alternative form is used in [105]:

$$(a,\, b) \cdot (A,\, B) \quad := \quad (a \cdot A - B^* \cdot b,\; b \cdot A^* + B \cdot a) \tag{37.16-4}$$

It leads to a table that is the transposed of figure 37.16-A.

```
          0   1   2   3   4   5   6   7    8   9   a   b   c   d   e   f

    0:   +0  +1  +2  +3  +4  +5  +6  +7   +8  +9  +a  +b  +c  +d  +e  +f
    1:   +1  -0  -3  +2  -5  +4  +7  -6   -9  +8  +b  -a  +d  -c  -f  +e
    2:   +2  +3  -0  -1  -6  -7  +4  +5   -a  -b  +8  +9  +e  +f  -c  -d
    3:   +3  -2  +1  -0  -7  +6  -5  +4   -b  +a  -9  +8  +f  -e  +d  -c
    4:   +4  +5  +6  +7  -0  -1  -2  -3   -c  -d  -e  -f  +8  +9  +a  +b
    5:   +5  -4  +7  -6  +1  -0  +3  -2   -d  +c  -f  +e  -9  +8  -b  +a
    6:   +6  -7  -4  +5  +2  -3  -0  +1   -e  +f  +c  -d  -a  +b  +8  -9
    7:   +7  +6  -5  -4  +3  +2  -1  -0   -f  -e  +d  +c  -b  -a  +9  +8

    8:   +8  +9  +a  +b  +c  +d  +e  +f   -0  -1  -2  -3  -4  -5  -6  -7
    9:   +9  -8  +b  -a  +d  -c  -f  +e   +1  -0  +3  -2  +5  -4  -7  +6
    a:   +a  -b  -8  +9  +e  +f  -c  -d   +2  -3  -0  +1  +6  +7  -4  -5
    b:   +b  +a  -9  -8  +f  -e  +d  -c   +3  +2  -1  -0  +7  -6  +5  -4
    c:   +c  -d  -e  -f  -8  +9  +a  +b   +4  -5  -6  -7  -0  +1  +2  +3
    d:   +d  +c  -f  +e  -9  -8  -b  +a   +5  +4  -7  +6  -1  -0  -3  +2
    e:   +e  +f  +c  -d  -a  +b  -8  -9   +6  +7  +4  -5  -2  +3  -0  -1
    f:   +f  -e  +d  +c  -b  -a  +9  -8   +7  -6  +5  +4  -3  -2  +1  -0
```

Figure 37.16-A: Multiplication table for the sedenions. The entry in row R, column C gives the product $R \cdot C$ of the components R and C (hexadecimal notation).

```
              0 1 2 3 4 5 6 7  8 9 a b c d e f

         0:   + + + + + + + +  + + + + + + + +
         1:   + - - + - + + -  - + + - + - - +
         2:   + + - - - - + +  - - + + + + - -
         3:   + - + - - + - +  - + - + + - + -
         4:   + + + + - - - -  - - - - + + + +
         5:   + - + - + - + -  - + - + - + - +
         6:   + - - + + - - +  - + + - - + + -
         7:   + + - - + + - -  - - + + - - + +

         8:   + + + + + + + +  - - - - - - - -
         9:   + - + - + - - +  + - + - + - - +
         a:   + - - + + + - -  + - - + + + - -
         b:   + + - - + - + -  + + - - + - + -
         c:   + - - - - + + +  + - - - - + + +
         d:   + + - + - - - +  + + - + - - - +
         e:   + + + - - + - -  + + + - - + - -
         f:   + - + + - - + -  + - + + - - + -
```

Figure 37.16-B: Signs in the multiplication table for sedenions.

By construction, $e_0^2 = e_0$, $e_k^2 = -e_0$, $e_0\, e_k = e_k\, e_0 = e_k$, and $e_k\, e_j = -e_j\, e_k$ whenever both of k and j are nonzero (and $k \neq j$). Further,

$$e_k\, e_j \quad = \quad \pm e_x \quad \text{where} \quad x = k \,\text{XOR}\, j \tag{37.16-5}$$

where the sign is to be determined. Figure 37.16-B shows the pattern of the signs of the sedenion algebra. The lower left quarter is the transposed of the upper left quarter, so is the lower right quarter, except for its top row. The upper right quarter is (except for its first row) the negated upper left quarter. These observations, together with the partial antisymmetry can be cast into an algorithm to compute the signs [FXT: arith/cayley-dickson-demo.cc]:

```
int CD_sign_rec(ulong r, ulong c, ulong n)
// Signs in the multiplication table for the
//   algebra of n-ions (where n is a power of two)
//   that is obtained by the Cayley-Dickson construction:
// If component r is multiplied with component c then the
//   result is CD_sign_rec(r,c,n) * (r XOR c).
{
    if ( (r==0) || (c==0) )  return +1;
    if ( c>=r )
    {
        if ( c>r )   return  -CD_sign_rec(c, r, n);
        else  return -1;  // r==c
    }
    // here r>c (triangle below diagonal)
    ulong h = n>>1;
    if ( c>=h )  // right
    {
```

```
        // (upper right unreached)
        return   CD_sign_rec(c-h, r-h, h); // lower right
    }
    else  // left
    {
        if ( r>=h )  return   CD_sign_rec(c, r-h, h); // lower left
        else         return   CD_sign_rec(r, c, h); // upper left
    }
}
```

The function uses at most $2 \cdot \log_2(n)$ steps. Note that the second row in the table is (the signed version of) the *Thue-Morse sequence*, see section 1.15.1 on page 37. The matrix filled with entries ± 1 according to figure 37.16-B is a *Hadamard matrix*, see section 18 on page 347. The sequence of signs, read by antidiagonals, and setting $0 := +$ and $1 := -$, is entry A118685 of [214].

An iterative version of the function is [FXT: arith/cayley-dickson-demo.cc]:

```
inline void cp2(ulong a, ulong b, ulong &u, ulong &v)  { u=a; v=b; }
//
inline int CD_sign_it(ulong r, ulong c, ulong n)
{
    int s = +1;

start:
    if ( (r==0) || (c==0) )  return s;
    if ( c==r )  return -s;
    if ( c>r )   { swap2(r,c); s=-s; }
    n >>= 1;
    if ( c>=n )  cp2(c-n, r-n, r, c);
    else if ( r>=n )  cp2(c, r-n, r, c);

    goto start;
}
```

The computation of all $2^{26} = 67,108,864$ signs in the multiplication table for the '2^{13}-ions' takes less than 6 seconds (about 8 seconds with the recursive routine).

```
          0  1  2  3  4  5  6  7

    0:   +0 +1 +2 +3 +4 +5 +6 +7          + + + + + + + +
    1:   +1 -0 +6 +4 -3 +7 -2 -5          + - + + - + - -
    2:   +2 -6 -0 +7 +5 -4 +1 -3          + - - + + - + -
    3:   +3 -4 -7 -0 +1 +6 -5 +2          + - - - + + - +
    4:   +4 +3 -5 -1 -0 +2 +7 -6          + + - - - + + -
    5:   +5 -7 +4 -6 -2 -0 +3 +1          + - + - - - + +
    6:   +6 +2 -1 +5 -7 -3 -0 +4          + + - + - - - +
    7:   +7 +5 +3 -2 +6 -1 -4 -0          + + + - + - - -
```

Figure 37.16-C: Alternative multiplication table for the octonions (left) and its sign pattern (right).

An alternative multiplication table for the octonions is given in figure 37.16-C. Its sign pattern is the 8×8 Hadamard matrix shown in figure 18.1-A on page 348. Properties of this representation and the relation to shift register sequences are described in [99].

37.16.2 Fast multiplication of quaternions

Quaternion multiplication can be achieved in eight real multiplication using the dyadic convolution (see section 22.7 on page 445): the scheme in figure 37.16-D suggests to use the dyadic convolution with bucket zero negated as a starting point which costs 4 multiplications. Some entries have to be corrected then which costs four more multiplications.

```
// f[] == [ re1, i1, j1, k1 ]
// g[] == [ re2, i2, j2, k2 ]
c0 := f[0] * g[0]
c1 := f[3] * g[2]
c2 := f[1] * g[3]
c3 := f[2] * g[1]

// length-4 dyadic convolution:
walsh(f[])
```

```
    +--  0   1   2   3      +--  0   1   2   3        +--  0   1   2   3
    |                      |                        |
    0:   0   1   2   3      0:  -0   1   2   3      1 0:   0*  1   2   3
    1:   1   0   3   2      1:   1  -0   3   2      i 1:   1  -0   3  -2*
    2:   2   3   0   1      2:   2   3  -0   1      j 2:   2  -3* -0   1
    3:   3   2   1   0      3:   3   2   1  -0      k 3:   3   2  -1* -0
```

Figure 37.16-D: Scheme for the length-4 dyadic convolution (left), same with bucket zero negated (middle) and the multiplication table for the units of the quaternions (right). The asterisks mark those entries where the sign is different from the scheme in the middle.

```
-0  1  2  3   4  5  6  7      0  1  2  3   4  5  6  7      #0  1  2  3   4  5  6  7
 1 -0  3  2   5  4  7  6      1 -0  3 -2   5 -4 -7  6       1  0  3 #2   5 #4 #7  6
 2  3 -0  1   6  7  4  5      2 -3 -0  1   6  7 -4 -5       2 #3  0  1   6  7 #4 #5
 3  2  1 -0   7  6  5  4      3  2 -1 -0   7 -6  5 -4       3  2 #1  0   7 #6  5 #4

 4  5  6  7  -0  1  2  3      4 -5 -6 -7  -0  1  2  3       4 #5 #6 #7   0  1  2  3
 5  4  7  6   1 -0  3  2      5  4 -7  6  -1 -0 -3  2       5  4 #7  6  #1  0 #3  2
 6  7  4  5   2  3 -0  1      6  7  4 -5  -2  3 -0 -1       6  7  4 #5  #2  3  0 #1
 7  6  5  4   3  2  1 -0      7 -6  5  4  -3 -2  1 -0       7 #6  5  4  #3 #2  1  0
```

Figure 37.16-E: Scheme for the length-8 dyadic convolution with bucket zero negated (left) and multiplication table for the octonions (middle, taken from [105]). There are 22 places where the signs differ (right, marked with '#'). This leads to an algorithm involving $8 + 22 = 30$ multiplications.

```
walsh(g[])
for i:=0 to 3  g[i] := (f[i] * g[i])
walsh(g[])

// normalization and correction:
g[0] :=   2 * c0 - g[0] / 4
g[1] := - 2 * c1 + g[1] / 4
g[2] := - 2 * c2 + g[2] / 4
g[3] := - 2 * c3 + g[3] / 4
```

The algorithm is taken from [138] which also gives a second variant.

The complex multiplication by three real multiplications (relation 37.12-2c on page 772) corresponds to one length-2 Walsh dyadic convolution and the correction for the product of the imaginary units:

```
// f[] == [ re1, im1 ]
// g[] == [ re2, im2 ]
c0 := f[1] * g[1]  // == im1 * im2
// length-2 dyadic convolution:
{ f[0], f[1] } := { f[0]+f[1], f[0]-f[1] }
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
g[0] := f[0] * g[0]
g[1] := f[1] * g[1]
{ g[0], g[1] } := { g[0]+g[1], g[0]-g[1] }
// normalization:
f[0] := f[0] / 2
g[0] := g[0] / 2
// correction:
g[0] := -2 * c0 + g[0]
// here:  g[] == [ re1 * re2 - im1 * im2,  re1 * im2 + im1 * re2 ]
```

For complex numbers of high precision multiplication is asymptotically equivalent to two real multiplications as one FFT based (complex linear) convolution can be used for the computation. Similarly, high precision quaternion multiplication is as expensive as four real multiplications. Figure 37.16-E shows an equivalent construction for the octonions leading to an algorithm with 30 multiplications.

Chapter 38

Binary polynomials

This chapter introduces binary polynomials and the arithmetic operations on them. We further describe tests for irreducibility and primitivity. Finally, a method for the factorization of binary polynomials is given. Many of the algorithms shown can easily be implemented in hardware. The arithmetic operations of binary polynomials are the underlying methods for computations in binary finite fields which are treated in chapter 40. Another important application, the linear feedback shift registers, are described in chapter 39

A polynomial with coefficients in the field GF(2) = $\mathbb{Z}/2\mathbb{Z}$ (that is, 'coefficients modulo 2') is called a *binary polynomial*. The operations proceed as for usual polynomials except that the coefficients have to be reduced modulo two.

To represent a binary polynomial in a binary computer one uses words where the bits are set at the positions where the polynomial coefficients are one. We stick to the convention that the constant term goes to the least significant bit. It turns out that the arithmetic operations can be implemented quite easily in an efficient manner.

38.1 The basic arithmetical operations

Addition of binary polynomials is the XOR operation. Subtraction is the very same operation.

Multiplication of a binary polynomial by its independent variable x is simply a shift to the left.

The following routines are given in [FXT: bpol/bitpol-arith.h].

38.1.1 Multiplication and squaring

Multiplication of two polynomials A and B is identical to the usual (binary algorithm for) multiplication, except that no carry occurs:

```
inline ulong bitpol_mult(ulong a, ulong b)
// Return  A * B
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        a <<= 1;
    }
    return  t;
}
```

As for integer multiplication with the C-type `unsigned long`, the result will silently overflow if $deg(A) + deg(B)$ is equal to or greater than the word length (`BITS_PER_LONG`). If the operation `t^=a;` was replaced with `t+=a;` the ordinary (integer) product would be returned [FXT: gf2n/bitpolmult-demo.cc]:

```
        1..11.111 * 11.1.1
                          product as bitpol    ordinary product
        1..11.111   1    t= .....1..11.111   c= ......1..11.111
                    0
       1..11.111..  1    t= ...1.1111.1.11   c= ....11....1..11
                    0
     1..11.111....  1    t= .1.11.1..11.11   c= ..11..11.....11
    1..11.111.....  1    t= 11.....1111.11   c= 1.......11...11
```

When a binary polynomial $p = \sum_{k=0}^{d} a_k\, x^k$ is squared, the result equals $p^2 = \sum_{k=0}^{d} a_k\, x^{2k}$:

```
           1..11.111 * 1..11.111
           1..11.111   1    t= ........1..11.111
          1..11.111.   1    t= ......11.1.11..1
         1..11.111..   1    t= ......1111....1.1
                       0
       1..11.111....   1    t= ...1.1..1111.1.1
      1..11.111.....   1    t= ...11..1....1.1.1
                       0
                       0
   1..11.111........   1    t= 1.....1.1...1.1.1
 t= 1.....1.1...1.1.1
```

Thereby one can, instead of using `bitpol_mult(a, a)`, square by using the relation

$$\sum_{k=0}^{d} c_k\, x^k \quad = \quad \sum_{k=0}^{d} c_k\, x^{2k} \tag{38.1-1}$$

Which can be verified by repeated application of $(a+b)^2 = a^2 + 2\,a\,b + b^2 = a^2 + b^2$. So we just have to move the bits from position k to position $2k$:

```
inline ulong bitpol_square(ulong a)
// Return A * A
{
    ulong t = 0,  m = 1UL;
    while ( a )
    {
        if ( a&1 )  t ^= m;
        m <<= 2;
        a >>= 1;
    }
    return t;  // == bitpol_mult(a, a);
}
```

This version will unlikely give a speedup, but the equivalent function `bit_zip()` (see section 1.14 on page 35) can be a win if the degree of a is not too small.

38.1.2 Optimization of the squaring and multiplication routines

The routines for multiplication and squaring can be optimized by partially unrolling which avoids branches. As given, the function is compiled to:

```
  0: 31 c9                xor    %ecx,%ecx  // t = 0
  2: 48 85 ff             test   %rdi,%rdi  // a
  5: ba 01 00 00 00       mov    $0x1,%edx  // m = 1
  a: 74 1b                je     27 <_Z13bitpol_squarem+0x27>  // a==0 ?

 10: 48 89 c8             mov    %rcx,%rax  // tmp = t
 13: 48 31 d0             xor    %rdx,%rax  // tmp ^= m
 16: 40 f6 c7 01          test   $0x1,%dil  // if ( a&1 )
 1a: 48 0f 45 c8          cmovne %rax,%rcx  // then t = tmp
 1e: 48 c1 e2 02          shl    $0x2,%rdx  // m <<= 2
 22: 48 d1 ef             shr    %rdi       // a >>= 1
 25: 75 e9                jne    10 <_Z13bitpol_squarem+0x10>  // a!=0 ?

 27: 48 89 c8             mov    %rcx,%rax
 2a: c3                   retq
```

The `if()`-statement does not cause a branch so we unroll the contents of the loop 4-fold. Further, we move the `while()` statement to the end the loop to avoid the initial branch:

```
inline ulong bitpol_square(ulong a)
{
    ulong t = 0,  m = 1UL;
    do
    {
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
        if ( a&1 )  t ^= m;
        m <<= 2;  a >>= 1;
    }
    while ( a );
    return t;
}
```

Now we obtain much better machine code:

```
   0: 31 c9                 xor    %ecx,%ecx  // t = 0
   2: ba 01 00 00 00        mov    $0x1,%edx  // m = 1

   7: 48 89 c8              mov    %rcx,%rax  // tmp = t
   a: 48 31 d0              xor    %rdx,%rax  // tmp ^= m
   d: 40 f6 c7 01           test   $0x1,%dil  // if ( a&1 )
  11: 48 0f 45 c8           cmovne %rax,%rcx  // then t = tmp
  15: 48 c1 e2 02           shl    $0x2,%rdx  // m <<= 2
  19: 48 d1 ef              shr    %rdi       // a >>= 1

  1c: 48 89 c8              mov    %rcx,%rax
  1f: 48 31 d0              xor    %rdx,%rax
  22: 40 f6 c7 01           test   $0x1,%dil
  26: 48 0f 45 c8           cmovne %rax,%rcx
  2a: 48 c1 e2 02           shl    $0x2,%rdx
  2e: 48 d1 ef              shr    %rdi

  31: 48 89 c8              mov    %rcx,%rax
  [--snip--]
  43: 48 d1 ef              shr    %rdi

  46: 48 89 c8              mov    %rcx,%rax
  [--snip--]
  58: 48 d1 ef              shr    %rdi

  5b: 75 aa                 jne    7 <_Z13bitpol_squarem+0x7>  // a!=0 ?
  5d: 48 89 c8              mov    %rcx,%rax
  60: c3                    retq
```

The multiplication algorithm is optimized in the same way. For squaring one can also use the bit-zip function given in section 1.14 on page 35:

```
inline ulong bitpol_square(ulong a)
{
    return bit_zip( a );
}
```

The higher half of the bits of the argument must be zero.

38.1.3 Exponentiation

With a multiplication (and squaring) function at hand, it is straightforward (see section 27.6 on page 537) to implement the algorithm for binary exponentiation:

```
inline ulong bitpol_power(ulong a, ulong e)
// Return A ** e
{
    if ( 0==e )  return 1;

    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpol_square(s);
        e >>= 1;
    }

    a = s;
```

```
    while ( 0!=(e>>=1) )
    {
        s = bitpol_square(s);
        if ( e & 1 )  a = bitpol_mult(a, s);
    }
    return  a;
}
```

Note that overflow will occur even for moderate exponents.

38.1.4 Quotient and remainder

The remainder a modulo b division can be computed by initializing $A = a$ and subtracting $B = x^j \cdot b$ with $\deg(B) = \deg(A)$ from A at each step. The computation is finished as soon as $\deg b > \deg A$. As C-code:

```
inline ulong bitpol_rem(ulong a, ulong b)
// Return  R = A % B = A - (A/B)*B
// Must have: B!=0
{
    const ulong db = highest_bit_idx(b);
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
    }
    return  a;
}
```

The function `highest_bit_idx()` is given in section 1.6 on page 16. The following version may be superior if the degree of a is small or if no fast version of the function `highest_bit_idx()` is available:

```
    while ( b <= a )
    {
        ulong t = b;
        while ( (a^t) > t )  t <<= 1;
        // =^= while ( highest_bit(a) > highest_bit(t) )  t <<= 1;
        a ^= t;
    }
    return  a;
```

The quotient of two polynomials is computed by a function that does the computes the remainder and additionally keeps track of the quotient:

```
inline void bitpol_divrem(ulong a, ulong b, ulong &q, ulong &r)
// Set R, Q so that  A == Q * B + R.
// Must have B!=0.
{
    const ulong db = highest_bit_idx(b);
    q = 0;  // quotient
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
        q ^= (1UL<<(da-db));
    }
    r = a;
}
```

The division routine does the same computation but discards the remainder:

```
inline ulong bitpol_div(ulong a, ulong b)
// Return  Q = A / B
// Must have B!=0.
{
    const ulong db = highest_bit_idx(b);
    ulong q = 0;  // quotient
    ulong da;
    while ( db <= (da=highest_bit_idx(a)) )
    {
        if ( 0==a )  break;  // needed because highest_bit_idx(0)==highest_bit_idx(1)
        a ^= (b<<(da-db));
```

```
        q ^= (1UL<<(da-db));
    }
    return  q;
}
```

38.1.5 Greatest common divisor (GCD)

The polynomial greatest common divisor (GCD) can be computed with the Euclidean algorithm [FXT: bpol/bitpol-gcd.h]:

```
inline ulong bitpol_gcd(ulong a, ulong b)
// Return  polynomial gcd(A, B)
{
    if ( a<b )  { ulong t=a; a=b; b=t; }
    // here: b<=a
    while ( 0!=b )
    {
        ulong c = bitpol_rem(a, b);
        a = b;
        b = c;
    }
    return  a;
}
```

The binary GCD algorithm can be implemented as follows:

```
inline ulong bitpol_binary_gcd(ulong a, ulong b)
{
    if ( a < b )  swap2(a, b);
    if ( b==0 )  return a;

    ulong k = 0;
    while ( !((a|b)&1) )  // both divisible by x
    {
        k++;
        a >>= 1;
        b >>= 1;
    }
    while ( !(a&1) )  a >>= 1;
    while ( !(b&1) )  b >>= 1;

    while ( a!=b )
    {
        if ( a < b )  { ulong t=a; a=b; b=t; };  // swap if deg(A)<deg(B)
        ulong t = (a^b) >> 1;
        while ( !(t&1) )  t >>= 1;
        a = t;
    }

    return  a << k;
}
```

With a fast bit-scan instruction we can optimize the function:

```
inline ulong bitpol_binary_gcd(ulong a, ulong b)
{
    { // one (or both) of a, b zero?
        ulong ta = a&b,  to = a|b;
        if ( ta==to )  return to;
    }

    ulong ka = lowest_bit_idx(a);
    a >>= ka;
    ulong kb = lowest_bit_idx(b);
    b >>= kb;
    ulong k = ( ka<kb ? ka : kb );

    while ( a!=b )
    {
        if ( a < b )  { ulong t=a; a=b; b=t; }  // swap if deg(A)<deg(B)
        ulong t = (a^b) >> 1;
        a = (t >> lowest_bit_idx(t));
    }
    return  a << k;
}
```

Note that the comment

```
if ( a < b )  { ulong t=a; a=b; b=t; };  // swap if deg(A)<deg(B)
```

is not strictly correct as the swap can also happen with $\deg(a) = \deg(b)$ but that does no harm.

38.1.6 Exact division

Let C be a binary polynomial in x with constant term one. We use the relation (for power series)

$$\frac{1}{C} \;=\; \frac{1}{1-Y} \;=\; (1+Y)\,(1+Y^2)\,(1+Y^4)\,(1+Y^8)\,\ldots\,(1+Y^{2^n}) \mod x^{2^{n+1}} \qquad (38.1\text{-}2)$$

where $Y = 1 - C$. Now let $Y = x^{e_1} + x^{e_2} + \ldots + x^{e_k}$ where $e_i \geq 1$ and $e_{i+1} > e_i$. Then $Y^q = x^{qe_1} + x^{qe_2} + \ldots + x^{qe_k}$ whenever q is a power of two, and the multiplication by $(1 - Y^q)$ is obtained by shifts and subtractions. If A is an exact multiple of C then $R = A/C$ is a polynomial that can be computed as follows. We assume that arrays of N bits are used for the polynomials.

1. Set $R := A$ and let e_i (for $i = 1, 2, \ldots, k$) be the (ordered) positions of the nonzero coefficients of C. Set $q := 1$.
2. If $qe_1 \geq N$ then return R.
3. Set $T := 0$. For $j = 1, 2, \ldots, k$, set $T := T + R\,x^{qe_j}$. The multiplications with x^{qe_j} are left shifts by qe_j positions. Set $R := T$.
4. Set $q := 2\,q$ and goto step 2.

The most simple example is $C = 1 + x$ where the above procedure reduces to the inverse reversed Gray code given in section 1.15.6 on page 41. The method is most efficient when k, the number of nonzero coefficients of $C - 1$, is small. Sometimes one can reduce the work by dividing by $C\,D$ and finally multiplying by D for some appropriate D. For example, with all-ones polynomials $C = 1+x+x^2+\ldots+x^k$ and $D = 1 + x$, then $C\,D = 1 + x^{k+1}$.

If C is of the form $x^u\,(1 + \ldots + x^k)$ then A/C can obviously be computed as $(A/x^u)/(C/x^u)$.

An analogue of the algorithm for the exact division by $C = 2^k \pm 1$ (over $\mathbb{Z}$) is given in section 1.22.2 on page 55.

We give two examples, the division by $x + 1$ can be done by

```
inline ulong bitpol_div_xp1(ulong a)
// Return power series  A / (x+1)
// If A is a multiple of x+1, then the returned value
//   is the exact division by x+1
{
    a ^= a<<1;  // rev_gray ** 1
    a ^= a<<2;  // rev_gray ** 2
    a ^= a<<4;  // rev_gray ** 4
    a ^= a<<8;  // rev_gray ** 8
    a ^= a<<16; // rev_gray ** 16
#if  BITS_PER_LONG >= 64
    a ^= a<<32;  // for 64bit words
#endif
    return a;
}
```

The function is identical to the inverse reversed Gray code [FXT: bits/revgraycode.h], see section 1.15.6 on page 41. For the division by $x^2 + 1$ use

```
inline ulong bitpol_div_x2p1(ulong a)
// Return power series  A / (x^2+1)
// If A is a multiple of x^2+1, then the returned value
//   is the exact division by x^2+1
{
    a ^= a<<2;  // rev_gray ** 2
    a ^= a<<4;  // rev_gray ** 4
    a ^= a<<8;  // rev_gray ** 8
```

```
    a ^= a<<16; // rev_gray ** 16
#if  BITS_PER_LONG >= 64
    a ^= a<<32;  // for 64bit words
#endif
    return a;
}
```

38.2 Multiplication for polynomials of high degree

We used the straightforward multiplication scheme whose asymptotic cost is $\sim N^2$ for polynomials of degree N. This is fine when working with polynomials of small degree. For the multiplication of two polynomials U and V both of (high, even) degree N write $U = U_0 + U_1\,x^{N/2}$, $V = V_0 + V_1\,x^{N/2}$ and use the scheme

$$U\,V \;=\; U_0 \cdot V_0\,(1 + x^{N/2}) + (U_1 - U_0) \cdot (V_0 - V_1)\,x^{N/2} + U_1 \cdot V_1\,(x^{N/2} + x^N) \tag{38.2-1}$$

recursively. Only the three multiplications indicated by a dot are expensive, the multiplications by a power of x are just shifts. The resulting scheme is the *Karatsuba multiplication for polynomials*, relation 27.2-3 on page 525 interpreted for polynomials (set $x^{N/2} = B$). Recursive application of the scheme leads to the asymptotic cost $\sim N^{\log_2(3)} \approx N^{1.585}$. When working with polynomials of high degree the implementation of the Karatsuba scheme is a must.

We give a generalization of the Karatsuba splitting that involves no constants, and several Toom-Cook schemes.

38.2.1 Splitting schemes that do not involve constants

A generalization of the Karatsuba scheme is given in [238] (see also [239]), it does not lead asymptotically better schemes than $\sim N^{\log_2(3)}$ but has a simple structure and avoids all multiplications by constants (the asymptotically better n-way splitting schemes method do involve multiplications by constants for all $n \geq 3$, see section 27.2 on page 524). We give the scheme for degree-2 (3-term) polynomials, recursive application for 3^n-term polynomials should be straightforward. Let

$$A \;=\; a_2\,x^2 + a_1\,x + a_0 \tag{38.2-2a}$$
$$B \;=\; b_2\,x^2 + b_1\,x + b_0 \tag{38.2-2b}$$
$$C \;=\; A\,B \;=\; c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0 \tag{38.2-2c}$$

We want to compute c_0, c_1, …, c_4. With

$$d_{0,0} \;=\; a_0\,b_0 \tag{38.2-3a}$$
$$d_{1,1} \;=\; a_1\,b_1 \tag{38.2-3b}$$
$$d_{2,2} \;=\; a_2\,b_2 \tag{38.2-3c}$$
$$d_{0,1} \;=\; (a_0 + a_1)\,(b_0 + b_1) \tag{38.2-3d}$$
$$d_{0,2} \;=\; (a_0 + a_2)\,(b_0 + b_2) \tag{38.2-3e}$$
$$d_{1,2} \;=\; (a_1 + a_2)\,(b_1 + b_2) \tag{38.2-3f}$$

the c_k can be obtained as

$$c_0 \;=\; d_{0,0} \tag{38.2-4}$$
$$c_1 \;=\; d_{0,1} - d_{0,0} - d_{1,1} \tag{38.2-5}$$
$$c_2 \;=\; d_{0,2} - d_{0,0} - d_{2,2} + d_{1,1} \tag{38.2-6}$$
$$c_3 \;=\; d_{1,2} - d_{1,1} - d_{2,2} \tag{38.2-7}$$
$$c_4 \;=\; d_{2,2} \tag{38.2-8}$$

The scheme involves 6 multiplications and 13 additions. Recursive application leads to the asymptotic cost $\sim N^{\log_3(6)} \approx 1.6309$ which is slightly worse than for the 2-term scheme. However, applying this scheme first for a polynomial with $N = 3 \cdot 2^n$ terms and then using the Karatsuba scheme recursively can be advantageous.

We generalize the method for n-term polynomials and denote the scheme by KA-n. The 2-term scheme KA-2 is the Karatsuba algorithm. With

$$A = \sum_{k=0}^{n-1} a_k\, x^k \tag{38.2-9a}$$

$$B = \sum_{k=0}^{n-1} b_k\, x^k \tag{38.2-9b}$$

$$C = A\,B =: \sum_{k=0}^{2n-2} c_k\, x^k \tag{38.2-9c}$$

define

$$d_{s,s} := a_s\, b_s \qquad \text{for } s = 0,\, 1,\, \ldots,\, n-1 \tag{38.2-10a}$$

$$d_{s,t} := (a_s + b_s)\,(a_t + b_t) \quad \text{for } s + t = i,\ t > s \geq 0,\ 1 \leq i \leq 2\,n-3 \tag{38.2-10b}$$

$$c_i^* = \sum_{\substack{s+t=i \\ t>s\geq 0}} d_{s,t} - \sum_{\substack{s+t=i \\ n-1\geq t>s\geq 0}} (d_{s,s} + d_{t,t}) \tag{38.2-10c}$$

Then

$$c_0 = d_{0,0} \tag{38.2-11a}$$

$$c_{2\,n-2} = d_{n-1,n-1} \tag{38.2-11b}$$

and for $0 < i < 2\,n-2$:

$$c_i = \begin{cases} c_i^* & \text{if} \quad i \text{ odd} \\ c_i^* + d_{i/2,i/2} & \text{else} \end{cases} \tag{38.2-11c}$$

The Karatsuba scheme is obtained for $n = 2$.

We give pari/gp code whose output is the KA-n algorithm for given n. We need to create symbols '`ak`' (for a_k), '`bk`', and so on:

```
fa(k)=eval(Str("a" k))
fb(k)=eval(Str("b" k))
fc(k)=eval(Str("c" k))
fd(k,j)=eval(Str("d" k "" j))
```

For example, we can create a symbolic polynomial of degree 3:

```
? sum(k=0,3, fa(k) * x^k)
  a3*x^3 + a2*x^2 + a1*x + a0
```

The next routine generates the definitions of all $d_{s,t}$. It returns the number of multiplications involved:

```
D(n)=
{
    local(mct);
    mct = 0; \\ count multiplications
    for (i=0, n-1,  mct+=1; print(fd(i,i), " = ", fa(i), " * ", fb(i) ) );
    for (t=1, n-1,
        for (s=0, t-1,
            mct += 1;
            print(fd(s,t), " = (", fa(s)+fa(t), ") * (", fb(s)+fb(t), ")" ) ;
        );
    );
    return(mct);
}
```

For $n = 3$ the output is

```
d00 = a0 * b0
d11 = a1 * b1
d22 = a2 * b2
d01 = (a0 + a1) * (b0 + b1)
d02 = (a0 + a2) * (b0 + b2)
d12 = (a1 + a2) * (b1 + b2)
```

The following routine prints c_i, the coefficient of the product, in terms of several $d_{s,t}$. It returns the additions involved:

```
C(i, n)=
{
    local(N, s, act);
    act = -1; \\ count additons
    print1(fc(i), " = ");
    for (s=0, i-1,
        t = i - s;
        if ( (t>s) && (t<n),
             act += 3;
             print1(" + ", fd(s,t));
             print1(" - ", fd(s,s));
             print1(" - ", fd(t,t));
        );
    );
    if ( 0==i%2,  act+=1; print1(" + ", fd(i/2,i/2)) );
    print();
    return( act );
}
```

It has to be called for all i where $0 \le i \le 2n-2$. The algorithm is generated as follows:

```
KA(n)=
{
    local(mct, act);
    act = 0; \\ count additons
    mct = 0; \\ count multiplications
    mct = D(n);  \\ generate definitions for the d_{s,t}
    \\ generate rules for computation of c_i in terms of d_{s,t}:
    for (i=0, 2*n-2,  act+=C(i,n) );
    act += n*(n-1); \\ additions when setting up d(i,j) for i!=j
    return( [mct, act] );
}
```

With $n = 3$ we obtain

```
c0 =  + d00
c1 =  + d01 - d00 - d11
c2 =  + d02 - d00 - d22 + d11
c3 =  + d12 - d11 - d22
c4 =  + d22
```

Now we generate the definitions for the KA-5 algorithm:

```
n=5 /* n terms, degree=n-1 */
default(echo, 0);
KA(n);
```

We obtain the algorithm KA-5 shown in figure 38.2-A. The format is valid pari/gp input, so we add a few lines that print code to check the algorithm:

```
print("A=",sum(k=0,n-1, fa(k) * x^k))
print("B=",sum(k=0,n-1, fb(k) * x^k))
print("/* direct computation of the product: */")
print("C=A*B")
print("/* Karatsuba computation of the product: */")
print("K=",sum(k=0,2*n-2, fc(k) * x^k))
print("qq=K-C")
print("print( if(0==qq, \"OK.\", \" **** OUCH!\") )")
```

This gives for $n = 5$:

```
A=a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
B=b4*x^4 + b3*x^3 + b2*x^2 + b1*x + b0
/* direct computation of the product: */
C=A*B
/* Karatsuba computation of the product: */
K=c8*x^8 + c7*x^7 + c6*x^6 + c5*x^5 + c4*x^4 + c3*x^3 + c2*x^2 + c1*x + c0
qq=K-C
```

```
    d00 = a0 * b0
    d11 = a1 * b1
    d22 = a2 * b2
    d33 = a3 * b3
    d44 = a4 * b4
    d01 = (a0 + a1) * (b0 + b1)
    d02 = (a0 + a2) * (b0 + b2)
    d12 = (a1 + a2) * (b1 + b2)
    d03 = (a0 + a3) * (b0 + b3)
    d13 = (a1 + a3) * (b1 + b3)
    d23 = (a2 + a3) * (b2 + b3)
    d04 = (a0 + a4) * (b0 + b4)
    d14 = (a1 + a4) * (b1 + b4)
    d24 = (a2 + a4) * (b2 + b4)
    d34 = (a3 + a4) * (b3 + b4)

    c0 =  + d00
    c1 =  + d01 - d00 - d11
    c2 =  + d02 - d00 - d22 + d11
    c3 =  + d03 - d00 - d33 + d12 - d11 - d22
    c4 =  + d04 - d00 - d44 + d13 - d11 - d33 + d22
    c5 =  + d14 - d11 - d44 + d23 - d22 - d33
    c6 =  + d24 - d22 - d44 + d33
    c7 =  + d34 - d33 - d44
    c8 =  + d44
```

Figure 38.2-A: Code for the algorithm KA-5.

```
    print( if(0==qq, "OK.", " **** OUCH!") )
```

We can feed the output into another pari/gp session to verify the algorithm. We use the option '`-f`' that prevents colorization of the output which would confuse the verification process, the option '`-q`' suppresses the output of the version:

```
   gp -f -q < karatsuba-n.gp  |  gp
```

We obtain (shortened and comments added):

```
/* definitions of d(s,t): */
b0*a0
b1*a1
b2*a2
b3*a3
b4*a4
(b0 + b1)*a0 + (a1*b0 + b1*a1)
(b0 + b2)*a0 + (a2*b0 + b2*a2)
[--snip--]
(b3 + b4)*a3 + (a4*b3 + b4*a4)

/* the c_i in terms of d(s,t), evaluated: */
b0*a0
b1*a0 + a1*b0
b2*a0 + (a2*b0 + b1*a1)
b3*a0 + (a3*b0 + (b2*a1 + a2*b1))
b4*a0 + (a4*b0 + (b3*a1 + (a3*b1 + b2*a2)))
b4*a1 + (a4*b1 + (b3*a2 + a3*b2))
b4*a2 + (a4*b2 + b3*a3)
b4*a3 + a4*b3
b4*a4

/* polynomials: */
a4*x^4 + a3*x^3 + a2*x^2 + a1*x + a0
b4*x^4 + b3*x^3 + b2*x^2 + b1*x + b0

/* direct computation of product: */
b4*a4*x^8 + (b4*a3 + a4*b3)*x^7 + (b4*a2 + (a4*b2 + b3*a3))*x^6 + [...]

/* Karatsuba computation of product: */
b4*a4*x^8 + (b4*a3 + a4*b3)*x^7 + (b4*a2 + (a4*b2 + b3*a3))*x^6 + [...]

/* difference: */
0

OK.  /* we are fine! */
```

The number of multiplications with algorithm KA-n is $(n^2+n)/2$ which is suboptimal except for $n = 2$. However, recursive application can be worthwhile. One should start with the biggest prime factors as the number of additions is then minimized. The number of multiplications does not depend on the order of recursion (see [238] which also tabulates the number of additions and multiplications for $n \leq 128$).

With n just below a highly composite number one should add (zero-valued) 'dummy' terms and recursively use KA-n algorithms for small n. For example, with polynomials of degree 63 recursion with KA-2 (and $n = 64$) will beat the scheme "KA-7, then KA-3".

One could write code generators that create expanded versions of the recursions for n the product of small primes. When the cost of multiplication is much higher than for addition (as for binary polynomial multiplication on general purpose CPUs) substantial savings can be expected.

38.2.2 Toom-Cook algorithms

Toom-Cook schemes with $n \geq 3$ that work for binary polynomials are described in [46].

3-way splitting

```
A = a2*Y^2 + a1*Y + a0
B = b2*Y^2 + b1*Y + b0

S3 = a2 + a1 + a0;
S2 = b2 + b1 + b0;
S1 = S3 * S2;          \\ Mult (1)
S0 = a2*x^2 + a1*x;
S4 = b2*x^2 + b1*x;
S3 += S0;
S2 += S4;
S0 += a0;
S4 += b0;
S3 *= S2;              \\ Mult (2)
S2 = S0 * S4;          \\ Mult (3)
S4 = a2 * b2;          \\ Mult (4)
S0 = a0 * b0;          \\ Mult (5)

S3 += S2;

S2 += S0;  S2 /= x;  S2 += S3;
T = S4;  T *= (x^3+1);  \\ temporary variable
S2 += T;  S2 /= (x+1);  \\ exact division

S1 += S0;
S3 += S1;  S3 /= x;  S3 /= (x+1);  \\ exact division
S1 += S4;  S1 += S2;
S2 += S3;

P = S4*Y^4 + S3*Y^3 + S2*Y^2 + S1*Y + S0;
Mod(1,2)* (P - A*B) \\ == zero
```

Figure 38.2-B: Implementation of the 3-way multiplication scheme for binary polynomials. The five expensive multiplications are commented with '`Mult (n)`'.

For the multiplication of two polynomials A and B both of degree $3N$ write

$$A \;=\; a_0 + a_1\,x^N + a_2\,x^{2N} \;=:\; a_0 + a_1\,Y + a_2\,Y^2 \tag{38.2-12}$$

and identically for B. A 3-way splitting scheme for multiplication is shown in figure 38.2-B. The multiplications and divisions by x are shifts and the exact divisions are linear operations if we use the method of section 38.1.6 on page 798.

4-way splitting

For the multiplication of two polynomials A and B both of degree $4N$ write

$$A \;=\; a_0 + a_1\,Y + a_2\,Y^2 + a_3\,Y^3 \tag{38.2-13}$$

where $Y := x^N$, and identically for B. The 4-way splitting multiplication scheme is shown in figure 38.2-C.

```
A = a3*Y^3 + a2*Y^2 + a1*Y + a0;
B = b3*Y^3 + b2*Y^2 + b1*Y + b0;

S1 = a3 + a2 + a1 + a0;
S2 = b3 + b2 + b1 + b0;
S3 = S1 * S2;                \\ Mult (1)
S0 = a1 + x*(a2 + x*a3);
S6 = b1 + x*(b2 + x*b3);
S4 = (S0 + a3*(x+1))*x + S1;
S5 = (S6 + b3*(x+1))*x + S2;
S0 = S0*x + a0;
S6 = S6*x + b0;
S5 = S5 * S4;                \\ Mult (2)
S4 = S0 * S6;                \\ Mult (3)
S0 = a0*x^3 + a1*x^2 + a2*x;
S6 = b0*x^3 + b1*x^2 + b2*x;
S1 = S1 + S0 + a0*(x^2+x);
S2 = S2 + S6 + b0*(x^2+x);
S0 = S0 + a3;
S6 = S6 + b3;
S1 = S1 * S2;                \\ Mult (4)
S2 = S0 * S6;                \\ Mult (5)
S6 = a3 * b3;                \\ Mult (6)
S0 = a0 * b0;                \\ Mult (7)

S1 = S1 + S2 + S0*(x^4+x^2+1);
S5 = (S5 + S4 + S6*(x^4+x^2+1) + S1) \ (x^4+x);
S2 = S2 + S6 + S0*x^6;
S4 = S4 + S2 + S6*x^6 + S0;
S4 = (S4 + S5*(x^5+x)) \ (x^4+x^2);
S3 = S3 + S0 + S6;
S1 = S1 + S3;
S2 = S2 + S1*x + S3*x^2;
S3 = S3 + S4 + S5;
S1 = (S1 + S3*(x^2+x)) \ (x^4+x);
S5 = S5 + S1;
S2 = (S2 + S5*(x^2+x)) \ (x^4+x^2);
S4 = S4 + S2;

P = S6*Y^6 + S5*Y^5 + S4*Y^4 + S3*Y^3 + S2*Y^2 + S1*Y + S0;
Mod(1,2)*(P - A*B)  \\ == zero
```

Figure 38.2-C: Implementation of the 4-way multiplication scheme for binary polynomials. The seven expensive multiplications are commented with '`Mult (n)`'.

38.2.3 FFT methods for binary polynomials of very high degree

For polynomials of very high degree FFT-based algorithms can be used. The most simple method is to use integer multiplication without the carry phase (which is polynomial multiplication!). Our multiplication example can be recomputed using decimal digits. The carry phase of the integer multiplication is replaced by a reduction modulo 2:

```
 100110111 * 110101
== 11022223331211  // integer multiplication
== 11000001111011  // parity of digits
```

The scheme will work for polynomials of degree less than nine only. When using an FFT multiplication scheme (see section 27.3 on page 532) we can multiply polynomials up to degree N as long as the integer values 0, 1, 2 ... $N+1$ can be distinguished after computing the FFT. This is hardly a limitation at all: with the C-type `float` (24 bit mantissa) polynomials up to degree one million can be multiplied assuming at least 20 bits are correct after the FFT. With type `double` (53-bit mantissa) there is no practical limit. Whether this method could ever beat a well implemented splitting routine is unclear. However, it is very easy to implement. A FFT method that works exclusively with binary polynomials is described in [209].

38.3 Modular arithmetic with binary polynomials

Here we consider arithmetic of binary polynomials modulo a binary polynomial. Addition and subtraction are again the XOR operation. The functions shown in this section are given in [FXT: bpol/bitpolmod-arith.h].

38.3.1 Multiplication and squaring

Multiplication of a polynomial A by x modulo (a polynomial) C is achieved by shifting left and subtracting C if the coefficient shifted out is one:

```
static inline ulong bitpolmod_times_x(ulong a, ulong c, ulong h)
// Return  (A * x) mod C
// where A and C represent polynomials over Z/2Z:
//  W = pol(w) =: \sum_k{ [bit_k(w)] * x^k}
//
// h needs to be a mask with one bit set:
//  h == highest_bit(c) >> 1  == 1UL << (degree(C)-1)
{
    ulong s = a & h;
    a <<= 1;
    if ( s )  a ^= c;
    return  a;
}
```

In order to avoid the repeated computation of the highest set bit we introduced the auxiliary variable h that has to be initialized as described in the comment. Section 1.6 on page 16 gives algorithms for the function `highest_bit()`. Note that h needs to be computed only if the degree of the modulus C changes, which is usually only once for a series of calculations. By using the variable h we can obtain the correct result even if the degree of C equals the number of bits in a word in which case C does not fit into a word.

Multiplication of two polynomials a and b modulo C can be achieved by adding a reduction step to the binary multiplication routine:

```
inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
// Return  (A * B) mod C
{
    ulong t = 0;
    while ( b )
    {
        if ( b & 1 )  t ^= a;
        b >>= 1;
        ulong s = a & h;
        a <<= 1;
        if ( s )  a ^= c;
    }
    return  t;
}
```

38.3.2 Optimization of the squaring and multiplication routines

Squaring a can be achieved as the multiplication $a \cdot a$. If many squaring have to be done with a fixed modulus then a optimization using a precomputed table of the residues x^{2k} mod C shown in section 40.1 on page 851 can be useful. Squaring of the polynomial $\sum_{k=0}^{d} a_k\, x^k$ can be achieved via the computation the sum $\sum_{k=0}^{d} a_k\, x^{2k}$ modulo C. We use the auxiliary function

```
static inline ulong bitpolmod_times_x2(ulong a, ulong c, ulong h)
// Return  (A * x * x) mod C
{
    { ulong s=a&h; a<<=1; if (s) a^=c; }
    { ulong s=a&h; a<<=1; if (s) a^=c; }
    return  a;
}
```

The squaring function, with a the 4-fold unrolled loop, is

```
static inline ulong bitpolmod_square(ulong a, ulong c, ulong h)
// Return A*A mod C
{
    ulong t = 0, s = 1;
    do
    {
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
        if (a&1) t^=s; a>>=1; s=bitpolmod_times_x2(s, c, h);
    }
    while ( a );
    return t;
}
```

Whether the unrolled code is used can be specified via the line

```
#define  MULT_UNROLL  // define to unroll loops 4-fold
```

The optimization used for the multiplication routine is also unrolling as described in section 38.1.2 on page 794:

```
static inline ulong bitpolmod_mult(ulong a, ulong b, ulong c, ulong h)
{
    ulong t = 0;
    do
    {
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
        { if(b&1) t^=a; b>>=1; ulong s=a&h; a<<=1; if(s) a^=c; }
    }
    while ( b );
    return  t;
}
```

It turns out that squaring via multiplication is slightly faster than via the described sum computation.

38.3.3 Exponentiation

A routine for modular exponentiation can be obtained using the right-to-left powering algorithm from section 27.6.1 on page 537:

```
inline ulong bitpolmod_power(ulong a, ulong e, ulong c, ulong h)
// Return (A ** e)  mod C
{
    if ( 0==e )  return 1;  // avoid hang with e==0 in next while()
    ulong s = a;
    while ( 0==(e&1) )
    {
        s = bitpolmod_square(s, c, h);
        e >>= 1;
    }
    a = s;
    while ( 0!=(e>>=1) )
    {
        s = bitpolmod_square(s, c, h);
        if ( e & 1 )  a = bitpolmod_mult(a, s, c, h);
    }
    return  a;
}
```

The left-to-right powering algorithm given in section 27.6.2 on page 538 can be implemented as:

```
inline ulong bitpolmod_power(ulong a, ulong e, ulong c, ulong h)
{
    ulong s = a;
    ulong b = highest_bit(e);
    while ( b>1 )
    {
        b >>= 1;
        s = bitpolmod_square(s, c, h); // s *= s;
```

```
        if ( e & b )  s = bitpolmod_mult(s, a, c, h);   // s *= a;
    }
    return s;
}
```

Computing a power of x can be optimized with this scheme:

```
inline ulong bitpolmod_xpower(ulong e, ulong c, ulong h)
// Return (x ** e)  mod C
{
    ulong s = 2;  // 'x'
    ulong b = highest_bit(e);
    while ( b>1 )
    {
        b >>= 1;
        s = bitpolmod_square(s, c, h); // s *= s;
        if ( e & b )  s = bitpolmod_times_x(s, c, h);   // s *= x;
    }
    return s;
}
```

38.3.4 Division by x

Division by x is possible if the modulus has a nonzero constant term (that is, $\gcd(C, x) = 1$):

```
static inline ulong bitpolmod_div_x(ulong a, ulong c, ulong h)
// Return  (A / x) mod C
// C must have nonzero constant term: (c&1)==1
{
    ulong s = a & 1;
    a >>= 1;
    if ( s )
    {
        a ^= (c>>1);
        a |= h;  // so it also works for  n == BITS_PER_LONG
    }
    return  a;
}
```

If we do not insist on correct results for the case that the degree of C equals the number of bits in a word, we could simply use the following two-liner:

```
    if ( a & 1 )  a ^= c;
    a >> 1;
```

The operation needs only about two CPU cycles. The inverse of x can be computed with:

```
static inline ulong bitpolmod_inv_x(ulong c, ulong h)
// Return  (1 / x) mod C
// C must have nonzero constant term: (c&1)==1
{
    ulong a = (c>>1);
    a |= h;  // so it also works for  n == BITS_PER_LONG
    return  a;
}
```

38.3.5 Extended GCD, computation of the inverse, and division

The algorithm for the computation of extended GCD (EGCD) is taken from [155] [FXT: bpol/bitpol-gcd.h]:

```
inline ulong bitpol_egcd(ulong u, ulong v, ulong &iu, ulong &iv)
// Return u3 and set u1,v1 so that  gcd(u,v) == u3 == u*u1 + v*u2
{
    ulong u1 = 1,  u2 = 0;
    ulong v1 = 0,  v3 = v;
    ulong u3 = u,  v2 = 1;
    while ( v3!=0 )
    {
        ulong q = bitpol_div(u3, v3);  // == u3 / v3;

        ulong t1 = u1 ^ bitpol_mult(v1, q);  // == u1 - v1 * q;
        u1 = v1;  v1 = t1;
```

```
        ulong t3 = u3 ^ bitpol_mult(v3, q);  // == u3 - v3 * q;
        u3 = v3;  v3 = t3;
        ulong t2 = u2 ^ bitpol_mult(v2, q);  // == u2 - v2 * q;
        u2 = v2;  v2 = t2;
    }
    iu = u1;  iv = u2;
    return u3;
}
```

The routine can be optimized using `bitpol_divrem()`: remove the lines

```
        ulong q = bitpol_div(u3, v3);  // == u3 / v3;
    [--snip--]
        ulong t3 = u3 ^ bitpol_mult(v3, q);  // == u3 - v3 * q;
```

and insert at the beginning of the body of the loop:

```
        ulong q, t3;
        bitpol_divrem(u3, v3, q, t3);
```

The routine computes the GCD g and two additional quantities i_u and i_v so that

$$g \quad = \quad u \cdot i_u + v \cdot i_v \tag{38.3-1}$$

When $g = 1$ we have

$$1 \quad \equiv \quad u \cdot i_u \bmod v \tag{38.3-2}$$

That is, i_u is the inverse of u modulo v. Thereby [FXT: bpol/bitpolmod-arith.h]:

```
inline ulong bitpolmod_inverse(ulong a, ulong c)
// Returns the inverse of A modulo C if it exists, else zero.
// Must have deg(A) < deg(C)
{
    ulong i, t;  // t unused
    ulong g = bitpol_egcd(a, c, i, t);
    if ( g!=1 )  i = 0;
    return i;
}
```

Modular division is obtained by multiplication with the inverse:

```
inline ulong bitpolmod_divide(ulong a, ulong b, ulong c, ulong h)
// Return a/b modulo c.
// Must have: gcd(b,c)==1
{
    ulong i = bitpolmod_inverse(b, c);
    a = bitpolmod_mult(a, i, c, h);
    return a;
}
```

When the modulus is an irreducible polynomial (see section 38.4) then the inverse can also be computed via powering:

```
inline ulong bitpolmod_inverse_irred(ulong a, ulong c, ulong h)
// Return (A ** -1)  mod C
// Must have: C irreducible.
{
    ulong r1 = (h<<1) - 2;  // max order minus one
    ulong i =  bitpolmod_power(a, r1, c, h);
    return  i;
}
```

38.4 Irreducible and primitive polynomials

A polynomial is called *irreducible* if it has no non-trivial factors (trivial factors are the constant polynomial '1' and the polynomial itself). A polynomial that has a non-trivial factorization is called *reducible*.

A polynomial with zero constant term is always reducible because it has the factor x, except if the polynomial equals x. A binary polynomial that is irreducible has at least one term of odd degree and an odd number of terms in total.

The irreducible polynomials are the 'primes' among the polynomials.

Whether a given polynomial can be factorized does depend on its coefficient field: $p(x) = x^2 + 1$ does not factor as an 'ordinary' polynomial (coefficients in $\mathbb{R}$ or $\mathbb{Z}$) while the factorization over $\mathbb{C}$ is $(x^2+1) = (x+i)\,(x-i)$. As a binary polynomial, the factorization is $(x^2 + 1) = (x + 1)^2$.

Let C be irreducible. Then the sequence $p_k = x^k \bmod (C)$, $k = 1, 2, \ldots$ is periodic and the (smallest) period m of the sequence is the order of x modulo C. We call m the *period* (or *order*) of the polynomial C. For a binary polynomial of degree n the maximal period equals $2^n - 1$. If the period is maximal then all nonzero polynomials of degree $n - 1$ occur in the sequence p.

For the period m of C we have $x^m = 1 \bmod C$, so $x^m - 1 = 0 \bmod C$. That is, C divides $x^m - 1$ but no polynomial $x^k - 1$ with $k < m$.

A polynomial is called *primitive* if its period is maximal. Then the powers of x generate all nonzero binary polynomials of degree $n - 1$. The polynomial x is a generator ('primitive root') modulo C. Primitivity implies irreducibility, the converse is not true.

The situation is somewhat parallel to the operations modulo an integer:

- Among those integers m that are prime some have the primitive root 2: the sequence 2^k for $k = 1, 2, \ldots, m - 1$ contains all nonzero numbers modulo m (see chapter 25 on page 507).
- Among those polynomials C that are irreducible some are primitive: the sequence x^k for $k = 1, 2, \ldots, 2^n - 1$ contains all nonzero polynomials modulo C.

Note that there is another notion of the term 'primitive', that of a polynomial for which the greatest common divisor of all coefficients is one.

Working modulo prime m the inverse of a number a can be obtained as $a^{-1} = a^{m-2}$ ($m-1$ is the maximal order of an element in $\mathbb{Z}/m\mathbb{Z}$). With an irreducible polynomial C of degree n the inverse modulo C of a polynomial A can be obtained by computing $A^{-1} = A^{2^n-2}$ as shown in section 38.3.5 on page 807.

The *weight* of a binary polynomial is the sum of its coefficients.

Roots of primitive polynomials have maximal order

A different characterization of primitivity is as follows. Suppose you want to do computations with linear combinations $A = \sum_{k=0}^{n-1} a_k\, \alpha^k$ (where $a_k \in \mathrm{GF}(2)$) of the powers of an (unknown!) root α of an irreducible polynomial $C = x^n + \sum_{k=0}^{n-1} c_k\, x^k$.

When multiplying A with the root α we obtain a term α^n which we want to get rid of. But we have

$$\alpha^n \quad = \quad \sum_{k=0}^{n-1} c_k\, \alpha^k \tag{38.4-1}$$

as α is a root of the polynomial C. Therefore we can use exactly the same modular reduction as with polynomial computation modulo C. The same is true for the multiplication of two linear combinations (of the powers of the same root α).

We see that the order of a polynomial p is the order of its root α modulo p, and that a polynomial is primitive if and only if all of its roots have maximal order.

38.4.1 Testing for irreducibility

Irreducibility tests for binary polynomials use the fact that the polynomial $x^{2^k} + x$ has all irreducible polynomials of degree k as a factor. For example,

$$\begin{aligned} x^{2^6} + x &= x^{64} + x \\ &= (x) \cdot (x+1) \cdot \\ &\quad \cdot \left(x^2 + x + 1\right) \cdot \\ &\quad \cdot \left(x^3 + x + 1\right) \cdot \left(x^3 + x^2 + 1\right) \cdot \\ &\quad \cdot \left(x^6 + x + 1\right) \cdot \left(x^6 + x^3 + 1\right) \cdot \left(x^6 + x^4 + x^2 + x + 1\right) \cdot \\ &\quad \cdot \left(x^6 + x^4 + x^3 + x + 1\right) \cdot \left(x^6 + x^5 + 1\right) \cdot \left(x^6 + x^5 + x^2 + x + 1\right) \cdot \\ &\quad \cdot \left(x^6 + x^5 + x^3 + x^2 + 1\right) \cdot \left(x^6 + x^5 + x^4 + x + 1\right) \cdot \left(x^6 + x^5 + x^4 + x^2 + 1\right) \end{aligned} \tag{38.4-2}$$

A bit more can be said about the polynomial $x^{2^k} + x$ used in the tests: if

$$z = x^{2^d} + x \tag{38.4-3a}$$

$$s = 1 + \sum_{k=0}^{d-1} x^{2^k} \tag{38.4-3b}$$

$$t = 0 + \sum_{k=0}^{d-1} x^{2^k} = s - 1 = z/s \tag{38.4-3c}$$

then s has all degree-d irreducible polynomials as factors where the coefficient of x^{d-1} is one and the factorization of t consists of polynomials where that coefficient is zero. As an example consider the case $d = 7$. We use the notation $[a, b, c, \ldots, h] := x^a + x^b + x^c + \ldots + x^h$:

```
z = [128,1]  = s * t

s = [64,32,16,8,4,2,1,0] =          t = [64,32,16,8,4,2,1] =
    [1,0] *                             [1] *
    [7,6,0] *                           [7,1,0] *
    [7,6,3,1,0] *                       [7,3,0] *
    [7,6,4,1,0] *                       [7,3,2,1,0] *
    [7,6,4,2,0] *                       [7,4,0] *
    [7,6,5,2,0] *                       [7,4,3,2,0] *
    [7,6,5,3,2,1,0] *                   [7,5,2,1,0] *
    [7,6,5,4,0] *                       [7,5,3,1,0] *
    [7,6,5,4,2,1,0] *                   [7,5,4,3,0] *
    [7,6,5,4,3,2,0]                     [7,5,4,3,2,1,0]
```

38.4.1.1 The Ben-Or test for irreducibility

A polynomial C of degree d is reducible if $\gcd(x^{2^k} + x \bmod C, C) \neq 1$ for any $k < d$. We compute $u_k = x^{2^k}$ (modulo C) for each $k < d$ by successive squarings and test whether $\gcd(u_k + x, C) = 1$ for all k. But as a factor of degree f implies another one of degree $d - f$ it suffices to do the first $\lfloor d/2 \rfloor$ of the tests. The implied algorithm is called the *Ben-Or irreducibility test*. A C++ implementation is given in [FXT: bpol/bitpol-irred-ben-or.cc]:

```
bool bitpol_irreducible_q(ulong c, ulong h)
// Return whether C is irreducible (via the Ben-Or irreducibility test_;
// h needs to be a mask with one bit set:
//  h == highest_bit(C) >> 1  == 1UL << (degree(C)-1)
{
    if ( c<4 )
    {
        if ( c>=2 )  return true;   // x, and 1+x are irreducible
        else         return false;  // constant polynomials are reducible
    }
```

```
    if ( 0==(1&c) )  return false;  // x is a factor
    // if ( 0==(c & 0xaaaaaaaaUL ) )  return 0; // at least one odd degree term
    // if ( 0==parity(c) )  return 0; // need odd number of nonzero coeff.
    // if ( 0!=bitpol_test_squarefree(c) )  return 0; // must be square free
    ulong d = h >> 1;
    ulong u = 2;  // =^= x
    while ( 0 != d )  // floor( degree/2 ) times
    {
        // Square r-times for coefficients of c in GF(2^r).
        // We have r==1:
        u = bitpolmod_square(u, c, h);
        ulong upx = u ^ 2;  // =^= u+x
        ulong g = bitpol_binary_gcd(upx, c);
        if ( 1!=g )  return  false;   // reducible
        d >>= 2;
    }
    return  true;  // irreducible
}
```

Commented out at the beginning are a few tests to check for trivial necessary conditions for irreducibility. For the test `bitpol_test_squarefree` (for a square factor) see section 38.8.2 on page 829. The routine will fail if $\deg c =$`BITS_PER_LONG`, because the gcd-computation fails in this case.

38.4.1.2 Rabin's test for irreducibility

Rabin's algorithm to test for irreducibility (given in [195]) can be stated as follows: A binary polynomial C of degree d is irreducible only if

$$\gcd\left(x^{2^d} - x \bmod C,\, C\right) \;=\; 0 \tag{38.4-4a}$$

and, for all prime divisors p_i of d,

$$\gcd\left(x^{2^{d/p_i}} - x \bmod C,\, C\right) \;=\; 1 \tag{38.4-4b}$$

The first condition is equivalent to $x^{2^d} \equiv x \bmod C$. Thereby the number of GCD computations equals the number of prime divisors of d.

When the prime divisors are processed in decreasing order the successive exponents are increasing and the power of x can be updated via squarings (the idea can be found in [116]). Thereby the total number of squarings equals d which is minimal. A pari/gp implementation is

```
polirred2_rabin(c)=
{
    local(d, f, np, p, e, X, m, g, ns);
    d = poldegree(c);
    c *= Mod(1,2);
    if ( c=='x,  return(1) );  \\  'x' is irreducible
    if ( 0==polcoeff(c,0),  return(0));  \\ reducible
    if ( d<1,  return(0) ); \\ '0' and '1' are not irreducible
    if ( 1!=gcd(c, deriv(c,'x)),  return(0));  \\ reducible
    ns = 0;  \\ numbers of squarings so far
    m = Mod( Mod(1,2) * 'x, c);
    if ( ! isprime(d),  \\ only if composite
        f = factor(d);
        np = matsize(f)[1];  \\ number of prime divisors
        forstep (k=np, 1, -1,  \\ biggest prime factor first
            p = f[k,1];  \\ prime divisor
            e = d/p;
            m = m^(2^(e-ns));
            \\ here: m == Mod('x,c)^(2^e);
```

```
            ns = e;
            g = gcd(component(m,2)-'x, c);
            if ( 1!=g, return(0) );  \\ reducible
        );
    );
    m = m^(2^(d-ns));
    \\ here: m == Mod('x,c)^(2^d);
    if ( 0!=m-'x,  return(0) );  \\ reducible
   return( 1 );  \\ irreducible
}
```

Rabin's test will be faster than the Ben-Or test if the polynomial is irreducible. When the polynomial is reducible and has small factors (as often the case with 'random' polynomials) then the Ben-Or test will terminate faster. A comparison of the tests is given in [116].

A C++ implementation of Rabin's test is given in [FXT: bpol/bitpol-irred-rabin.cc]. A table of auxiliary bitmasks gives the number of squarings between the GCD computations:

```
static const ulong rabin_tab[] =
{
 0UL,   // x = 0  (bits: ..........)    OPS:
 0UL,   // x = 1  (bits: ..........)    OPS:   finally sqr 1 times
 0UL,   // x = 2  (bits: ..........)    OPS:   finally sqr 2 times
 0UL,   // x = 3  (bits: ..........)    OPS:   finally sqr 3 times
 4UL,   // x = 4  (bits: ........1..)   OPS:  sqr 2 times,    finally sqr 2 times
 0UL,   // x = 5  (bits: ..........)    OPS:   finally sqr 5 times
 12UL,  // x = 6  (bits: .......11..)   OPS:  sqr 2 times,   sqr 1 times,   finally sqr 3 times
 0UL,   // x = 7  (bits: ..........)    OPS:   finally sqr 7 times
 16UL,  // x = 8  (bits: ......1....)   OPS:  sqr 4 times,    finally sqr 4 times
 8UL,   // x = 9  (bits: .......1...)   OPS:  sqr 3 times,    finally sqr 6 times
 36UL,  // x = 10 (bits: .....1..1..)   OPS:  sqr 2 times,   sqr 3 times,   finally sqr 5 times
  [--snip--]
```

The testing routine is

```
inline bool bitpol_irreducible_rabin_q(ulong c, ulong h)
// Return whether C is irreducible (via Rabin's irreducibility test).
// h needs to be a mask with one bit set:
//  h == highest_bit(C) >> 1  == 1UL << (degree(C)-1)
{
    if ( c<4 )
    {
        if ( c>=2 )  return true; // x, and 1+x are irreducible
        else         return false; // constant polynomials are reducible
    }
    if ( 0==(1&c) )  return false; // x is a factor
    ulong d = 1 + lowest_bit_idx(h);  // degree
    ulong rt = rabin_tab[d];
    ulong m = 2; // =^= 'x'
    while ( rt > 1 )
    {
        do
        {
            --d;
            m = bitpolmod_square(m, c, h);
            rt >>= 1;
        }
        while ( 0 == (rt & 1) );
        ulong g = bitpol_binary_gcd( m ^ 2UL, c );
        if ( g!=1 )  return false;
    }
    do  { m = bitpolmod_square(m, c, h); }  while ( --d );
    if ( m ^ 2UL )  return false;
    return true;
}
```

38.4.1.3 Transformations that preserve irreducibility

When a polynomial is irreducible then the composition with $x+1$ is also irreducible. Similar, the reversed word corresponds to another irreducible polynomial. Reversion also preserves primitivity, composition with $x+1$ does not in general: the most simple example is the primitive polynomial $p(x) = x^4 + x^3 + 1$ where $p(x+1) = x^4 + x^3 + x^2 + x + 1$ has the order 5. The order of x modulo $p(x)$ equals the order of $x+1$ modulo $p(x+1)$.

Composition with $x+1$ can be computed by [FXT: bpol/bitpol-irred.h]:

```
inline ulong bitpol_compose_xp1(ulong c)
// Return C(x+1).
// Self-inverse.
{
    ulong z = 1;
    ulong r = 0;
    while ( c )
    {
        if ( c & 1 )  r ^= z;
        c >>= 1;
        z ^= (z<<1);
    }
    return  r;
}
```

A version that finishes in time $\log_2(b)$ (where $b =$ bits per word) is

```
inline ulong bitpol_compose_xp1(ulong c)
{
    ulong s = BITS_PER_LONG >> 1;
    ulong m = ~0UL << s;
    while ( s )
    {
        c ^= ( (c&m) >> s );
        s >>= 1;
        m ^= (m>>s);
    }
    return  c;
}
```

Which is exactly the `blue_code()` from section 1.18 on page 45.

The *reciprocal* of a polynomial $F(x)$ is the polynomial

$$F^*(x) \quad = \quad x^{\deg F} F(1/x) \tag{38.4-5}$$

The roots of $F^*(x)$ are the inverses of the roots of $F(x)$. The reciprocal of a binary polynomial is the reversed binary word:

```
inline ulong bitpol_recip(ulong c)
// Return x^deg(C) * C(1/x)  (the reciprocal polynomial)
{
    ulong t = 0;
    while ( c )
    {
        t <<= 1;
        t |= (c & 1);
        c >>= 1;
    }
    return  t;
}
```

Alternatively, one can use the bit-reversion routines given in section 1.13 on page 30.

In general the sequence of successive 'compose' and 'reverse' operations leads to 6 different polynomials:

```
C= [11, 10, 4, 3, 0]

   [11, 10, 4, 3, 0]  -- recip (C=bitpol_recip(C)) -->
   [11,  8, 7, 1, 0]   -- compose (C=bitpol_compose_xp1(C)) -->
   [11, 10, 9, 7, 6, 5, 4, 1, 0]  -- recip -->
   [11, 10, 7, 6, 5, 4, 2, 1, 0]  -- compose -->
   [11,  9, 7, 2, 0]  -- recip -->
   [11,  9, 4, 2, 0]  -- compose -->
   [11, 10, 4, 3, 0]  == initial value
```

A list of all binary irreducible polynomials of degrees $2 \le d \le 11$ is given in [FXT: data/all-irredpoly.txt].

38.4.1.4 Self-reciprocal polynomials

```
              irred. poly      irred. SRP
 1:    11...1..11    111..1..111..1..111
 2:    1..111.111    1.11111.111.11111.1
 3:    1.11.11.11    1..111.11111.111..1
 4:    11111...11    11.1...1.1.1...1.11
 5:    1....1.111    1.1..1.11111.1..1.1
 6:    11.11.1.11    11111..1.1.1..11111
 7:    111...1111    11..1..11111..1..11
 8:    1....11.11    1.1..11..1..11..1.1
 9:    11.111..11    11111111111111111111
10:    1..11.1111    1.111....1....111.1
11:    1..1.11111    1.11.11..1..11.11.1
12:    1111..1.11    11.11..11111..11.11
13:    1.11..1111    1..11...111...11..1
14:    11.1.11.11    1111.111.1.111.1111
15:    1111111.11    11.1.11111111.1.11
16:    1111...111    11.11.1..1..1.11.11
17:    1.1.11.111    1....1.1.1.1.1....1
18:    11.1..1111    1111..1..1..1..1111
19:    11.1111111    111111...1...111111
20:    1.1.1.1111    1.....1111111.....1
21:    1.1.1...11    1.......1.......1
22:    1..1..1.11    1.11..11.1.11..11.1
23:    1.1....111    1...1.11.1.11.1...1
24:    1.......11    1.1.....111.....1.1
25:    11..11111     111..111.1.111..111
26:    11..111.11    111.11...1...11.111
27:    1...11..11    1.1.111.111.111.1.1
28:    11..1...11    111.1.1.111.1.1.111
```

Figure 38.4-A: All irreducible self-reciprocal binary polynomials of degree 18 (right) and the corresponding irreducible polynomials of degree 9 with constant linear coefficient (left).

A polynomial is called *self-reciprocal* if it is its own reciprocal. The irreducible self-reciprocal polynomials (SRPs), except for $1+x$, are of even degree $2\,d$. They can be computed from the irreducible polynomials of degree d with nonzero linear coefficient. Let $F(x) = \sum_{j=0}^{d} f_j\, x^j$ and $S_F(x)$ the corresponding SRP, then

$$S_F(x) \;\; = \;\; x^d\, F(x+1/x) \;=\; \sum_{j=0}^{d} F_j\, x^{d-j}\, (1+x^2)^j \tag{38.4-6}$$

The irreducible SRPs of degree 18 and their corresponding polynomials are shown in figure 38.4-A [FXT: gf2n/bitpol-srp-demo.cc]. The conversion can be implemented as [FXT: bpol/bitpol-srp.h]:

```
inline ulong bitpol_pol2srp(ulong f, ulong d)
// Return  the self-reciprocal polynomial S=x^d*F(x+1/x) where d=deg(f).
// W = sum(j=0, d,   F(j)*x^(d-j)*(1+x^2)^j )  where
//   F(j) is the j-th coefficient of F.
// Must have: d==degree(F)
{
    ulong w = 1;  // == (x^2+1)^j
    ulong s = 0;
    do  // for j = 0 ... d:
    {
        if ( f & 1 )  s ^= (w << d);  // S += F(j)*x^(d-j)*(1+x^2)^j
        w ^= (w<<2);                  // w *= (1+x^2)
        f >>= 1;                      // next coefficient to low end
    }
    while ( d-- );
    return s;
}
```

The inverse function is given in [176]:

1. Initialization: set $F := 0$, and $j := 0$.

2. If $S \bmod (x^2+1) \equiv 0$ then set $f_j := 0$, else set $f_j := 1$.
3. Set $S := (S - f_j\, x^{d-j})/(x^2+1)$ [the division is exact].
4. Set $j := j+1$. If $j \le d$ goto step 2.
5. Return F $(= \sum_{j=0}^{d} f_j\, x^j)$.

The computation of $S \bmod (x^2+1)$ can be omitted because the quantity is zero exactly if the central coefficient of S is zero. The assignment $S := (S - f_j\, x^{d-j})/(x^2+1)$ can be replaced by $S := S/(x^2+1)$ (as power series) because no coefficient beyond the position $d-j$ is needed by the following steps. We use the power series division shown in section 38.1.6 on page 798 for this computation:

```
inline ulong bitpol_srp2pol(ulong s, ulong hd)
// Inverse of bitpol_pol2srp().
// Must have: hd = degree(s)/2 (note: _half_ of the degree).
// Only the lower half coefficients are accessed, i.e.
//   the routine works for degree(S) <= 2*BITS_PER_LONG-2.
{
    ulong f = 0;
    ulong mh = 1UL << hd;
    ulong ml = 1;
    do
    {
        ulong b = s & mh;  // central coefficient
//        s ^= b;  // set central coefficient to zero (not needed)
        if ( b )  f |= ml;  // positions 0,1,...,hd
        ml <<= 1;
        s = bitpol_div_x2p1(s);  // exact division by (x^2+1)
    }
    while ( (mh>>=1) );
    return f;
}
```

The self-reciprocal polynomials of degree $2n$ are factors of the polynomial $2^{2^n+1} - 1$ (see [181]). For example, for $n = 5$ we obtain

```
? lift(factormod(x^(2^5+1)-1,2))
  [x + 1 1]
  [x^2 + x + 1 1]
  [x^10 + x^7 + x^5 + x^3 + 1 1]
  [x^10 + x^9 + x^5 + x + 1 1]
  [x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 1]
```

The order of a self-reciprocal polynomial of degree $2n$ is a divisor of $2^{2^n} + 1$. The list of all SRP of even degree up to degree 22 is given in [FXT: data/all-irred-srp.txt].

38.4.2 Testing for primitivity

Checking a degree-d binary polynomial for primitivity by directly using the definition costs proportional 2^n operations which is prohibitive except for tiny d.

Improvements come from the multiplicative structure of the problem: it suffices to check whether $x^k \equiv 1 \bmod C$ for all divisors of the maximal order that are greater than n. However, this soon gets impractical quickly as n grows. Already with $n = 144$ one has maximal order $m = 2^{144} - 1$ (which has 262144 divisors of which 262112 have to be tested) so the computation gets expensive. Note that the implied algorithm needs the factorization of $m = 2^n - 1$.

A much better solution is a modification of the algorithm to determine the oder in a finite field given on page 744. The implementation given here uses the pari/gp language:

```
polorder(p) =
/* Order of x modulo p (p irreducible over GF(2)) */
{
  local(g, g1, te, tp, tf, tx);
  g = 'x;
  p *= Mod(1,2);
  te = nn_;
```

```
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
    );
  );
  return( te );
}
```

The function uses the following global variables that must be set up before call:

```
nn_ = 0;  /* max order  =  2^n-1 */
np_ = 0;  /* number of primes in factorization */
vp_ = []; /* vector of primes */
vf_ = []; /* vector of factors (prime powers) */
vx_ = []; /* vector of exponents */
```

As given, the algorithm will do n_p exponentiations modulo p where n_p is the number of different primes in the factorization in m. For $n = 144$ one has $n_p = 17$. An implementation of the algorithm in C++ is given in [FXT: bpol/bitpol-order.cc].

A shortcut that makes the algorithm terminate as soon as the computed order drops below maximum is

```
polmaxorder_q(p) =
/* Whether order of x modulo p is maximal  (p irreducible over GF(2)) */
/* Early-out variant */
{
  local(g, g1, te, tp, tf, tx, ct);
  g = 'x;
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    ct = 0;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
      ct = ct + 1;
    );
    if ( ct<tx,  return(0) );
  );
  return(1);
}
```

Range	time	# irred. tests	tests/sec
90 .. 99:	<1 sec	258	>300
190 .. 199:	6 sec	588	100
290 .. 299:	10 sec	474	58
390 .. 399:	30 sec	810	27

Determination of irreducible polynomials.

Range	time	# irred. tests	# prim. tests
90 .. 99:	4 sec	313	13
190 .. 199:	40 sec	1371	25
290 .. 299:	90 sec	864	14
390 .. 399:	260 sec	1529	14

Determination of primitive polynomials.

Figure 38.4-B: Timings and number of tests involved in the determination of 10 low-bit irreducible (top) and primitive (bottom) binary polynomials in selected ranges of their degrees. The small number of tests for primitivity indicates that an irreducible polynomial is likely also primitive.

Using `polmaxorder_q()` and pari's built-in `polisirreducible()` the search for the lowest-bit primitive

polynomials up to degree $n = 100$ is a matter of about 10 seconds. Extending the list up to $n = 200$ takes 3 minutes. The computation of all polynomials up to degree $n = 400$ takes less than on hour. Figure 38.4-B shows approximate timings with the determination of irreducible and primitive polynomials.

Again, the algorithm depends an precomputed factorizations. The table [FXT: data/mersenne-factors.txt] taken from [68] was used in order to save computation time.

For prime $m = 2^n - 1$ (that is, m is a Mersenne prime) irreducibility suffices for primality: The one-liner

```
x=127; for(z=1,x-1,if(polisirreducible(Mod(1,2)+t^z+t^x),print1(" ",z)))
```

finds all primitive trinomials for exponents of Mersenne primes in no time:

```
  89:  38 51
 127:  1 7 15 30 63 64 97 112 120 126
 521:  32 48 158 168 353 363 473 489
 607:  105 147 273 334 460 502
```

The computation (for $d = 607$) takes about a minute. Note we did not exploit the symmetry (reversed polynomials are also primitive). Techniques to find primitive trinomials whose degree are very big Mersenne exponents are described in [65].

Here is a surprising theorem: Let $p(x) = \sum_{k=0}^{d} c_k\, x^k$ be an irreducible binary polynomial, and $L_p(x) := \sum_{k=0}^{d} c_k\, x^{2^k}$. Then all irreducible factors of $L_p(x)/x$ (a polynomial of degree $2^d - 1$) are of degree equal to $\mathrm{ord}(p)$ (the order of x modulo $p(x)$). Especially, if $p(x)$ is primitive, then $L_p(x)/x$ is irreducible. The theorem is proven in [252] (also in [169, p.110]). An example: $x^7 + x + 1$ is primitive, so $x^{127} + x + 1$ is irreducible. But, as $2^{127}-1$ is prime, $x^{127}+x+1$ is also primitive. Thereby the polynomial $x^{2^{127}-1} + x + 1$ is irreducible.

38.4.3 Irreducible and primitive polynomials of special forms *

We give lists of irreducible and primitive polynomials of special forms. For the computation of irreducible polynomials the built-in routine `polisirreducible()` of the pari/gp package (see [189]) was used. Primitivity was tested with the routines given so far. The abbreviation 'PP' is used for 'primitive polynomial' in what follows.

38.4.3.1 All irreducible polynomials for low degrees

For degrees $n \le 8$ the complete list of irreducible polynomials is shown in figure 38.4-C. The list up to degree $n = 11$ is given in [FXT: data/all-irredpoly.txt]. The list of PPs for $n \le 11$ is given in [FXT: data/all-primpoly.txt].

38.4.3.2 All irreducible and primitive trinomials for low degrees

A *trinomial* is a polynomial with exactly three nonzero coefficients. The irreducible binary trinomials for degrees $n \le 49$ are shown in figure 38.4-D (there are no irreducible trinomials for degrees 50 and 51). A list of all irreducible trinomials up to degree $n = 400$ is given in [FXT: data/all-trinomial-irredpoly.txt]. A more compact form of the list can is given in [FXT: data/all-trinomial-primpoly-short.txt]:

```
 2:  1
 3:  1 2
 4:  1 3
 5:  2 3
 6:  1 5
 7:  1 3 4 6
 9:  4 5
10:  3 7
11:  2 9
15:  1 4 7 8 11 14
```

A line starts with the entry for the degree followed by all possible positions of the middle coefficient. The corresponding files giving primitive trinomials only are [FXT: data/all-trinomial-primpoly.txt] and

```
2,1,0
3,1,0
3,2,0
4,1,0
4,3,0
# non-primitive:
4,3,2,1,0
5,2,0
5,3,0
5,3,2,1,0
5,4,2,1,0
5,4,3,1,0
5,4,3,2,0
6,1,0
6,4,3,1,0
6,5,0
6,5,2,1,0
6,5,3,2,0
6,5,4,1,0
# non-primitive:
6,3,0
6,4,2,1,0
6,5,4,2,0
7,1,0
7,3,0
7,3,2,1,0
7,4,0
7,4,3,2,0
7,5,2,1,0
7,5,3,1,0
7,5,4,3,0
7,5,4,3,2,1,0
7,6,0
7,6,3,1,0
7,6,4,1,0
7,6,4,2,0
7,6,5,2,0
7,6,5,3,2,1,0
7,6,5,4,0
7,6,5,4,2,1,0
7,6,5,4,3,2,0
8,4,3,2,0
8,5,3,1,0
8,5,3,2,0
8,6,3,2,0
8,6,4,3,2,1,0
8,6,5,1,0
8,6,5,2,0
8,6,5,3,0
8,6,5,4,0
8,7,2,1,0
8,7,3,2,0
8,7,5,3,0
8,7,6,1,0
8,7,6,3,2,1,0
8,7,6,5,2,1,0
8,7,6,5,4,2,0
# non-primitive:
8,4,3,1,0
8,5,4,3,0
8,5,4,3,2,1,0
8,6,5,4,2,1,0
8,6,5,4,3,1,0
8,7,3,1,0
8,7,4,3,2,1,0
8,7,5,1,0
8,7,5,4,0
8,7,5,4,3,2,0
8,7,6,4,2,1,0
8,7,6,4,3,2,0
8,7,6,5,4,1,0
8,7,6,5,4,3,0
```

Figure 38.4-C: All binary irreducible polynomials up to degree 8.

```
 2,1    10,3    17,3    22,1    -30,1    35,2    -42,7
 3,1    10,7    17,5    22,21   -30,9    35,33   -42,35
 3,2    11,2    17,6    23,5    -30,21  -36,9    -44,5
 4,1    11,9    17,11   23,9    -30,29   36,11   -44,39
 4,3   -12,3    17,12   23,14    31,3   -36,15   -46,1
 5,2   -12,5    17,14   23,18    31,6   -36,21   -46,45
 5,3   -12,7   -18,3    25,3     31,7    36,25    47,5
 6,1   -12,9    18,7    25,7     31,13  -36,27    47,14
-6,3   -14,5   -18,9    25,18    31,18   39,4     47,20
 6,5   -14,9    18,11   25,22    31,24   39,8     47,21
 7,1    15,1   -18,15   -28,1    31,25   39,14    47,26
 7,3    15,4    20,3    28,3     31,28   39,25    47,27
 7,4    15,7   -20,5    28,9    -33,10   39,31    47,33
 7,6    15,8   -20,15   28,13    33,13   39,35    47,42
-9,1    15,11   20,17   28,15    33,20   41,3     49,9
 9,4    15,14   21,2    28,19   -33,23   41,20    49,12
 9,5            -21,7   28,25   -34,7    41,21    49,15
-9,8            -21,14 -28,27   -34,27   41,38    49,22
                 21,19   29,2                     49,27
                         29,27                    49,34
                                                  49,37
                                                  49,40
```

Figure 38.4-D: All irreducible trinomials $x^n + x^k + 1$ for degrees $n \leq 49$. The format of the entries is `n,k` for primitive trinomials, and `-n,k` for non-primitive trinomials.

[FXT: data/all-trinomial-primpoly-short.txt]. A list of irreducible trinomials that are *not* primitive is [FXT: data/all-trinomial-nonprimpoly.txt].

Regarding trinomials, there is a theorem by Swan (given in [223]): The trinomial $x^n + x^k + 1$ over $GF(2)$ has an even number of irreducible factors (and so is reducible) if

1. n is even, k is odd, $n \neq 2k$, and either $nk/2 \equiv 0 \bmod 4$ or $nk/2 \equiv 1 \bmod 4$
2. n is odd, k is even and does not divide $2n$, and $n \equiv \pm 3 \bmod 8$
3. n is even, k is odd and does divide $2n$, and $n \equiv \pm 1 \bmod 8$
4. Any of the above holds for k replaced by $n - k$ (that is, for the reciprocal trinomial)

The first condition implies that no irreducible trinomial for n a multiple of 8 exists (as n is even, k must be odd, else the trinomial is a perfect square; and $nk/2 \equiv 0 \bmod 4$). Further, if n is a prime with $n \equiv \pm 3 \bmod 8$ then the trinomial can be irreducible only if $k = 2$ (or $n - k = 2$). In the note [81] it is shown that no irreducible trinomial exists for n a prime such that $n \equiv 13 \bmod 24$ or $n \equiv 19 \bmod 24$.

The primitive trinomials of the form $x^n + x + 1$ for $n \leq 400$ are those with $n \in$

```
2, 3, 4, 6, 7, 15, 22, 60, 63, 127, 153
```

These numbers are the sequence A073639 of [214], where one finds in addition 471, 532, 865, 900, 1366 with the next candidate being 4495.

For some applications one may want to use reducible trinomials whose period is close to that of a primitive one. For example, the trinomial (given in [82])

$$x^{32} + x^{15} + 1 = \left(x^{11} + x^9 + x^7 + x^2 + 1\right) \cdot \left(x^{21} + x^{19} + x^{15} + x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^4 + x^2 + 1\right) \quad (38.4\text{-}7)$$

has the period $p = 4,292,868,097$ which is very close to $2^{32} - 1 = 4,294,967,295$. Note that the degree is a multiple of eight, so no irreducible trinomial of that degree exists.

38.4.3.3 Irreducible trinomials of the form $1 + \mathbf{x^k} + \mathbf{x^d}$

k=1: The polynomials of the form $1 + x + x^d$ are irreducible for the following $2 \leq d \leq 34353$:

```
2, 3, 4, 6, 7, 9, 15, 22, 28, 30, 46, 60, 63,
127, 153, 172, 303, 471, 532, 865, 900,
1366, 2380, 3310, 4495, 6321, 7447,
10198, 11425, 21846, 24369, 27286, 28713, 32767, 34353
```

This is sequence A002475 of [214].

k=2: $1 + x^2 + x^d$ is irreducible for the following $3 \leq d \leq 57341$ (sequence A057460):

```
3, 5, 11, 21, 29, 35, 93, 123, 333, 845, 4125,
10437, 10469, 14211, 20307, 34115, 47283, 50621, 57341
```

k=3: $p = 1 + x^3 + x^d$ is irreducible for the following $4 \leq d \leq 1000$ (sequence A057461):

```
4, 5, 6, 7, 10, 12, 17, 18, 20, 25, 28, 31, 41, 52, 66,
130, 151, 180, 196, 503, 650, 761, 986
```

k=4: $p = 1 + x^4 + x^d$ is irreducible for the following $5 \leq d \leq 1000$ (sequence A057463):

```
7, 9, 15, 39, 57, 81, 105
```

k=5: $p = 1 + x^5 + x^d$ is irreducible for the following $6 \leq d \leq 1000$ (sequence A057474):

```
6, 9, 12, 14, 17, 20, 23, 44, 47, 63, 84,
129, 236, 278, 279, 297, 300, 647, 726, 737,
```

38.4.3.4 Irreducible trinomials of the form $1 + x^d + x^{kd}$

k=2: The polynomial $p = 1 + x^d + x^{2d}$ is irreducible whenever d is a power of three:

```
  1:  x^2 +  x +  1
  3:  x^6 +  x^3 +  1
  9:  x^18 +  x^9 +  1
 27:  x^54 +  x^27 +  1
 81:  x^162 +  x^81 +  1
243:  x^486 +  x^243 +  1
...
```

k=3: Similarly, $p = 1 + x^d + x^{3d}$ is irreducible whenever d is a power of seven:

```
  1:  x^3 +  x +  1
  7:  x^21 +  x^7 +  1
 49:  x^147 +  x^49 +  1
343:  x^1029 +  x^343 +  1
```

k=4: The polynomial $p = 1 + x^d + x^{4d}$ whenever $d = 3^i\,5^j$, $i, j \in \mathbb{N}$:

```
4, 12, 20, 36, 60, 100, 108, 180, 300, 324, 500, 540, 900, 972,
1500, 1620, 2500, 2700, 2916, 4500, 4860, 7500, 8100, 8748, 12500, ...
```

Similar regularities can be observed for related forms, see [43].

38.4.3.5 Primitive pentanomials

A *pentanomial* is a polynomial that has exactly five nonzero coefficients. PPs that are pentanomials are given in [FXT: data/pentanomial-primpoly.txt]. No primitive pentanomial exists for degrees $n < 5$ but for all higher degrees one seems to exist but this has not been proven so far. Entries with the special form $x^n + x^3 + x^2 + x + 1$ are $n \in \{5, 7, 17, 25, 31, 41, 151\}$.

38.4.3.6 Primitive minimum-weight and low-bit polynomials

```
[1,0]              17,3,0             33,13,0            49,9,0
2,1,0              18,7,0             34,8,4,3,0         50,4,3,2,0
3,1,0              19,5,2,1,0         35,2,0             51,6,3,1,0
4,1,0              20,3,0             36,11,0            52,3,0
5,2,0              21,2,0             37,6,4,1,0         53,6,2,1,0
6,1,0              22,1,0             38,6,5,1,0         54,8,6,3,0
7,1,0              23,5,0             39,4,0             55,24,0
8,4,3,2,0          24,4,3,1,0         40,5,4,3,0         56,7,4,2,0
9,4,0              25,3,0             41,3,0             57,7,0
10,3,0             26,6,2,1,0         42,7,4,3,0         58,19,0
11,2,0             27,5,2,1,0         43,6,4,3,0         59,7,4,2,0
12,6,4,1,0         28,3,0             44,6,5,2,0         60,1,0
13,4,3,1,0         29,2,0             45,4,3,1,0         61,5,2,1,0
14,5,3,1,0         30,6,4,1,0         46,8,7,6,0         62,6,5,3,0
15,1,0             31,3,0             47,5,0             63,1,0
16,5,3,2,0         32,7,6,2,0         48,9,7,4,0         64,4,3,1,0
```

Figure 38.4-E: Binary primitive polynomials of minimum weight.

```
[1,0]              17,3,0             33,6,4,1,0         49,6,5,4,0
2,1,0              18,5,2,1,0         34,7,6,5,2,1,0     50,4,3,2,0
3,1,0              19,5,2,1,0         35,2,0             51,6,3,1,0
4,1,0              20,3,0             36,6,5,4,2,1,0     52,3,0
5,2,0              21,2,0             37,5,4,3,2,1,0     53,6,2,1,0
6,1,0              22,1,0             38,6,5,1,0         54,6,5,4,3,2,0
7,1,0              23,5,0             39,4,0             55,6,2,1,0
8,4,3,2,0          24,4,3,1,0         40,5,4,3,0         56,7,4,2,0
9,4,0              25,3,0             41,3,0             57,5,3,2,0
10,3,0             26,6,2,1,0         42,5,4,3,2,1,0     58,6,5,1,0
11,2,0             27,5,2,1,0         43,6,4,3,0         59,6,5,4,3,1,0
12,6,4,1,0         28,3,0             44,6,5,2,0         60,1,0
13,4,3,1,0         29,2,0             45,4,3,1,0         61,5,2,1,0
14,5,3,1,0         30,6,4,1,0         46,8,5,3,2,1,0     62,6,5,3,0
15,1,0             31,3,0             47,5,0             63,1,0
16,5,3,2,0         32,7,5,3,2,1,0     48,7,5,4,2,1,0     64,4,3,1,0
```

Figure 38.4-F: Binary primitive polynomials of small weight and nonzero coefficient at low indices.

The data in [FXT: data/minweight-primpoly.txt] lists minimal-weight PPs where in addition the coefficients are as close to the low end as possible. The first entries are shown in figure 38.4-E A list of minimal-weight PPs that fit into a machine word is given in [FXT: bpol/primpoly-minweight.cc].

Choosing those PPs where the highest nonzero coefficient is as low as possible one obtains the list in [FXT: data/lowbit-primpoly.txt]. It starts as shown in figure 38.4-F. The corresponding extract for small degrees is given in [FXT: bpol/primpoly-lowbit.cc]. The index (position) of the second highest nonzero coefficient (the *subdegree* of the polynomial) grows slowly with n and is ≤ 12 for all $n \leq 400$. Thereby one can store the list compactly as an array of 16-bit words.

38.4.3.7 All primitive low-bit polynomials for certain degrees

A list of all PPs $x^n + \sum_{j=0}^{k} c_j\, x^j$ for degree $n = 256$ with the second-highest order $k \leq 15$ (and the first few polynomials for $k = 16$) is given in [FXT: data/lowbit256-primpoly.txt]. The list starts as

```
256,10,5,2,0
256,10,8,5,4,1,0
256,10,9,8,7,4,2,1,0
256,11,8,4,3,2,0
256,11,8,6,4,3,0
256,11,10,9,4,2,0
256,11,10,9,7,4,0
256,12,7,5,4,2,0
256,12,8,7,6,3,0
```

Equivalent tables for degrees `DEG=` 63, 64, 127, 128, 256, 512, 521, 607, 1000, and 1024, can be found in the files `data/lowbitDEG-primpoly.txt` (where `DEG` has to be replaced by the number).

38.4.3.8 Primitive low-block polynomials

A low-block polynomial has the special form $x^n + \sum_{j=0}^{k} x^j$. Such PPs exist for 218 degrees $n \leq 400$. These are especially easy to store in an array (saving the index of the second highest nonzero coefficient in array element n). Moreover, simplified and possible more efficient routines based on the special forms might be possible. A complete list of all low-block PPs with degree $n \leq 400$ is given in [FXT: data/all-lowblock-primpoly.txt]. A short form of the list is [FXT: data/all-lowblock-primpoly-short.txt]. Among the low-block PPs are a few where just one bit (the coefficient after the leading coefficient) is not set. For $n \leq 400$ this is for the following degrees:

```
3, 5, 7, 13, 15, 23, 37, 47, 85, 127, 183, 365, 383
```

The PPs listed in [FXT: data/lowblock-primpoly.txt] have the smallest possible block of set bits.

38.4.3.9 Irreducible all-ones polynomials

Irreducible polynomials of the form $x^n + x^{n-1} + x^{n-2} + \ldots + x + 1$ (so-called *all-ones polynomials*) exist whenever $s := n + 1$ is a prime number for which 2 is a primitive root. The all-ones polynomials $x^{s-1} + \ldots + 1$ are irreducible for the following $s < 400$:

```
2, 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, 83, 101, 107, 131, 139, 149,
163, 173, 179, 181, 197, 211, 227, 269, 293, 317, 347, 349, 373, 379, 389
```

The sequence is entry A001122 of [214]. A list of values up to 2000 is shown in figure 39.6-B on page 842.

With the exception of x^2+x+1, none of the all-ones polynomials is primitive. In fact, the order of x equals s, which is immediate when printing the powers of x (example using $s=5$, $p=x^4+x^3+x^2+x+1$):

```
k     x^k
0    ...1
1    ..i.
2    .i..
3    i...
4    1111
5    ...1  == x^5 == 1
```

The all-ones polynomials are a special case for the factorization of cyclotomic polynomials, see section 38.7 on page 826. Irreducible polynomials of high weight are considered in [5] where irreducible polynomials of the form $(x^{n+1}+1)/(x+1)+x^k$ up to degree 340 are given.

38.4.3.10 Irreducible normal polynomials

```
 2,1,0                 8,7,2,1,0             -9,8,0
                      -8,7,3,1,0              9,8,4,1,0
 3,2,0                 8,7,3,2,0              9,8,4,2,0
                      -8,7,4,3,2,1,0          9,8,4,3,2,1,0
 4,3,0                -8,7,5,1,0              9,8,5,4,0
-4,3,2,1,0             8,7,5,3,0              9,8,5,4,3,1,0
                      -8,7,5,4,0             -9,8,6,3,0
 5,4,2,1,0            -8,7,5,4,3,2,0          9,8,6,3,2,1,0
 5,4,3,1,0             8,7,6,1,0              9,8,6,4,3,1,0
 5,4,3,2,0             8,7,6,3,2,1,0          9,8,6,5,3,1,0
                      -8,7,6,4,2,1,0          9,8,6,5,3,2,0
 6,5,0                -8,7,6,4,3,2,0          9,8,6,5,4,1,0
 6,5,2,1,0             8,7,6,5,2,1,0          9,8,7,2,0
 6,5,4,1,0            -8,7,6,5,4,1,0          9,8,7,3,2,1,0
-6,5,4,2,0             8,7,6,5,4,2,0          9,8,7,5,4,3,0
                      -8,7,6,5,4,3,0          9,8,7,6,2,1,0
 7,6,0                                        9,8,7,6,3,1,0
 7,6,3,1,0                                    9,8,7,6,4,2,0
 7,6,4,1,0                                    9,8,7,6,4,3,0
 7,6,4,2,0                                    9,8,7,6,5,1,0
 7,6,5,2,0                                    9,8,7,6,5,4,3,1,0
 7,6,5,3,2,1,0
 7,6,5,4,2,1,0
```

Figure 38.4-G: All normal irreducible polynomials up to degree $n=9$. Polynomials that are not primitive are marked with a '-'.

The *normal* irreducible polynomials are those whose roots are linearly independent (see section 40.6 on page 865). A complete list up to degree $n=13$ is given in [FXT: data/all-normalpoly.txt], figure 38.4-G shows the polynomials up to degree $n=8$. Normal polynomials must have subdegree $n-1$, that is, they are of the form $x^n+x^{n-1}+\ldots$. The condition is necessary but not sufficient: not all irreducible polynomials of subdegree $n-1$ are normal. A list of primitive normal polynomials $x^n+x^{n-1}+\ldots+x^w+1$ with w as big as possible is given in [FXT: data/highbit-normalpoly.txt]. Primitive normal polynomials $x^n+x^{n-1}+x^w+\ldots+1$ where w is as small as possible are given in [FXT: data/lowbit-normalprimpoly.txt]. The all-ones polynomials are normal.

38.4.3.11 Irreducible alternating polynomials

The 'alternating' polynomial $1+\sum_{k=0}^{d} x^{2\,k+1} = 1+x+x^3+x^5\ldots+x^{2d+1}$ can be irreducible only if d is odd:

```
d:   (irred. poly.)
1:    x^3 + x + 1
3:    x^7 + x^5 + x^3 + x + 1
5:    x^11 + x^9 + x^7 + x^5 + x^3 + x + 1
```

The list up to $d=1000$ (sequence A107220 of [214])

```
1, 3, 5, 7, 9, 13, 23, 27, 31, 37, 63, 69, 117, 119, 173, 219, 223,
247, 307, 363, 383, 495, 695, 987,
```

can be obtained (within about 10 minutes) via

```
for(d=1,1000, p=(1+sum(t=0,d,x^(2*t+1))); if(polisirreducible(Mod(1,2)*p),print1(d,", ")))
```

A computational simplification with modular reduction can be observed by noting that

$$\left(1 + x + x^3 + x^5 + \ldots + x^n\right) \cdot \left(1 + x^2\right) \quad = \quad 1 + x + x^2 + x^{n+2} \tag{38.4-8}$$

One does all computations modulo the product (with cheap reductions) and only reduces the final result modulo the alternating polynomial.

38.4.3.12 Primitive polynomials with uniformly distributed coefficients

Primitive polynomials with (roughly) equally spaced coefficients are given in [197] for degrees from 9 to 660. Polynomials with weight 5 (pentanomials) are given in [FXT: data/eq-primpoly-w5.txt], the polynomials around degree 500 are

```
498 372 247 124 0
499 380 253 125 0
500 378 250 127 0
501 375 255 125 0
502 370 240 121 0
```

The polynomials with weight 7 are given in [FXT: data/eq-primpoly-w7.txt], the list for weight 9 is [FXT: data/eq-primpoly-w9.txt].

38.5 The number of irreducible and primitive polynomials

n:	I_n	n:	I_n	n:	I_n	n:	I_n
1:	2	11:	186	21:	99858	31:	69273666
2:	1	12:	335	22:	190557	32:	134215680
3:	2	13:	630	23:	364722	33:	260300986
4:	3	14:	1161	24:	698870	34:	505286415
5:	6	15:	2182	25:	1342176	35:	981706806
6:	9	16:	4080	26:	2580795	36:	1908866960
7:	18	17:	7710	27:	4971008	37:	3714566310
8:	30	18:	14532	28:	9586395	38:	7233615333
9:	56	19:	27594	29:	18512790	39:	14096302710
10:	99	20:	52377	30:	35790267	40:	27487764474

Figure 38.5-A: The number of irreducible binary polynomials for degrees $n \leq 40$.

n:	P_n	n:	P_n	n:	P_n	n:	P_n
1:	1	11:	176	21:	84672	31:	69273666
2:	1	12:	144	22:	120032	32:	67108864
3:	2	13:	630	23:	356960	33:	211016256
4:	2	14:	756	24:	276480	34:	336849900
5:	6	15:	1800	25:	1296000	35:	929275200
6:	6	16:	2048	26:	1719900	36:	725594112
7:	18	17:	7710	27:	4202496	37:	3697909056
8:	16	18:	7776	28:	4741632	38:	4822382628
9:	48	19:	27594	29:	18407808	39:	11928047040
10:	60	20:	24000	30:	17820000	40:	11842560000

Figure 38.5-B: The number of primitive binary polynomials for degrees $n \leq 40$.

The number of irreducible binary polynomials of degree n is

$$I_n \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(d)\,2^{n/d} \;=\; \frac{1}{n}\sum_{d\backslash n}\mu(n/d)\,2^{d} \tag{38.5-1}$$

The Möbius function μ is defined by relation 35.1-6 on page 657. The expression is identical to the formula for the number of Lyndon words (relation 17.2-2 on page 343). If n is prime then $I_n = \frac{2^n-2}{n}$. Figure 38.5-A for gives I_n $n \le 40$, the sequence is entry A001037 of [214]. The list of all irreducible polynomials up to degree 11 is given in [FXT: data/all-irredpoly.txt].

The number of primitive binary polynomials of degree n equals

$$P_n \;=\; \frac{\varphi(2^n-1)}{n} \tag{38.5-2}$$

If n is the exponent of a Mersenne prime we have $P_n = \frac{2^n-2}{n} = I_n$. The values of P_n for $n \le 40$ are shown in figure 38.5-B. The sequence is entry A011260 of [214]. The list of all irreducible polynomials up to degree 11 is given in [FXT: data/all-primpoly.txt].

n:	D_n	n:	D_n	n:	D_n	n:	D_n
1:	1	11:	10	21:	15186	31:	0
2:	0	12:	191	22:	70525	32:	67106816
3:	0	13:	0	23:	7762	33:	49284730
4:	1	14:	405	24:	422390	34:	168436515
5:	0	15:	382	25:	46176	35:	52431606
6:	3	16:	2032	26:	860895	36:	1183272848
7:	0	17:	0	27:	768512	37:	16657254
8:	14	18:	6756	28:	4844763	38:	2411232705
9:	8	19:	0	29:	104982	39:	2168255670
10:	39	20:	28377	30:	17970267	40:	15645204474

Figure 38.5-C: The number of irreducible non-primitive binary polynomials for degrees $n \le 40$.

The difference $D_n := I_n - P_n$ is the number of irreducible non-primitive polynomials (see figure 38.5-C). Whenever n is the exponent of a Mersenne prime we have $D_n = 0$. The complete list of these polynomials up to degree 12 inclusive is given in [FXT: data/all-nonprim-irredpoly.txt].

38.6 Generating irreducible polynomials from necklaces

That the number of length-n Lyndon words (see section 17.2 on page 343) is equal to the number of degree-n irreducible polynomials is not a coincidence. Indeed, [73] gives an algorithm that, given a primitive polynomial, generates an irreducible polynomial from a Lyndon word: Let b be a Lyndon word, c be a irreducible polynomial of degree n and a an element of maximal order modulo c. Set $e = a^b$ and compute the polynomial $p(x)$ over GF(2^n), defined as

$$p(x) \;:=\; \left(x-e\right)\left(x-e^2\right)\left(x-e^4\right)\left(x-e^8\right)\cdots\left(x-e^{2^{n-1}}\right) \tag{38.6-1}$$

Then all coefficients of $p(x)$ are either zero or one and the polynomial is irreducible over GF(2).

An implementation in C++ is given in [FXT: `class necklace2bitpol` in bpol/necklace2bitpol.h]:

```
class necklace2bitpol
{
public:
    ulong p_[BITS_PER_LONG+1];  // polynomial over GF(2**n_)
    ulong n_;  // degree of c_
    ulong c_;  // modulus (irreducible polynomial)
```

```
      b             e        p                    b             e        p
   ....1 4 m     ...1.    .11..1 P             ....1 4 m     ...1.    .11..1 P
   ...1. 4       ..1..    .11..1 P             ...11 4 m     .1...    .11111
   ...11 4 m     .1...    .11111               ..1.1 2 m     .1.11    .1.1.1 red.
   ..1.. 4       .1..1    .11..1 P             ..111 4 m     ..111    .1..11 P
   ..1.1 2 m     .1.11    .1.1.1 red.          .1111 1 m     ....1    .1...1 red.
   ..11. 4       .1111    .11111
   ..111 4 m     ..111    .1..11 P
   .1... 4       .111.    .11..1 P
   .1..1 4       ..1.1    .11111
   .1.1. 2       .1.1.    .1.1.1 red.
   .1.11 4       .11.1    .1..11 P
   .11.. 4       ...11    .11111
   .11.1 4       ..11.    .1..11 P
   .111. 4       .11..    .1..11 P
   .1111 1 m     ....1    .1...1 red.
```

Figure 38.6-A: Characteristic polynomials of the powers $e = x^b$ of the primitive element x modulo $c = x^4 + x^3 + 1$ (left). If only necklaces are used as exponents b each polynomial is obtained only once (right). Irreducible polynomials are obtained for aperiodic necklaces.

```
    ulong h_;  // mask used for computation
    ulong a_;  // generator modulo c
    ulong e_;  // a^b
public:
    necklace2bitpol(ulong n, ulong c=0, ulong a=0)
        : n_(n), c_(c), a_(a)
    {
        if ( 0==c )  c_ = lowbit_primpoly[n];
        if ( 0==a )  a_ = 2UL;  // 'x'
        h_ = (highest_bit(c_) >> 1);
    }
    ~necklace2bitpol()  { ; }
    ulong poly(ulong b)
    {
        for (ulong k=0; k<=n_; ++k)  p_[k] = 0;
        p_[0] = 1;
        ulong e = bitpolmod_power(a_, b, c_, h_);
        e_ = e;  // for reference
        for (ulong d=1; d<=n_; ++d)
        {
            for (ulong k=d; k; --k)  p_[k] = p_[k-1];
            p_[0] = 0;
            for (ulong k=0; k<d; ++k)
            {
                p_[k] ^= bitpolmod_mult( p_[k+1], e, c_, h_);
            }
            e = bitpolmod_square(e, c_, h_);
        }
        ulong p2 = 0;
        for (ulong j=0; j<=n_; ++j)  p2 |= (p_[j] << j);
        return p2;
    }
};
```

Figure 38.6-A (left) shows all polynomials that are generated with $c = x^4+x^3+1$ and the generator $a = x$. This is the output of [FXT: gf2n/necklace2irred-demo.cc]. The columns are: b and its cyclic period (an 'm' appended if the word is the cyclic minimum), e and p where a 'P' indicates that p is primitive. Observe that cyclic shifts of the same word give identical polynomials p. Further, if the period is not maximal then p is reducible. Restricting our attention to the necklaces b we obtain each polynomial just once (right of figure 38.6-A). The Lyndon words b give all degree-n irreducible polynomials. The primitive polynomials are exactly those where $\gcd(b,\, 2^n - 1) = 1$.

To generate all irreducible binary polynomials of fixed degree use [FXT: `class all_irredpoly` in bpol/all-irredpoly.h]. The usage is shown in [FXT: gf2n/all-irredpoly-demo.cc]. For $n = 24$ one obtains $I_{24} = 698,870$ irreducible polynomials (among them are $P_{24} = 276,480$ primitive polynomials).

degree:	necklaces	search
24:	49 k/sec	147 k/sec
35:	17 k/sec	64 k/sec
45:	8 k/sec	36 k/sec
63:	3 k/sec	18 k/sec

Figure 38.6-B: Rate of generation of irreducible polynomials via necklaces and with exhaustive search.

The computation takes about 15 seconds. It turns out that the generation via exhaustive search [FXT: gf2n/bitpol-search-irred-demo.cc] is actually faster, figure 38.6-B gives the rates of generation for various degrees and both methods.

38.7 Irreducible and cyclotomic polynomials *

The primitive binary polynomials of degree n can be obtained by factoring the cyclotomic polynomial (see section 35.1.1 on page 655) Y_N over GF(2) where $N = 2^n - 1$. For example, with $n = 6$,

```
? n=6; N=2^n-1; lift( factormod(polcyclo(N),2) )
    [x^6 + x + 1  1]
    [x^6 + x^4 + x^3 + x + 1  1]
    [x^6 + x^5 + 1  1]
    [x^6 + x^5 + x^2 + x + 1  1]
    [x^6 + x^5 + x^3 + x^2 + 1  1]
    [x^6 + x^5 + x^4 + x + 1  1]
```

We use a routine (`pcfprint(N)`) that prints the N-th cyclotomic polynomial and its factors in symbolic form. With $n = 6$, $N = 2^n - 1 = 63$ we obtain

```
? n=6; N=2^n-1;
? pcfprint(N)
    63: [ 36 33 27 24 18 12 9 3 0 ]
     [ 6 1 0 ]
     [ 6 4 3 1 0 ]
     [ 6 5 0 ]
     [ 6 5 2 1 0 ]
     [ 6 5 3 2 0 ]
     [ 6 5 4 1 0 ]
```

The irreducible but non-primitive binary polynomials are factors of cyclotomic polynomials Y_d where $d \backslash N$, $d < N$ and the order of 2 modulo d equals n:

```
? fordiv(N,d,if(n==znorder(Mod(2,d)) && (d<N), pcfprint(d) ));
     9: [ 6 3 0 ]
        [ 6 3 0 ]
    21: [ 12 11 9 8 6 4 3 1 0 ]
        [ 6 4 2 1 0 ]
        [ 6 5 4 2 0 ]
```

The number of factors of Y_d equals $\varphi(d)/n$ so we can count how many degree-n irreducible polynomials correspond to which divisor of $N = 2^n - 1$:

```
     1:  [1:1]    1
     2:  [3:1]    1
     3:  [7:2]    2
     4:  [5:1]  [15:2]    3
     5:  [31:6]    6
     6:  [9:1]  [21:2]  [63:6]    9
     7:  [127:18]    18
     8:  [17:2]  [51:4]  [85:8]  [255:16]    30
     9:  [73:8]  [511:48]    56
    10:  [11:1]  [33:2]  [93:6]  [341:30]  [1023:60]    99
    11:  [23:2]  [89:8]  [2047:176]    186
```

Line 6 tells us that one irreducible polynomial of degree 6 is due to the factor 9, two are due to the factor 21 and the 6 primitive polynomials correspond to $N = 63$ itself which we have verified a moment ago. Further, the a polynomials corresponding to an entry `[d:a]` all have order d. The list was produced using

```
{ for (n=1, 11,
    print1(n,": ");
    s = 0;
    N = 2^n-1;
    fordiv (N, d,
        if ( n==znorder(Mod(2,d)) ,
            a = eulerphi(d)/n;
            print1(" [",d,":",a,"] ");
            s += a;
        );
    );
    print("   ",s);
 ); }
```

38.8 Factorization of binary polynomials

This section treats the factorization of binary polynomials. The first part describes how to factorize polynomials that do not contain a square factor. The second part gives algorithms to detect and remove square factors. Finally, an algorithm to factorize arbitrary binary polynomials is given.

38.8.1 Factorization of squarefree polynomials

$c = x^7 + x + 1$ (irreducible):

```
 Q=          Q-id=          nullspace=
1......     .......           1......
...1..     .1..1..
.1..1.     .11.1..
....1.     ...1.1.
..1..1     ..1.11.
......1    .....11
...1..1    ...1...
```

$c = x^7 + x^3 + x + 1 = (x+1)(x^2+x+1)(x^4+x+1)$:

```
 Q=          Q-id=             nullspace=
1......     .......           1....
....1.1     .1..1.1           .1..1.
.1..1.1     .11.1.1           .1.1.11
....1.      ...1.1.
..1.111     ..1..11
......1     .....11
...1.11     ...1.1.
```

$c = (1+x)^7 = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ (not square-free):

```
 Q=          Q-id=
1...1..     ....1..           nullspace=
.......     .1....            1......
.1...1.     .11..1.
.......     ...1...
..1...1     ..1.1.1
.......     .....1.
...1...     ...1..1
```

Figure 38.8-A: The Q-matrices for three binary polynomials and the nullspaces of $Q - \mathrm{id}$.

In order to factorize a polynomial that must not contain a square factor we will use *Berlekamp's Q-matrix algorithm*. The algorithm consists of two main steps: the computation of the nullspace of a matrix, and a refinement phase that finds the distinct irreducible factors.

Let c be a binary polynomial of degree d. The Q-matrix is a $d \times d$ matrix whose n-th column can be computed as the binary polynomial $x^{2\,n} \pmod c$. The algorithm will use the nullspace of $Q - \mathrm{id}$.

The routine to compute the Q-matrix is [FXT: `setup_q_matrix()` in bpol/berlekamp.cc]:

```
void
setup_q_matrix(ulong c, ulong d, ulong *ss)
```

```
// Compute the Q-matrix for the degree-d polynomial c.
// Used in Berlekamp's factorization algorithm.
{
    ulong h = 1UL << (d-1);
    {
        ulong x2 = 2UL;  // == 'x'
        ulong q = 1UL;
        x2 = bitpolmod_mult(x2, x2, c, h);
        for (ulong k=0;  k<d;  ++k)
        {
            ss[k] = q;
            q = bitpolmod_mult(q, x2, c, h);
        }
        bitmat_transpose(ss, d, ss);
    }
}
```

For the irreducible binary polynomial $c = x^7 + x + 1$ we obtain what is shown at the top of figure 38.8-A. This is the output of the program [FXT: gf2n/qmatrix-demo.cc]. The vector $n_0 = [1, 0, \ldots, 0]$ lies in the nullspace of $Q - \mathrm{id}$ for every polynomial c. For $c = x^7 + x^3 + x + 1 = (x+1)\,(x^2+x+1)\,(x^4+x+1)$ we obtain a nullspace of rank three (middle of figure 38.8-A). In fact, the rank r of the nullspace equals the number of distinct irreducible factors of c. That is, for polynomials containing a square factor we do not get the total number of factors. For example, with $c = (1+x)^7 = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ we obtain what is shown at the bottom of figure 38.8-A.

The vectors spanning the nullspace must be post-processed in order to obtain the irreducible factors of c. The algorithm can be described as follows: let F be a set of binary polynomials whose product equals c, the refinement step R_i proceeds as follows:
Let t be the i-th element of the nullspace. For each element $f \in F$ do the following: if the degree of f equals one, keep it in the set, else delete f from the set and add from the set $X = \{\gcd(f,t), \gcd(f,t+1)\}$ those elements whose degrees are greater or equal to one.

One starts with $F = \{c\}$ and does the refinement steps $R_0, R_1, \ldots\ R_{r-1}$ corresponding to the vectors of the nullspace. Afterwards the set F will contain exactly the distinct irreducible factors of c.

This is implemented in the routine [FXT: `bitpol_refine_factors()` in bpol/berlekamp.cc]:

```
ulong
bitpol_refine_factors(ulong *f, ulong nf, const ulong *nn, ulong r)
// Given the nullspace nn[0,...,r-1] of (Q-id)
// and nf factors f[0,...,nf-1] whose product equals c
// (typically nf=1 and f[0]==c)
// then get all r irreducible factors of c.
{
    ulong ss[r];
    for (ulong j=0; j<r; ++j) // for all elements t in nullspace
    {
        ulong t = nn[j];

        // skip trivial elements in nullspace:
        if ( bitpol_deg(t)==0 )  continue;

        ulong sc = 0;
        for (ulong b=0; b<nf; ++b)  // for all elements bv in set
        {
            ulong bv = f[b];
            ulong db = bitpol_deg(bv);
            if ( db <= 1 )  // bv cannot be reduced
            {
                ss[sc++] = bv;
            }
            else
            {
                for (ulong s=0; s<2; ++s)  // for all elements in GF(2)
                {
                    ulong ti = t ^ s;
                    ulong g = bitpol_gcd(bv, ti);
                    if ( bitpol_deg(g) >= 1 )  ss[sc++] = g;
                }
            }
```

```
        }
        nf = sc;
        for (ulong k=0; k<nf; ++k)  f[k] = ss[k];
        if ( nf>=r )  break;  // nf>r would be an error
    }
    return nf;
}
```

As can be seen, one can skip elements corresponding to constant polynomials. Further, as soon as the set F contains r elements, all factors are found and the algorithm terminates.

Now Berlekamp's algorithm can be implemented as

```
ulong
bitpol_factor_squarefree(ulong c, ulong *f)
// Fill irreducible factors of squarefree polynomial c into f[]
// Return number of factors.
{
    ulong d = bitpol_deg(c);
    if ( d<=1 )  // trivial cases:  0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c )  d = 1;  // 0==0^1
        return d;
    }
    ulong ss[d];
    setup_q_matrix(c, d, ss);
    bitmat_add_unit(ss, d);
    ulong nn[d];
    ulong r = bitmat_nullspace(ss, d, nn);

    f[0] = c;
    ulong nf = 1;
    if ( r>1 )  nf = bitpol_refine_factors(f, nf, nn, r);
    return r;
}
```

The algorithm for the computation of the nullspace was taken from [155], find the implementation in [FXT: bmat/bitmat-nullspace.cc].

Berlekamp's algorithm is given in [83] in a more general form: In order to factorize a polynomial with coefficients in the finite field GF(q) set up the Q-matrix with columns $x^{q\,i}$ and in the refinement step set $X = \{\gcd(f, t+0), \gcd(f, t+1), \ldots, \gcd(f, t+(q-1))\}$. The algorithm is efficient only if q is small.

38.8.2 Extracting the squarefree part of a polynomial

To test whether a polynomial c has a square factor one computes $g = \gcd(c, c')$ where c' is the derivative. If $g \neq 1$ then c has the square factor g: let $c = a \cdot b^2$, then $c' = a'\,b^2 + a\,2\,b\,b' = a'\,b^2$, so $\gcd(c, c') = b^2$. The corresponding routine for binary polynomials is given in [FXT: bpol/bitpol-squarefree.h]:

```
inline ulong bitpol_test_squarefree(ulong c)
// Return  0 if polynomial is squarefree
// else return square factor != 0
{
    ulong d = bitpol_deriv(c);
    if ( 0==d )  return  (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return  (1==g ? 0 : g);
}
```

The derivative of a binary polynomial can be computed easily [FXT: bpol/bitpol-deriv.h] (64-bit version):

```
inline ulong bitpol_deriv(ulong c)
// Return derived polynomial
{
    c &= 0xaaaaaaaaaaaaaaaaUL;
```

```
    return  (c>>1);
}
```

The coefficients at the even powers have to be deleted because derivation multiplies them with an even factor which equals zero modulo two.

If the derivative of a binary polynomial is zero, then it is a perfect square or a constant polynomial:

```
inline ulong bitpol_pure_square_q(ulong c)
// Return whether polynomial is a pure square != 1
{
    if ( 1UL==c )  return 0;
    c &= 0xaaaaaaaaaaaaaaaaUL;
    return  (0==c);
}
```

The following routine returns zero if c is squarefree or equal to one. If c is has a square factor $s \neq 1$ then s is returned:

```
inline ulong bitpol_test_squarefree(ulong c)
{
    ulong d = bitpol_deriv(c);
    if ( 0==d )  return  (1==c ? 0 : c);
    ulong g = bitpol_gcd(c, d);
    return  (1==g ? 0 : g);
}
```

In case a polynomial is a perfect square then its square root can be computed as

```
inline ulong bitpol_pure_sqrt(ulong c)
{
    ulong t = 0;
    for (ulong mc=1,mt=1;  mc;  mc<<=2,mt<<=1)
    {
        if ( mc & c )  t |= mt;
    }
    return  t;
}
```

For the factorization algorithm for general polynomials we have to extract the product of all distinct irreducible factors (the squarefree part) from a polynomial. The routine [FXT: `bitpol_sreduce()` in bpol/bitpol-squarefree.cc] returns a polynomial where the even exponents are reduced:

```
ulong
bitpol_sreduce(ulong c)
{
    ulong s = bitpol_test_squarefree(c);
    if ( 0==s )  return c;  // c is squarefree

    ulong f = bitpol_div(c, s);

    do  // here s is a pure square and s>1
    {
        s = bitpol_pure_sqrt(s);
    }
    while ( bitpol_pure_square_q(s) );

    ulong g = bitpol_gcd(s, f);
    s = bitpol_div(s, g);
    f = bitpol_mult(f, s);

    return f;
}
```

With $c = f \cdot s^{k\,2^t}$ (k odd, the factors of f and s not necessarily distinct) the returned polynomial

equals $f \cdot s^k$. Some examples:

$$\begin{aligned} a^2 &\mapsto a \\ a^4 &\mapsto a \\ a^3 = a\,a^2 &\mapsto a\,a \\ a^5 = a\,a^4 &\mapsto a\,a \\ a\,b^2 &\mapsto a\,b \\ a\,b\,b^2 &\mapsto a\,b\,b = a\,b^2 \mapsto a\,b \\ f \cdot s^{k\,2^t} &\mapsto f \cdot s^k \end{aligned}$$

To extract the squarefree part of a polynomial call the routine repeatedly until the returned polynomial equals the input:

```
inline ulong bitpol_make_squarefree(ulong c)
{
    ulong z = c, t;
    while ( z!=(t=bitpol_sreduce(z)) )  z = t;
    return z;
}
```

The reduction routine will be called at most $\log_2(n)$ times for a polynomial of degree n: the worst case is a perfect power $p = a^{2^k-1}$ where $2^k - 1 \le n$. Observe that $(2^k - 1) = 1 + 2\,(2^{k-1} - 1)$, so the reduction routine will split p as $p = a\,s^2 \mapsto a\,s$ where $s = a^{2^{k-1}-1}$ is of the same form.

38.8.3 Factorization of arbitrary polynomials

The factorization routine for arbitrary binary polynomials [FXT: `bitpol_factor()` in bpol/bitpol-factor.cc] extracts the squarefree part f of its input c, uses Berlekamp's algorithm to factor f and updates the exponents according to the polynomial $s = c/f$:

```
ulong
bitpol_factor(ulong c, ulong *f, ulong *e)
// Factorize the binary polynomial c:
// c = \prod_{i=0}^{fct-1}{f[i]^e[i]}
// The number of factors (fct) is returned.
{
    ulong d = bitpol_deg(c);
    if ( d<=1 )  // trivial cases:  0, 1, x, x+1
    {
        f[0] = c;
        if ( 0==c )  d = 1;  // 0==0^1
        return d;
    }

    // get squarefree part:
    ulong cf = bitpol_make_squarefree(c);

    // ... and factor it:
    ulong fct = bitpol_factor_squarefree(cf, f);

    // All exponents are one:
    for (ulong j=0; j<fct; ++j)  { e[j] = 1; }

    // Here f[],e[] is a valid factorization of the squarefree part cf

    // Update exponents with square part:
    ulong cs = bitpol_div(c, cf);
    for (ulong j=0; j<fct; ++j)
    {
        if ( 1==cs )  break;
        ulong fj = f[j];
        ulong g = bitpol_gcd(cs, fj);
        while ( 1!=g )
        {
            ++e[j];
            cs = bitpol_div(cs, fj);
```

```
            if ( 1==cs )  break;
            g = bitpol_gcd(cs, fj);
        }
    }
    return fct;
}
```

As given the algorithm makes just one call to the routine that computes a nullspace.

```
 0:    x^5                   ==   (x)^5
 1:    x^5                +1 ==   (x+1) *  (x^4+x^3+x^2+x+1)
 2:    x^5             +x    ==   (x) *  (x+1)^4
 3:    x^5             +x+1  ==   (x^3+x^2+1) *  (x^2+x+1)
 4:    x^5         +x^2      ==   (x)^2 *  (x+1) *  (x^2+x+1)
 5:    x^5         +x^2  +1  ==   (x^5+x^2+1)
 6:    x^5         +x^2+x    ==   (x) *  (x^4+x+1)
 7:    x^5         +x^2+x+1  ==   (x+1)^2 *  (x^3+x+1)
 8:    x^5    +x^3           ==   (x)^3 *  (x+1)^2
 9:    x^5    +x^3       +1  ==   (x^5+x^3+1)
10:    x^5    +x^3    +x    ==   (x) *  (x^2+x+1)^2
11:    x^5    +x^3    +x+1  ==   (x+1) *  (x^4+x^3+1)
12:    x^5    +x^3+x^2      ==   (x)^2 *  (x^3+x+1)
13:    x^5    +x^3+x^2  +1  ==   (x+1)^3 *  (x^2+x+1)
14:    x^5    +x^3+x^2+x    ==   (x) *  (x+1) *  (x^3+x^2+1)
15:    x^5    +x^3+x^2+x+1  ==   (x^5+x^3+x^2+x+1)
16:    x^5+x^4              ==   (x)^4 *  (x+1)
17:    x^5+x^4           +1 ==   (x^2+x+1) *  (x^3+x+1)
18:    x^5+x^4         +x    ==   (x) *  (x^4+x^3+1)
19:    x^5+x^4         +x+1  ==   (x+1)^5
20:    x^5+x^4    +x^2       ==   (x)^2 *  (x^3+x^2+1)
21:    x^5+x^4    +x^2  +1   ==   (x+1) *  (x^4+x+1)
22:    x^5+x^4    +x^2+x     ==   (x) *  (x+1)^2 *  (x^2+x+1)
23:    x^5+x^4    +x^2+x+1   ==   (x^5+x^4+x^2+x+1)
24:    x^5+x^4+x^3           ==   (x)^3 *  (x^2+x+1)
25:    x^5+x^4+x^3       +1  ==   (x+1)^2 *  (x^3+x^2+1)
26:    x^5+x^4+x^3    +x     ==   (x) *  (x+1) *  (x^3+x+1)
27:    x^5+x^4+x^3    +x+1   ==   (x^5+x^4+x^3+x+1)
28:    x^5+x^4+x^3+x^2       ==   (x)^2 *  (x+1)^3
29:    x^5+x^4+x^3+x^2  +1   ==   (x^5+x^4+x^3+x^2+1)
30:    x^5+x^4+x^3+x^2+x     ==   (x) *  (x^4+x^3+x^2+x+1)
31:    x^5+x^4+x^3+x^2+x+1   ==   (x+1) *  (x^2+x+1)^2
```

Figure 38.8-B: Factorizations of the binary polynomials of degree 5.

Figure 38.8-B shows the factorizations of the binary polynomials of degree 5, it was created with the program [FXT: gf2n/bitpolfactor-demo.cc]. Factoring the first million polynomials of degrees 20, 30, 40 and 60 takes about 5, 10, 15 and 30 seconds, respectively.

A variant of the factorization algorithm often given uses the so-called *squarefree factorization* $c = \prod_i a_i^i$ where the polynomials a_i are squarefree and pairwise coprime. Given the squarefree factorization one has to call the core routine for each non-trivial a_i.

As noted, the refinement step becomes expensive if the coefficients are in a field GF(q) where q is not small because q computations of the polynomial gcd are involved. For an algorithm that is efficient also for large values of q see [83] or [119, ch.14].

Chapter 39

Shift registers

We describe shift register sequences (SRS) and their generation via linear feedback shift registers (LFSR). The underlying mechanism is the modular arithmetic of binary polynomials described in section 38.3. We give an expression for the number of shift registers sequences of maximal length, the so-called m-sequences. Two related mechanisms, feedback carry shift registers (FCSR) and linear hybrid cellular automata (LHCA), are described. Most of the algorithms given can easily be implemented in hardware.

39.1 Linear feedback shift registers (LFSR)

```
      c = 1..11 == 0x13 == 19  (deg = 4)
       0    w =   15  = 1111  1  ...1 =    1 = a
       1    w =   14  = 111.  .  ..1. =    2 = a
       2    w =   12  = 11..  .  .1.. =    4 = a
       3    w =    8  = 1...  .  1... =    8 = a
       4    w =    1  = ...1  1  ..11 =    3 = a
       5    w =    2  = ..1.  .  .11. =    6 = a
       6    w =    4  = .1..  .  11.. =   12 = a
       7    w =    9  = 1..1  1  1.11 =   11 = a
       8    w =    3  = ..11  1  .1.1 =    5 = a
       9    w =    6  = .11.  .  1.1. =   10 = a
      10    w =   13  = 11.1  1  .111 =    7 = a
      11    w =   10  = 1.1.  .  111. =   14 = a
      12    w =    5  = .1.1  1  1111 =   15 = a
      13    w =   11  = 1.11  1  11.1 =   13 = a
      14    w =    7  = .111  1  1..1 =    9 = a
   >>  0    w =   15  = 1111  1  ...1 =    1 = a  << period restarts
       1    w =   14  = 111.  .  ..1. =    2 = a
       2    w =   12  = 11..  .  .1.. =    4 = a
       3    w =    8  = 1...  .  1... =    8 = a
       4    w =    1  = ...1  1  ..11 =    3 = a
       5    w =    2  = ..1.  .  .11. =    6 = a
```

Figure 39.1-A: Linear feedback shift register using the primitive polynomial $C = x^4 + x + 1$.

Multiplication of a binary polynomial A by x modulo a polynomial C is particularly easy as shown near the beginning of section 38.3 on page 805: Simply shift the input to the left (multiplication) and if the result $A \cdot x$ has the same degree as C then subtract (XOR) the polynomial C (modular reduction).

The underlying mechanism of shifting and conditionally feeding back certain bits is called a *linear feedback shift register* (LFSR). A *shift register sequence* (SRS) can be obtained by computing $A_k = x^k$, $k = 0, 1, \ldots, 2^n - 1$ modulo C and setting bit k of the SRS to the least significant bit of A_k. In the context of LFSRs the polynomial C is sometimes called the *connection polynomial* of the shift register.

When the modulus C is a primitive polynomial (see section 38.4 on page 808) of degree n then the SRS is a sequence of zeros and ones that contains all nonzero words of length n. Further, if a word W is

updated at each step by left shifting and adding the bit of the SRS then this sequence also contains all nonzero words.

This is demonstrated in [FXT: gf2n/lfsr-demo.cc], which for $n = 4$ uses the primitive polynomial $C = x^4 + x + 1$ and gives the output shown in figure 39.1-A. Here we pasted the first few lines after the end of the actual output to emphasize the periodicity of the sequences. The corresponding SRS of period 15 is (extra spaces to mark start of period):

```
1 0 0 0 1 0 0 1 1 0 1 0 1 1 1    1 0 0 0 1 0 0 1 1 0 1 0 1 1 1    1 0 ...
```

In fact any of the bits of the words $A_k = x^k \bmod C$ (or linear combination of two or more bits) could be used, each producing a cyclically shifted version of the SRS.

The cheapest way to generate a SRS is to compute the negative powers x^{-k} modulo C, that is, to repeatedly divide by x:

```
ulong c = a primitive polynomial;
ulong n = degree of C;
ulong a = 1;
for (ulong k=0; k<n; ++k)
{
    ulong s = a & 1;
    // Use s here.
    if ( s )  a ^= c;
    a >> 1;
}
```

The routine will work for $\deg(C) < n$ where n is the number of bits in a machine word. A version that also works for $\deg(C) = n$ is given in section 38.3 on page 805.

A C++ class implementing a LFSR with a few convenient extras is [FXT: `class lfsr` in bpol/lfsr.h]:

```
class lfsr
// (binary) Linear Feedback Shift Register
// Produces a shift register sequence (SRS)
// generated by  a_k=x^k (mod c) where
// c is a primitive polynomial of degree n.
// The period of SRS is 2^n - 1
// (non-primitive c lead to smaller periods)
{
public:
    ulong a_;  // internal state (polynomial modulo c)
    ulong w_;  // word of the shift_register_sequence (SRS)
    ulong c_;  // (mod 2) poly  e.g. x^4+x+1 == 0x13 == 1..11
    ulong h_;  // highest bit in SRS word  e.g. (above) == 16 = 1...
    ulong mask_;  // mask  e.g. (above) == 15 == 1111
    ulong n_;  // degree of polynomial  e.g. (above) == 4
public:
    lfsr(ulong n, ulong c=0)
    // n: degree of polynomial c
    // c: polynomial (defaults to minimum weight primitive polynomial)
    {
 [--snip--]
```

The crucial computation is implemented as

```
ulong next()
{
    ulong s = a_ & h_;
    a_ <<= 1;
    w_ <<= 1;
    if ( 0!=s )
    {
        a_ ^= c_;
        w_ |= 1;
    }
    w_ &= mask_;
    return w_;
}
```

Up to the lines that update the word `w_` this function is identical to `bitpolmod_times_x()` given in section 38.3 on page 805.

The method `next_w()` skips to the next word by calling `next()` n times:

```
    ulong next_w()
    {
        for (ulong k=0; k<n_; ++k)  next();
        return w_;
    }
```

Let a and w a pair of values that correspond to each other. The following two methods directly set one of these two while keeping the pair consistent. Setting a to a given value:

```
    void set_a(ulong a)
    {
        a_ = a;
        w_ = 0;
        ulong b = 1;
        for (ulong j=0; j<n_; ++j)
        {
            if ( a & 1 )
            {
                w_ |= b;
                a ^= c_;
            }
            b <<= 1;
            a >>= 1;
        }
    }
```

The loop executes n times where n is the degree of the modulus. Setting w to a given value:

```
    void set_w(ulong w)
    {
        w_ = w;
        a_ = 0;
        ulong c = c_;
        while ( w )
        {
            if ( w & 1 )  a_ ^= c;
            c <<= 1;
            w >>= 1;
        }
        a_ &= mask_;
    }
```

The supplied value must be nonzero for both methods.

Going back one step is possible via the method `prev()`

```
public:
    ulong prev()
    {
        prev_a();
        set_a(a_);
        return w_;
    }
```

which calls `prev_a()`:

```
private:
    void prev_a()
    {
        ulong s = a_ & 1;
        a_ >>= 1;
        if ( s )
        {
            a_ ^= (c_>>1);
            a_ |= h_;  // so it works for  n_ == BITS_PER_LONG
        }
    }
```

The method `prev_a()` leaves the value of w inconsistent with a and therefore cannot be called directly. Note that stepping back is more expensive than stepping forward because `set_a()` is rather expensive.

It is also possible to go backwards word-wise:

```
    ulong prev_w()
    {
        for (ulong k=0; k<n_; ++k)  prev_a();
```

```
        set_a(a_);
        return w_;
    }
```

As this routine involves only one call to `set_a()` it is about as expensive as stepping one word forward using `next_w()`.

39.2 Galois and Fibonacci setup

The type of shift registers considered so far is the so-called *Galois setup* of a binary shift register. The mechanism is to detect whether a one is being shifted out and, if so, subtract the polynomial modulus. The left- and right- shift operations can be implemented as

```
ulong galois_left(ulong x, ulong c, ulong h)
{
    ulong s = x & h;
    x <<= 1;
    if ( 0!=s )  x ^= c;
    return x;
}
```

and

```
ulong galois_right(ulong x, ulong c, ulong)
{
    ulong s =  ( x & 1UL );
    x >>= 1;
    if ( s )  x ^= (c>>1);
    return x;
}
```

```
    c = .11..1 == 0x19 == 25  (deg = 4)
    r = .1..11 == 0x13 == 19  (deg = 4)

           -------- Galois -------------          --------- Fibonacci ---------
     k:      Lc      Lr      Rc      Rr            Lc      Lr      Rc      Rr
     1:    ...1    ...1    ...1    ...1          ...1    ...1    ...1    ...1
     2:    ..1.    ..1.    .11.    .1..          ..1.    ..11    .1..    .1..
     3:    .1..    .1..    .1..    .1..          .1..    .111    .11.    .1..
     4:    1...    1...    ...1    1111          1..1    1111    111.    ...1
     5:    1..1    ...1    .11.    111.          ...1    .111    1111    1..1
     6:    .11.    ..11    1.1.    ..11          ..11    .11.    ..11    11..
     7:    1111    .11.    ..1.    .1.1          .11.    1.1.    .1.1    ..11
     8:    ..11    .1.1    .11.    ..1.          .1.1    ..1.    ..1.    .1.1
     9:    .11.    ..1.    ..11    1.11          ..1.    .1.1    .1..    ..1.
    10:    ..1.    .1.1    1111    .1..          .1.1    ..1.    .1.1    .1.1
    11:    .1.1    ..11    .1.1    ..1.          ..11    .1..    ..1.    .1.1
    12:    .1.1    .11.    .1..    ...1          1111    1..1    ...1    .11.
    13:    ...1    1111    1...    1...          .11.    ...1    1..1    1111
    14:    ..1.    .1.1    ..1.    ..1.          .1..    ..1.    ..1.    ..11
    15:    .1..    1..1    ...1    ...1          1...    1...    ...1    ...1
    16:    ....    ....    ....    ....          ....    ....    ....    ....
```

Figure 39.2-A: Sequences of words generated with the Galois- and Fibonacci- mechanisms, either with the left- or the right- shift (capital letters '`L`' and '`R`' on top of columns) and primitive polynomial '`c`' or its reciprocal '`r`'. Each track of any sequence is a shift register sequence.

Four sequences of binary words that (starting identically and) are generated with either the left- or right-shift and a primitive polynomial or its reciprocal (reversed word) are shown in figure 39.2-A. A different set of sequences shown in the same figure is obtained via the so-called *Fibonacci setup*. In the Fibonacci setup the sum (modulo 2) of bits determined by the used polynomial is shifted in at each step.

The left- and right- shift operations can be implemented as

```
ulong fibonacci_left(ulong x, ulong c, ulong h)
{
    x <<= 1;
    ulong s = parity( x & c );
    if ( 0!=s )  x ^= 1;
    x &= ~(h<<1);  // remove excess bit at high end
    return x;
```

```
}
```

and

```
ulong fibonacci_right(ulong x, ulong c, ulong h)
{
    ulong s = parity( x & c );
    x >>= 1;
    if ( s )  x ^= h;
    return x;
}
```

As the parity computation is expensive on most machines (see section 1.15.1 on page 37) the Galois setup should usually be preferred. The programs [FXT: gf2n/lfsr-fibonacci-demo.cc] and [FXT: gf2n/lfsr-galois-demo.cc] can be used to create the binary patterns shown in figure 39.2-A. With both the polynomial modulus can be specified.

39.3 Generating all revbin pairs

```
polynomial:  c      cr                    c      cr
           1.1111  1111.1               1.1111  1111.1

      k:     x       xr            k:     x       xr
      1:   ...1   1...            17:   11...   ...11
      2:   1.111  111.1            18:   .11..   ..11.
      3:   111..  ..111            19:   ..11.   .11..
      4:   .111.  .111.            20:   ...11   11...
      5:   ..111  111..            21:   1.11.   .11.1
      6:   1.1..  ..1.1            22:   .1.11   11.1.
      7:   .1.1.  .1.1.            23:   1..1.   .1..1
      8:   ..1.1  1.1..            24:   .1..1   1..1.
      9:   1.1.1  1.1.1            25:   1..11   11..1
     10:   111.1  1.111            26:   1111.   .1111
     11:   11..1  1..11            27:   .1111   1111.
     12:   11.11  11.11            28:   1....   ....1
     13:   11.1.  .1.11            29:   .1...   ...1.
     14:   .11.1  1.11.            30:   ..1..   ..1..
     15:   1...1  1...1            31:   ...1.   .1...
     16:   11111  11111            32:   ....1   1.... <--= initial pair
```

Figure 39.3-A: All nonzero 5-bit revbin pairs generated by a LFSR.

Using a primitive polynomial of degree n and its reverse one can generate all nonzero pairs `x` and `revbin(x,n)` as follows [FXT: gf2n/lfsr-revbin-demo.cc]:

```
inline void revbin_next(ulong &x, ulong c, ulong &xr, ulong cr)
// if x and xr are (nonzero) n-bit words that are a revbin pair
// compute the next revbin pair.
// c must be a primitve polynomial, cr its reverse (the reciprocal polynomial).
{
    ulong s =  ( x & 1UL );
    x >>= 1;
    xr <<= 1;
    if ( s )
    {
        x ^= (c>>1);
        xr ^= (cr);
    }
}
```

An equivalent technique for computing the revbin permutation (see section 2.1 on page 85) has been proposed in [188]. Figure 39.3-A shows all nonzero 5-bit revbin pairs generated with the primitive polynomial $c = x^5 + x^3 + x^2 + x + 1$ and its reverse.

39.4 The number of m-sequences and De Bruijn sequences

The shift register sequences generated with a polynomial degree n is of maximal length if the polynomial is primitive. The corresponding shift register sequences are called *m-sequences*.

```
degree = 2
c=111 :  .11

degree = 3
c=1.11 :  ..1.111
c=11.1 :  ..111.1

degree = 4
c=1..11 :  ...1..11.1.1111
c=11..1 :  ...1111.1.11..1

degree = 5
c=1..1.1 :  ....1..1.11..11111...11.111.1.1
c=1111.1 :  ....11..1..11111.111...1.1.11.1
c=11.111 :  ....111..11.11111.1...1..1.1.11
c=1.1111 :  ....1.11.1.1...111.11111..1..11
c=111.11 :  ....11.1.1..1...1.11111.11..111
c=1.1..1 :  ....1.1.111.11...11111..11.1..1

degree = 6
c=1....11 :  .....1....11...1.1..1111.1...111..1..1.11.111.11..11.1.1.111111
c=11..111 :  .....1111..1..1.1.1..11.1....1...1.11.111111.1.111...11..111.11
c=1.11.11 :  .....1.111111..1.1.1...11..1111.111.1.11.1..11.11...1..1....111
c=11.11.1 :  .....111....1..1...11.11..1.11.1.111.1111..11...1.1.1..111111.1
c=111..11 :  .....11.111..11...111.1.111111.11.1...1....1.11..1.1.1..1..1111
c=11....1 :  .....111111.1.1.11..11.111.11.1..1..111...1.1111..1.1...11....1
```

Figure 39.4-A: All m-sequences for $n = 2$, 3, 4, 5, and 6. Dots denote zeros.

We now consider all sequences that are cyclic shifts of each other as the same sequence. For given n there are as many m-sequences as primitive polynomials ($P_n = \varphi(2^n - 1)/n$, see section 38.5 on page 823). These can be generated using the linear feedback shift registers described in section 39.1 on page 833.

One might suspect that the using the powers of other elements than x might lead to additional m-sequences, but this is not the case. Further, the powers of elements of maximal order modulo irreducible non-primitive polynomials do not give additional m-sequences.

The program [FXT: gf2n/all-primpoly-srs-demo.cc] computes all m-sequences for a given n. The output for $n = 2, 3, 4, 5, 6$ is shown in figure 39.4-A.

```
    n = 4  nn = 16            n = 5  nn = 32
    ....1111.11..1.1          .....11111.111..11.1.11...1.1..1
    ....1111.1.11..1          .....11111.111..11.1.11...1..1.1
    ....1111.1..1.11          .....11111.111..11.1.1..1.11...1
    ....1111..1.11.1          .....11111.111..11.1.1..1...1.11
    ....11.1111..1.1          .....11111.111..11.1.1...1...1.11
    ....11.1.1111..1          .....11111.111..11.1..1.1.11...1
    ....11.1..1.1111          .....11111.111..11.1..1...1.1.11
    ....11..1.1111.1           [--snip--]
    ....1.1111.1..11          .....1...11..1.1.11.111.1..11111
    ....1.1111..11.1          .....1...11..1.1.11.1..11111.111
    ....1.11.1..1111          .....1...11..1.1.11.1..111.11111
    ....1.11..1111.1          .....1...11..1.1..11111.11.1.111
    ....1.1..1111.11          .....1...11..1.1..11111.1.11.111
    ....1.1..11.1111          .....1...11..1.1..111.11.1.11111
    ....1..1111.1.11          .....1...11..1.1..111.1.11.11111
    ....1..11.1.1111          total # of DBS found = 2048
total # of DBS found = 16
```

Figure 39.4-B: All De Bruijn sequences for $n = 4$ (left), the first and last sequence correspond to the two m-sequences for $n = 4$. The first and last few De Bruijn sequences for $n = 5$ (right).

When a zero is inserted to the (unique) run of $n - 1$ zeros in an m-sequence then a *De Bruijn sequence* (DBS) is obtained. A DBS contains all binary words including the all-zero word.

For all $n \geq 4$ given there exist more DBSs than m-sequences. For example, for $n = 6$ there are 6 m-sequences and 67,108,864 DBSs. An exhaustive search for all DBSs of given length $L = 2^n$ is possible only for tiny n. The program [FXT: bits/all-dbs-demo.cc] finds all DBSs for $n = 3, 4, 5$. Its output with $n = 4$ and $n = 5$ (partly) is shown in figure 39.4-B.

n :	$L_n = 2^n$	$x = 2^{n-1} - n$	$S_n = 2^x$
1:	2	0	1
2:	4	0	1
3:	8	1	2
4:	16	4	16
5:	32	11	2,048
6:	64	26	67,108,864
7:	128	57	144,115,188,075,855,872
8:	256	120	1,329,227,995,784,915,872,903,807,060,280,344,576

Figure 39.4-C: The number S_n of DBSs of length L_n.

The total number of DBSs equals

$$S_n \quad = \quad 2^x \quad \text{where} \quad x = 2^{n-1} - n \tag{39.4-1}$$

The two DBSs for $n = 3$ are `[...111.1]` and `[...1.111]=[1.111...]`, reversed sequences are considered different by the formula. The first few values of S_n are shown in figure 39.4-C. One has $S_{n+1} = S_n^2 \, L_{n-1}$, equivalently $x_{n+1} = 2\,x_n + n - 1$.

The general formula for the number of length-n base-m DBSs is $S_n = m!^{m^{n-1}} / m^n$, as given in [157]. A graph theoretical proof of the formula for the case $m = 2$ can be found in [170, p.56]. For a more efficient approach to generate all DBSs of given length see section 19.2.2 on page 359.

We note that there are several ways to generalize the idea of the De Bruijn cycles to *universal cycles* as described in [78].

39.5 Auto correlation of m-sequences

```
 1:  [ +-++--+-+----+++ ]   C= +15 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 2:  [ +-++----+-+--+++ ]   C= +15 -1 -1 -1 -1 +3 -5 -1 -1 -5 +3 -1 -1 -1 -1
 3:  [ +-+-++--+----+++ ]   C= +15 -1 -1 -1 -1 -1 +3 -5 -5 +3 -1 -1 -1 -1 -1
 4:  [ +-+-++----+--+++ ]   C= +15 -1 -1 -1 -1 +3 -1 -5 -5 -1 +3 -1 -1 -1 -1
 5:  [ +-+--++----+-+++ ]   C= +15 -1 -1 -1 -1 -5 +3 -1 -1 +3 -5 -1 -1 -1 -1
 6:  [ +-+--+-++----+++ ]   C= +15 -1 -1 -1 -9 -1 +3 +3 +3 +3 -1 -9 -1 -1 -1
 7:  [ +-+----++--+-+++ ]   C= +15 -1 -1 -1 -1 -5 +3 -1 -1 +3 -5 -1 -1 -1 -1
 8:  [ +-+----+-++--+++ ]   C= +15 -1 -1 -1 -1 -1 -5 +3 +3 -5 -1 -1 -1 -1 -1
 9:  [ +--++-+----+-+++ ]   C= +15 -1 -1 -1 -1 -1 -5 +3 +3 -5 -1 -1 -1 -1 -1
10:  [ +--+-++-+----+++ ]   C= +15 -1 -1 -1 -9 +3 -1 +3 +3 -1 +3 -9 -1 -1 -1
11:  [ +--+-+----++-+++ ]   C= +15 -1 -1 -1 -1 +3 -5 -1 -1 -5 +3 -1 -1 -1 -1
12:  [ +--+----++-+-+++ ]   C= +15 -1 -1 -1 -1 +3 -1 -5 -5 -1 +3 -1 -1 -1 -1
13:  [ +----++-+--+-+++ ]   C= +15 -1 -1 -1 -9 -1 +3 +3 +3 +3 -1 -9 -1 -1 -1
14:  [ +----+-++-+--+++ ]   C= +15 -1 -1 -1 -9 +3 -1 +3 +3 -1 +3 -9 -1 -1 -1
15:  [ +----+-+--++-+++ ]   C= +15 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
16:  [ +----+--++-+-+++ ]   C= +15 -1 -1 -1 -1 -1 +3 -5 -5 +3 -1 -1 -1 -1 -1
```

Figure 39.5-A: Cyclic auto correlations for all truncated (signed) De Bruijn sequences for $n = 4$. Only two (the first and the second last) out of the 16 sequences are m-sequences.

We have seen that a De Bruijn sequence (DBS) can be obtained from an m-sequence by inserting a single zero at the longest run of zeros. In the other direction, if we take a DBS and delete a zero from the longest run of zeros then we obtain a sequence of length $N = 2^n - 1$ that contains every n-bit nonzero word. But these sequences are *not* m-sequences in general: most of them can not be obtained with an n-bit LFSR and miss an important property of m-sequences.

For a sequence M of $N-1$ zeros and ones M_k define the sequence S of elements S_k via

$$S_k \;:=\; \begin{cases} +1 & \text{if} \quad M_k = 1 \\ -1 & \text{else} \end{cases} \tag{39.5-1}$$

Then, if M is an length-L m-sequence, we have for the cyclic auto correlation (or *auto correlation function*, ACF) of S

$$C_\tau \;:=\; \sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \;=\; \begin{cases} L & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{39.5-2}$$

where $L = N-1$ and $N = 2^n$. That is, C_0 equals the length of the sequence, all other entries are of minimal absolute value: they cannot be zero because L is odd.

This property does not hold for most of the 'truncated' DBS (where one zero in the single run of n consecutive zeros is removed). Figure 39.5-A shows all (signed) truncated DBS for $n = 4$ and their auto correlations. Only 2 out of the 16 truncated DBS have an auto correlation satisfying relation 39.5-2, these are exactly the m-sequences for $n = 4$.

For odd primes q one can obtain sequences of length $L = q$ whose ACF satisfies

$$\sum_{k=0}^{L-1} S_k\, S_{k+\tau \bmod L} \;=\; \begin{cases} L-1 & \text{if} \quad \tau = 0 \\ -1 & \text{else} \end{cases} \tag{39.5-3}$$

However, the sequences start with a single zero: set $S_0 = 0$ and for $1 \le k < q$ set $S_k = 1$ if k is s square modulo q, else $S_k = -1$. The first three such sequences for primes of the form $4\,k+1$ and their ACFs are:

```
  5:   S: [0, +1, -1, -1, +1]
       C: [4, -1, -1, -1, -1]

 13:   S: [ 0, +1, -1, +1, +1, -1, -1, -1, -1, +1, +1, -1, +1]
       C: [12, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

 17:   S: [ 0, +1, +1, -1, +1, -1, -1, -1, +1, +1, -1, -1, -1, +1, -1, +1, +1]
       C: [16, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

With primes of the form $q = 4\,k+3$ one can construct a sequence of length $L = q$ satisfying relation 39.5-2 by simply setting $S_0 = 1$ ($S_0 = -1$ also works) in the sequence just constructed. The sequences for the first three primes $q = 4\,k+3$ and their ACFs are:

```
  3:   S: [+1, +1, -1]
       C: [ 3, -1, -1]

  7:   S: [+1, +1, +1, -1, +1, -1, -1]
       C: [ 7, -1, -1, -1, -1, -1, -1]

 11:   S: [+1, +1, -1, +1, +1, +1, -1, -1, -1, +1, -1]
       C: [11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

These sequences can be used for the construction of Hadamard matrices, see section 18 on page 347.

39.6 Feedback carry shift register (FCSR)

There is a nice analogue of the LFSR in the modulo world, the *feedback carry shift register* (FCSR). With the LFSR we needed an irreducible ('prime') polynomial C where x has maximal order. The powers of x modulo C did run through all different (nonzero) words. Now take a prime c where 2 has maximal order (that is, 2 is primitive root modulo c, see section 37.4 on page 739). Then the powers of 2 modulo c run through all nonzero values less than c.

The implementation is [FXT: `class fcsr` in bpol/fcsr.h]:

```
          c = 1..1.1  = 37

          0 :   a= .....1 =    1        w= .1..11 =   19
          1 :   a= ....1. =    2        w= 1..11. =   38
          2 :   a= ...1.. =    4        w= ..11.. =   12
          3 :   a= ..1... =    8        w= .11... =   24
          4 :   a= .1.... =   16        w= 11.... =   48
          5 :   a= 1..... =   32        w= 1....., =   32
          6 :   a= .11.11 =   27        w= .....1 =    1
          7 :   a= .1...1 =   17        w= ....11 =    3
          8 :   a= 1...1. =   34        w= ...11. =    6
          9 :   a= .11111 =   31        w= ..11.1 =   13
         10 :   a= .11..1 =   25        w= .11.11 =   27
         11 :   a= ..11.1 =   13        w= 11.111 =   55
         12 :   a= .11.1. =   26        w= 1.111. =   46
         13 :   a= ..1111 =   15        w= .111.1 =   29
         14 :   a= .1111. =   30        w= 111.1. =   58
         15 :   a= .1.111 =   23        w= 11.1.1 =   53
         16 :   a= ..1..1 =    9        w= 1.1.11 =   43
         17 :   a= .1..1. =   18        w= .1.11. =   22
         18 :   a= 1..1.. =   36        w= 1.11.. =   44
         19 :   a= 1...11 =   35        w= .11..1 =   25
         20 :   a= 1....1 =   33        w= 11..11 =   51
         21 :   a= .111.1 =   29        w= 1..111 =   39
         22 :   a= .1.1.1 =   21        w= ..1111 =   15
         23 :   a= ...1.1 =    5        w= .11111 =   31
         24 :   a= ..1.1. =   10        w= 11111. =   62
         25 :   a= .1.1.. =   20        w= 1111.. =   60
         26 :   a= ....11 =    3        w= 111..1 =   57
         27 :   a= ...11. =    6        w= 11..1. =   50
         28 :   a= ..11.. =   12        w= 1..1.. =   36
         29 :   a= .11... =   24        w= ..1... =    8
         30 :   a= ..1.11 =   11        w= .1...1 =   17
         31 :   a= .1.11. =   22        w= 1...1. =   34
         32 :   a= ...111 =    7        w= ...1.1 =    5
         33 :   a= ..111. =   14        w= ..1.1. =   10
         34 :   a= .111.. =   28        w= .1.1.. =   20
         35 :   a= .1..11 =   19        w= 1.1..1 =   41

         36 :   a= .....1 =    1        w= .1..11 =   19 <-- period restarts
         37 :   a= ....1. =    2        w= 1..11. =   38
         38 :   a= ...1.. =    4        w= ..11.. =   12
```

Figure 39.6-A: Successive states of a FCSR with modulus $c = 37$.

```
class fcsr
{
public:
    ulong a_;     // internal state (a_0*2**k modulo c),  1 <= a < c
    ulong w_;     // word of the SRS,  1 <= w <= mask
    ulong c_;     // a prime with primitive root 2, e.g. 37 = 1..1.1
    ulong mask_;  // mask  e.g. (with above)    mask == 63 == 111111
public:
    fcsr(ulong c)
    {
        c_ = c;
        const ulong h = highest_bit(c_);
        mask_ = ( h | (h-1) );
        set_a(1);
    }

    ~fcsr()  { ; }

    ulong next()
    {
        a_ <<= 1;                   // a *= 2
        if ( a_ > c_ )  a_ -= c_;   // reduce mod c

        // update w:
        w_ <<= 1;
        w_ |= (a_ & 1);
        w_ &= mask_;

        return w_;
    }

  [--snip--]

    void set_a(ulong a)
    {
```

```
        w_ = 0;
        ulong t = c_;
        while ( (t>>=1) )
        {
            if ( 0==(a & 1) )  a >>= 1;
            else
            {
                a = (a & c_) + ((a ^ c_) >> 1);
            }
        }
        a_ = a;
        next_w();
    }
    ulong get_a()  const  { return a_; }
    ulong get_w()  const  { return w_; }
};
```

The routine is corresponds to the so-called Galois setup, described (for the LFSR) in section 39.2 on page 836, see also [122]. Figure 39.6-A shows the successive states of a FCSR with modulus $c = 37$. This is the output of [FXT: gf2n/fcsr-demo.cc]. Note that w does not run through all values smaller than c but through a subset of $c - 1$ distinct values smaller than 2^6.

```
     x:  p prime with 2 a primitive root, 2**x < p < 2**(x+1)
     1:  3
     2:  5
     3:  11 13
     4:  19 29
     5:  37 53 59 61
     6:  67 83 101 107
     7:  131 139 149 163 173 179 181 197 211 227
     8:  269 293 317 347 349 373 379 389 419 421 443 461 467 491 509
     9:  523 541 547 557 563 587 613 619 653 659 661 677 701 709 757
         773 787 797 821 827 829 853 859 877 883 907 941 947 1019
    10:  1061 1091 1109 1117 1123 1171 1187 1213 1229 1237 1259 1277
         1283 1291 1301 1307 1373 1381 1427 1451 1453 1483 1493 1499
         1523 1531 1549 1571 1619 1621 1637 1667 1669 1693 1733 1741
         1747 1787 1861 1867 1877 1901 1907 1931 1949 1973 1979 1987
         1997 2027 2029
```

Figure 39.6-B: List of primes $p < 2048$ where 2 is s primitive root.

A list of all primes less than 2048 for which 2 is a primitive root is shown in figure 39.6-B. The shown sequence is entry A001122 of [214]. For further information on the correspondence between LFSR and FCSR see [152].

39.7 Linear hybrid cellular automata (LHCA)

Linear hybrid cellular automata (LHCA) are 1-dimensional cellular automata (with 0 and 1 the only possible states) where two different rules are applied dependent of the position (therefore the 'hybrid' in the name). An implementation in C is [FXT: `lhca_next()` in bpol/lhca.h]:

```
inline ulong lhca_next(ulong x, ulong r, ulong m)
// LHCA := (1-dim) Linear Hybrid Cellular Automaton.
// Return next state (after x) of the LHCA with
// rule (defined by) r:
//   Rule 150 is applied for cells where r is one, rule 90 else.
//   Rule 150 :=  next(x) = x + leftbit(x) + rightbit(x)
//   Rule 90  :=  next(x) = leftbit(x) + rightbit(x)
// Length defined by m:
//   m has to be a burst of the n lowest bits (n: length of automaton)
{
    r &= x;
    ulong t = (x>>1) ^ (x<<1);
    t ^= r;
    t &= m;
    return  t;
}
```

Note that the routine is branch free. Implementation in hardware is trivial as only neighboring bits interact.

The naming convention for the rules is as follows: draw a table of the eight possible states of a cell together with its neighbors then draw the new states below:

```
XXX  XX0  X0X  X00  0XX  0X0  00X  000
 0    X    0    X    X    0    X    0
```

Now read the lower row as a binary number, the result equals $01011010_2 = 90$, so this is rule 90. Rule 150 corresponds to $10010110_2 = 150$:

```
XXX  XX0  X0X  X00  0XX  0X0  00X  000
 X    0    0    X    0    X    X    0
```

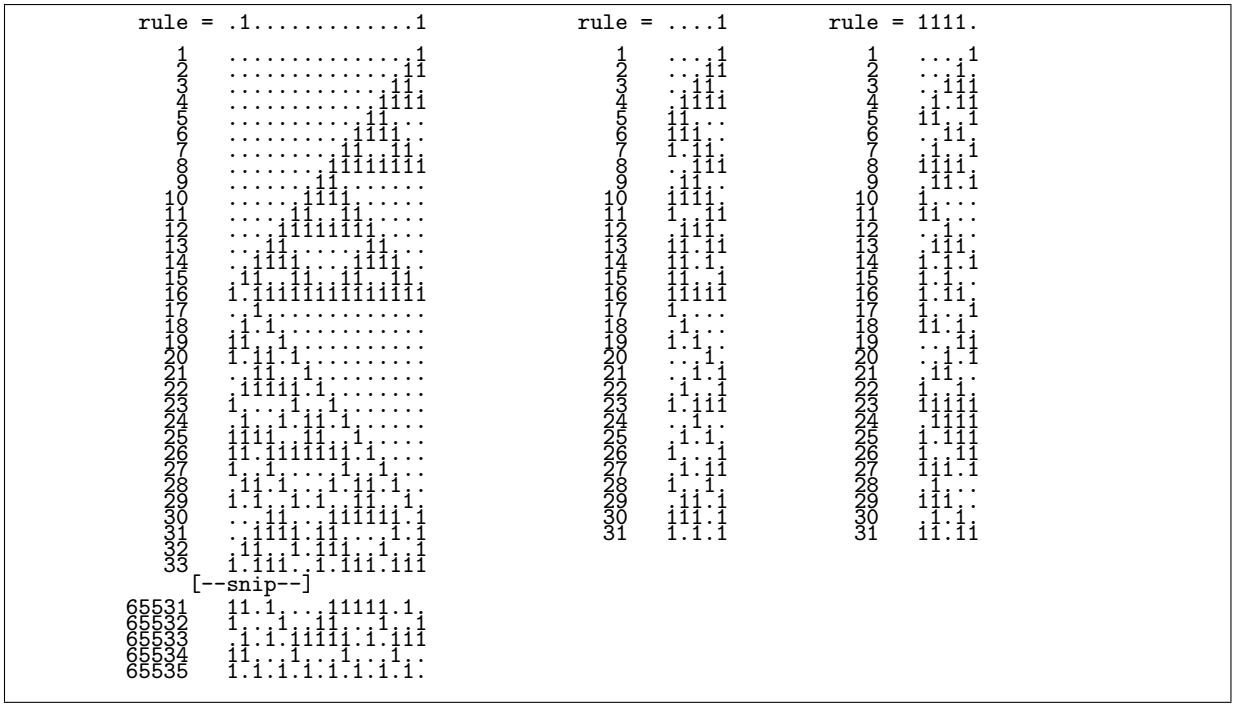


Figure 39.7-A: Partial run of a 16-bit LHCA (left) and complete runs of two 5-bit LHCAs (middle and right). These LHCAs have maximal periods.

A run of successive values for the length-16 weight-2 rule vector $r = 4001_{16}$ starting with 1 is shown on the left side of figure 39.7-A. For certain rule vectors r all $m = 2^n - 1$ nonzero values occur, the period is maximal. This is demonstrated in [FXT: gf2n/lhca-demo.cc], which for $n = 5$ and rule $r = 1$ gives the output shown in the middle of figure 39.7-A. Rule vectors with minimal weight that lead to maximal period are given in [72]. The list [FXT: `minweight_lhca_rule()` in bpol/lhcarule-minweight.cc] is taken from that source:

```
#define R1(n,s1)     (1UL<<s1)
#define R2(n,s1,s2)  (1UL<<s1) | (1UL<<s2)
extern const ulong minweight_lhca_rule[]=
// LHCA rules of minimum weight that lead to maximal period.
{
      0,  // (empty)
 R1(  1,  0),
 R1(  2,  0),
 R1(  3,  0),
 R2(  4,  0, 2),
 R1(  5,  0),
 R1(  6,  0),
 R1(  7,  2),
 R2(  8,  1, 2),
 R1(  9,  0),
 R2( 10,  1, 6),
 R1( 11,  0),
 R2( 12,  2, 6),
```

```
 R1( 13,  4),
 [--snip--]
};
```

Up to $n = 500$ there is always a rule with weight at most 2 that leads to the maximal period. The full list of these rules is given in [FXT: data/minweight-lhca-rules.txt].

39.7.1 Conversion to binary polynomials

To convert a length-n LHCA to a binary polynomial proceed as follows: initialize $p_{-1} := 0$, $p_0 := 1$ and iterate for $k = 1, 2, \ldots, n$:

$$p_k \quad := \quad (x + r_{k-1})\, p_{k-1} + p_{k-2} \tag{39.7-1}$$

where r_i denotes bit i of rule r. The degree of the returned polynomial p_n is n. An implementation of the algorithm is [FXT: `lhca2poly()` in bpol/lhca.h]:

```
inline ulong lhca2poly(ulong r, ulong n)
// Return binary polynomial p that corresponds
//  to the length-n LHCA rule r.
// p has degree n.
{
    ulong p2 = 0,  p1 = 1;
    while ( n-- )
    {
        ulong m = r & 1;
        r >>= 1;
        ulong p = (p1<<1) ^ p2;
        if ( m )  p ^= p1;
        p2 = p1;  p1 = p;

    }
    return  p1;
}
```

```
                          LHCA rule                              polynomial
   1:    r=...............................1  c=..............................11  r= [0]
   2:    r=...............................1  c=.............................111  r= [0]
   3:    r=...............................1  c=............................11.1  r= [0]
   4:    r=.............................1.1  c=...........................1..11  r= [0, 2]
   5:    r=...............................1  c=..........................11.111  r= [0]
   6:    r=...............................1  c=.........................111..11  r= [0]
   7:    r=.............................1..  c=........................11.....1  r= [2]
   8:    r=.............................11.  c=.......................1...111.1  r= [1, 2]
   9:    r=...............................1  c=......................11.111..11  r= [0]
  10:    r=.........................1....1.  c=.....................1...11.1111  r= [1, 6]
  11:    r=...............................1  c=....................11.1...111.1  r= [0]
  12:    r=.........................1...1..  c=...................1.....11111.1  r= [2, 6]
  13:    r=...........................1....  c=..................11..11.....111  r= [4]
  14:    r=...............................1  c=.................111..11......11  r= [0]
  15:    r=.............................1..  c=................11.....1...1...1  r= [2]
  16:    r=.................1.............1  c=...............1..11.111.....111  r= [0, 14]
  17:    r=...........................1....  c=..............11..11..11.1....11  r= [4]
  18:    r=...............1...............1  c=.............1..1..1.1.1.....1.1  r= [0, 16]
  19:    r=.............................1..  c=............11.....111.1....11.1  r= [2]
  20:    r=.............................11.  c=...........1...111.....1...1...1  r= [1, 2]
  21:    r=......................1........1  c=..........1.11..1....1..1.1.1111  r= [0, 9]
  22:    r=...........................1....  c=.........1111..11...1.11.1111.11  r= [4]
  23:    r=...............................1  c=........11.1...1.......111.1...1  r= [0]
  24:    r=....................1...1.......  c=.......1.....1.1...1.1..11111..1  r= [7, 11]
  25:    r=.......................1........  c=......11..11..11.........111..11  r= [8]
  26:    r=...............................1  c=.....111..11.111..........11.111  r= [0]
  27:    r=............1..................1  c=....1.111..1..111....11.1..111.1  r= [0, 19]
  28:    r=.............................1..  c=...111111.1.1111........1.1.1111  r= [2]
  29:    r=...............................1  c=..11.111.....111.............111  r= [0]
  30:    r=...............................1  c=.111..11......11..............11  r= [0]
  31:    r=.....................1..........  c=11.....................1...1...1  r= [10]
```

Figure 39.7-B: Minimum-weight LHCA rules and the corresponding binary polynomials.

The first minimum-weight LHCA rules and their binary polynomials are shown in figure 39.7-B. The table was created by the program [FXT: gf2n/lhca2poly-demo.cc].

For rules of maximal period the polynomials are primitive. In fact there is an isomorphism between the cycle structure of the successive states of an LHCA and its corresponding shift register.

An LHCA rule and its reverse give the identical polynomial.

The polynomials corresponding to a LHCA rule and its complement are either both reducible or irreducible: if $p(x)$ corresponds to the LHCA rule r then $p(x+1)$ corresponds to the rule that is the complement of r.

```
                    LHCA rule                          polynomial
 1:   r=...............................1  c=...............................11  r= [0]
 2:   r=...............................1  c=..............................111  r= [0]
 3:   r=...............................1  c=............................11.1  r= [0]
 4:   r=.............................1.1  c=............................1..11  r= [0, 2]
 5:   r=...............................1  c..........................11.111  r= [0]
 6:   r=...............................1  c=.........................111..11  r= [0]
 7:   r=.............................1..  c=.......................11.....1  r= [2]
 8:   r=............................11.  c=......................1...111.1  r= [1, 2]
 9:   r=...............................1  c.....................11.111..11  r= [0]
10:   r=...........................1111  c=....................1.111111.11  r= [0, 1, 2, 3]
11:   r=...............................1  c=..................11.1...111.1  r= [0]
12:   r=...........................1.11.  c=.................11..1.1.....1  r= [1, 2, 4]
13:   r=...........................1..1  c=................1.11..1.1111.1  r= [0, 3]
14:   r=...............................1  c................111..11......11  r= [0]
15:   r=.............................1..  c..............11.....1...1...1  r= [2]
16:   r=...........................1.1.1  c.............11.1..1111..1.111  r= [0, 2, 4]
17:   r=..............................11  c............1.1111..11......11  r= [0, 1]
18:   r=...........................1.11.  c............11...11.1.1..1..1.1  r= [1, 2, 4]
19:   r=...........................1..  c............11.....111.1....11.1  r= [2]
20:   r=..............................11.  c..........1...111.....1...1...1  r= [1, 2]
21:   r=...........................11..1.  c..........1111.11.1111.1..1.11.1  r= [1, 4, 5]
22:   r=...........................1.11  c.........11.1.111..11.11.11.1.11  r= [0, 1, 3]
23:   r=...............................1  c........11.1...1.......111.1...1  r= [0]
24:   r=.........................1.111.  c.......1.1..1.1....11....1..1..1  r= [1, 2, 3, 5]
25:   r=..............................1.11  c......111....1.1...1.11.1....1.1  r= [0, 1, 3]
26:   r=...............................1  c.....111..11.111..........11.111  r= [0]
27:   r=...........................111..  c....1111..1.1111...1......1.1111  r= [2, 3, 4]
28:   r=...........................1..  c...111111.1.1111........1.1.1111  r= [2]
29:   r=...............................1  c=..11.111.....111.............111  r= [0]
30:   r=...............................1  c=.111..11......11..............11  r= [0]
31:   r=...........................11.1.  c=11111.1.....1.11.......1....1.11  r= [1, 3, 4]
```

Figure 39.7-C: Low-bit LHCA rules and the corresponding binary polynomials.

Figure 39.7-C shows a list of LHCA rules where the highest bit lies in the lowest possible position. The list can be produced with [FXT: gf2n/lowbit-lhca-demo.cc]. For software implementation of LHCA that need more than one machine word low-bit rules are advantageous. A table of low-bit LHCA rules corresponding to primitive polynomials up to $n = 400$ is given in [FXT: data/lowbit-lhca-rules.txt]. The maximal value of a rule that occurs is $r = 1293$ for $n = 380$ so the table can be stored compactly.

Figure 39.7-D shows a list of LHCA rules that have minimal weight and the highest bit in the lowest possible position. The list can be produced with [FXT: gf2n/minweight-lowbit-lhca-demo.cc].

39.7.2 Conversion of irreducible binary polynomials to LHCAs

The LHCA corresponding to an irreducible binary polynomial $p(x)$ proceeds in two steps. Firstly, the quadratic equation (over GF(2^n))

$$Z^2 + (x^2 + x)\, p'(x)\, Z + 1 \quad = \quad 0 \mod p(x) \tag{39.7-2}$$

must be solved for Z. The algorithm is given in section 40.4 on page 861, the implementation is given in [FXT: bpol/bitpolmod-solvequadratic.cc]. The second step is a GCD computation. Set $Z_{n-1} := p$ and

```
                         LHCA rule                            polynomial
   1:    r=..............................1  c=..............................11  r= [0]
   2:    r=..............................1  c=.............................111  r= [0]
   3:    r=..............................1  c=............................11.1  r= [0]
   4:    r=............................1.1  c=...........................1..11  r= [0, 2]
   5:    r=..............................1  c=..........................11.111  r= [0]
   6:    r=..............................1  c=.........................111..11  r= [0]
   7:    r=............................1..  c=........................11.....1  r= [2]
   8:    r=............................11.  c=.......................1...111.1  r= [1, 2]
   9:    r=..............................1  c=......................11.111..11  r= [0]
  10:    r=........................1....1.  c=.....................1...11.1111  r= [1, 6]
  11:    r=..............................1  c=....................11.1...111.1  r= [0]
  12:    r=........................1...1..  c=...................1.....11111.1  r= [2, 6]
  13:    r=..........................1....  c=..................11..11.....111  r= [4]
  14:    r=..............................1  c=.................111..11......11  r= [0]
  15:    r=............................1..  c=................11.....1...1...1  r= [2]
  16:    r=........................1.1....  c=...............1..........111111  r= [4, 6]
  17:    r=..........................1....  c=..............11..11..11.1....11  r= [4]
  18:    r=.........................1...1.  c=.............1...11.11.1111111.1  r= [1, 5]
  19:    r=............................1..  c=............11.....111.1....11.1  r= [2]
  20:    r=............................11.  c=...........1...111.....1...1...1  r= [1, 2]
  21:    r=........................11.....  c=..........1.1...1......1.1.1...1  r= [5, 6]
  22:    r=..........................1....  c=.........1111..11...1.11.1111.11  r= [4]
  23:    r=..............................1  c=........11.1...1.......111.1...1  r= [0]
  24:    r=...................1...1.......  c=.......1.....1.1...1.1..11111..1  r= [7, 11]
  25:    r=......................1........  c=......11..11..11.........111..11  r= [8]
  26:    r=..............................1  c=.....111..11.111..........11.111  r= [0]
  27:    r=........................1..1...  c=....1.1......1..1..11111.....1.1  r= [3, 6]
  28:    r=............................1..  c=...111111.1.1111........1.1.1111  r= [2]
  29:    r=..............................1  c=..11.111.....111.............111  r= [0]
  30:    r=..............................1  c=.111..11......11..............11  r= [0]
  31:    r=....................1..........  c=11.....................1...1...1  r= [10]
```

Figure 39.7-D: Minimum-weight low-bit LHCA rules and the corresponding binary polynomials.

$Z_{n-2} := Z$, successively compute Z_{n-3}, Z_{n-4}, ..., Z_0 so that

$$Z_{n-1} = (x + r_0)\, Z_{n-2} + Z_{n-3} \tag{39.7-3a}$$

$$Z_{n-2} = (x + r_1)\, Z_{n-3} + Z_{n-4} \tag{39.7-3b}$$

$$\vdots$$

$$Z_2 = (x + r_{n-3})\, Z_1 + 1 \tag{39.7-3c}$$

$$Z_1 = (x + r_{n-2})\, Z_0 + 1 \tag{39.7-3d}$$

$$Z_0 = (x + r_{n-1})\, 1 + 0 \tag{39.7-3e}$$

Each step consists of a computation of polynomial quotient and remainder (see section 38.1 on page 793). The vector $[r_{n-1}, r_{n-2}, \ldots, r_0]$ then gives the LHCA rule. The implementation is given [FXT: bpol/bitpol2lhca.cc]:

```
ulong
poly2lhca(ulong p)
// Return LHCA rule corresponding to the binary polynomial P.
// Must have: P irreducible.
{
    ulong dp = bitpol_deriv(p);
    const ulong h = bitpol_h(p);

    ulong b = dp;
    b ^= bitpolmod_times_x(b, p, h); // p' * (x+1)
    b = bitpolmod_times_x(b, p, h);  // p' * (x^2+x)

    ulong r0, r1;  // solutions of  1 + (p'*(x*x+x))*z + z*z == 0 modulo p
    bool q = bitpolmod_solve_quadratic(1, b, 1, r0, r1, p);
    if ( 0==q )  return 0;

    // GCD steps:
    ulong r = 0;  // rule vector
    ulong x = p,  y = r0;  // same result with r1
    while ( y )
    {
        ulong tq, tr;
```

```
        bitpol_divrem(x, y, tq, tr);
        r <<= 1;
        r |= (tq & 1);
        x = y;
        y = tr;
    }
    return r;
}
```

The described algorithm is given in the very readable paper [71] which is recommended for further studies. Note that the paper uses the reversed rule which can be obtained by inserting the line

```
    r = revbin(r, bitpol_deg(p) );
```

just before the final return statement. The program [FXT: gf2n/poly2lhca-demo.cc] converts a given polynomial into the corresponding LHCA rule.

39.8 Additive linear hybrid cellular automata

39.8.1 Conversion into binary polynomials

The algorithm for the conversion of LHCA rules to binary polynomials is a special case of a general method for additive cellular automata. An automaton is called additive if, for all words a and b,

$$N(a) + N(b) \quad = \quad N(a + b) \tag{39.8-1}$$

where $N(x)$ is the next state after the state x and addition is bit-wise (XOR).

For additive automata the action of N on a binary word can be described by a matrix over GF(2): Let e_k be the word with bit number k the only set bit (e_k, $k = 0, \ldots, n-1$ is the canonical basis). The matrix whose k-th row is $N(e_k)$ is the matrix we seek. For example, for the LHCA with cyclic boundary condition (CLHCA) whose action N can be implemented as [FXT: bpol/clhca.h]

```
ulong clhca_next(ulong x, ulong r, ulong n)
{
    r &= x;
    ulong t = x ^ bit_rotate_right(x, 1, n);
    t ^= r;
    return  t;
}
```

we obtain for the rule $r := [r_0, r_1, \ldots, r_{n-1}]$ the matrix

$$M_r \quad := \quad \begin{bmatrix} s_0 & & & & & & & & 1 \\ 1 & s_1 & & & & & & & \\ & 1 & s_2 & & & & & & \\ & & 1 & s_3 & & & & & \\ & & & \ddots & \ddots & & & & \\ & & & & 1 & s_{n-3} & & & \\ & & & & & 1 & s_{n-2} & & \\ & & & & & & 1 & s_{n-1} & \end{bmatrix} \tag{39.8-2}$$

where s_k is the complement of r_k and blank entries denote zero. The binary polynomial corresponding to the automaton is the characteristic polynomial of the matrix M_r. We use the routine `bitmat_charpoly()` given in [FXT: bmat/bitmat-charpoly.cc] (which uses a reduction to the Hessenberg form as given in [83]):

```
inline ulong clhca2poly(ulong r, ulong n)
// Compute the binary polynomial corresponding
//   to the length-n CLHCA with rule r.
{
    ALLOCA(ulong, M, n);
    for (ulong k=0; k<n; ++k)
    {
```

```
        M[k] = clhca_next( 1UL<<k, r, n );
    }
    ulong c = bitmat_charpoly(M, n);
    return c;
}
```

The routine computes the polynomial for any additive automaton if the call to `clhca_next()` is replaced by the update for the automaton. Note that with our particular example the matrix is already in Hessenberg form so we could directly call `bitmat_hessenberg2charpoly()`.

39.8.2 Properties of the CLHCA

```
          r = .111    r = 1.11    r = 11.1    r = 111.

      1:      ...1        ...1        ...1        ...1
      2:      1...        1...        1...        1..1
      3:      11..        .1..        .1..        11.1
      4:      111.        .11.        ..1.        1111
      5:      1111        .111        ..11        111.
      6:      .111        1111        1.11        .111
      7:      1.11        1.11        1111        1.1.
      8:      .1.1        11.1        11.1        .1.1
      9:      1.1.        1.1.        111.        1.11
     10:      11.1        .1.1        .1.1        11..
     11:      .11.        111.        1.1.        .11.
     12:      ..11        ..11        .111        ..11
     13:      1..1        1..1        1..1        1...
     14:      .1..        11..        11..        .1..
     15:      ..1.        ..1.        .11.        ..1.
```

Figure 39.8-A: Rules of identical weight lead to essentially identical CLHCA: all tracks in the successive states of all weight-3 (length-4) automata are cyclic shifts of each other.

```
     k      r =  CLHCA rule            c = binary polynomial
     0:     r = ................       c = 11..............1.
     1:     r = ...............1       c = 1................11
     2:     r = ..............11       c = 1111111111111111.1
     3:     r = .............111       c = 1.1.1.1.1.1.1.1..1  P
     4:     r = ............1111       c = 11..11..11..11...1
     5:     r = ...........11111       c = 1...1...1...1....1  P
     6:     r = ..........111111       c = 1111....1111.....1  P
     7:     r = .........1111111       c = 1.1.....1.1......1
     8:     r = ........11111111       c = 11......11.......1
     9:     r = ........11111111       c = 1.......1........1
    10:     r = .......111111111       c = 11111111.........1
    11:     r = ......1111111111       c = 1.1.1.1..........1  P
    12:     r = .....11111111111       c = 11..11...........1  P
    13:     r = ....111111111111       c = 1...1............1
    14:     r = ...1111111111111       c = 1111.............1  P
    15:     r = ..11111111111111       c = 1.1..............1
    16:     r = .111111111111111       c = 11...............1
    17:     r = 1111111111111111       c = 1................1
```

Figure 39.8-B: Binary polynomials (right) corresponding to rule vectors for the length-17 cyclic linear hybrid cellular automata (left) with k bits set. Automata with maximal period are marked with '`P`'.

The polynomials for the given automata depend only on the number of bits set in the rule r. Figure 39.8-A shows the successive states for all length-4 CLHCA with rules of weight 3. As there are essentially only $n+1$ different automata of length n we only need to investigate the rules with the lowest k bits set for $k = 0, 1, \ldots, n$. The polynomials for the length-17 automata are shown in figure 39.8-B. The polynomials can be computed with the simple routine

```
inline ulong clhca2poly(ulong r, ulong n)
{
    ulong c = 1UL << n;
    for (ulong k=0; k<n; ++k)  if ( 0==(r & (1UL<<k)) )  c ^= (c>>1);
    c ^= 1;
    return c;
}
```

```
w=2:  r=...11          w=3:  r=..111          w=1:  r=....1

 1:      ....1          1:      ....1          1:      ....1
 2:      1....          2:      1....          2:      1....
 3:      11...          3:      11...          3:      11...
 4:      1.1..          4:      1.1..          4:      1.1..
 5:      1111.          5:      11.1.          5:      1111.
 6:      1..11          6:      1.1.1          6:      1...1
 7:      .1..1          7:      .1.1.          7:      .1...
 8:      111..          8:      .11.1          8:      .11..
 9:      1..1.          9:      1111.          9:      .1.1.
10:      11..1         10:      1.111         10:      .1111
11:      ..1..         11:      .1.11         11:      11..1
12:      ..11.         12:      111.1         12:      ..1..
13:      ..111         13:      ..11.         13:      ..11.
14:      1.111         14:      ...11         14:      ..1.1
15:      .1111         15:      1...1         15:      1.11.
16:      11.11         16:      .1...         16:      111.1
17:      ..1.1         17:      .11..         17:      ...1.
18:      1.11.         18:      .111.         18:      ...11
19:      11111         19:      .1111         19:      1..11
20:      ...11         20:      11111         20:      .1.11
21:      1...1         21:      ..111         21:      11111
22:      .1...         22:      1..11
23:      .11..         23:      .1..1
24:      .1.1.         24:      111..
25:      .11.1         25:      1.11.
26:      11.1.         26:      11.11
27:      1.1.1         27:      ..1.1
28:      .111.         28:      1..1.
29:      .1.11         29:      11..1
30:      111.1         30:      ..1..
31:      ...1.         31:      ...1.
```

Figure 39.8-C: The length-5, weight-w CLHCA has maximal period for $w = 2$ and $w = 3$. The successive states are shown for both automata (left and middle). The weight-1 rule leads to a period of 21 (right).

```
  n  w:      c = polynomial

  2  1:      c = 111
  3  1:      c = 1.11
  4  3:      c = 11..1
  5  2:      c = 1111.1
  6  1:      c = 11..111
  7  1:      c = 1.1.1.11
 [8]
  9  4:      c = 11..11...1
 10  3:      c = 11111111..1
 11  2:      c = 11......11.1
[12,13,14]
 15  4:      c = 1111....1111...1
[16]
 17  3:      c = 1.1.1.1.1.1.1.1..1
 18  7:      c = 1111....1111......1
[19]
 20  3:      c = 11............11..1
 21  2:      c = 1111..........1111.1
 22 21:      c = 11..................1
 23  5:      c = 1.1............1.1....1
[24]
 25  3:      c = 1.1.1.1.........1.1.1.1..1
[26,27]
 28  9:      c = 1111...........1111.........1
 29  2:      c = 1111....1111....1111....1111.1
[30]
```

Figure 39.8-D: The polynomials (right) corresponding rules of lowest weight (w) so that the length-n ($n \leq 30$) CLHCA has maximal period.

If we write $C_{n,k}(x)$ for the polynomial for the length-n, k-bit rule then $C_{n,n-k}(x) = C_{n,k}(x+1)$. Indeed, they can be given in the closed form $C_{n,k}(x) = 1 + x^k\,(1+x)^{n-k}$.

With $n = 5$ there are just two rules that lead to maximal periods, $r = [0, 0, 0, 1, 1]$ (weight 2), and $r = [0, 0, 1, 1, 1]$ (weight 3). The successive states for both rules are shown in figure 39.8-C. The polynomials corresponding to the rules of minimal weight for all length-n automata where $n \leq 30$ are given in figure 39.8-D. The sequence of values n so that a primitive length-n CLHCA exists starts:

```
2, 3, 4, 5, 6, 7, 9, 10, 11, 15, 17, 18, 20, 21, 22, 23, 25, 28, 29, 31, 33,
35, 36, 39, 41, 47, 49, 52, 55, 57, 58, 60, 63, ...
```

It coincides with entry A073726 of [214], values n such that there is a primitive trinomial of degree n. The sequence was computed with the program [FXT: gf2n/clhca-demo.cc]. A list of CLHCA rules with maximal period is given in [FXT: data/clhca-rules.txt].

Chapter 40

Binary finite fields: $\mathrm{GF}(2^n)$

This chapter introduces the binary finite fields $\mathrm{GF}(2^n)$. The polynomial representation is used for stating some basic properties. An introduction of the representation by normal bases follows. Certain normal bases are advantageous for hardware implementations of the arithmetic algorithms. Finally, several ways of computing the number of irreducible binary normal polynomials are given.

Binary finite fields are important for applications like error correcting codes and cryptography.

The underlying arithmetical algorithms are given in chapter 38.

40.1 Arithmetic and basic properties

In chapter 25 we have met the finite fields $\mathbb{Z}/p\mathbb{Z} = \mathrm{GF}(p)$ for p a prime. The 'GF' stands for *Galois Field*, another symbol often used is $\mathbb{F}_p$. The arithmetic in $\mathrm{GF}(p)$ is the simply the arithmetic modulo p.

There are more finite fields: for every prime p there are fields with $Q = p^n$ elements for all $n \geq 1$. All elements in a finite field $\mathrm{GF}(p^n)$ can be represented as polynomials modulo an degree-n irreducible polynomial C with coefficients over the field $\mathrm{GF}(p)$. The arithmetic to be used is polynomial arithmetic modulo C. As in general there is more than one irreducible polynomial of degree n it might seem that there is more than one field $\mathrm{GF}(Q)$ for given $Q = p^n$. There isn't. Using different polynomials as modulus leads to isomorphic representations of the same field. The field $\mathrm{GF}(p^n)$ is called an *extension field* of $\mathrm{GF}(p)$. The field $\mathrm{GF}(p)$ is called the *ground field* of $\mathrm{GF}(p^n)$.

When speaking about an element of $\mathrm{GF}(Q)$ one can think of a polynomial modulo some fixed irreducible polynomial C (modulus). For example, the product of two elements can be computed as the polynomial product modulo C. For the equivalent construction using the polynomial $x^2 + 1$ with real coefficients that leads to the complex numbers, see section 37.12 on page 770.

The elements zero (the neutral element of addition) and one (neutral element of multiplication) are the constant polynomials with constant zero and one, respectively. This does not depend on the choice of the modulus.

Multiplication with an element of the ground field is called *scalar multiplication*. In this section an element of the ground field is denoted by u. Scalar multiplication corresponds to the multiplication of every coefficient of (the polynomial representing) the field element a by u.

We restrict our attention mostly to $Q = 2^n$, that is, the *binary finite fields* $\mathrm{GF}(2^n)$ in what follows as we have seen the algorithms for the underlying arithmetic.

40.1.1 Characteristic and linear functions

If we add any element of the field GF(p^n) p times to zero the result will be zero. One calls p the *characteristic* of the field. For infinite fields such as $\mathbb{C}$ the characteristic is said to be zero (while it should really be ∞). Note that the notion "multiplication is repeated addition" is meaningless in extension fields.

For GF(p^n) one has

$$(u+v)^p \quad = \quad u^p + v^p \tag{40.1-1}$$

because the binomial coefficients $\binom{p}{k}$ are divisible by p for $k = 1, 2, \ldots, p-1$. For GF(2^n):

$$(u+v)^2 \quad = \quad u^2 + v^2 \tag{40.1-2}$$

We call a function f *linear* if the relation

$$f(u_1 \cdot a + u_2 \cdot b) \quad = \quad u_1 \cdot f(a) + u_2 \cdot f(b) \tag{40.1-3}$$

holds for u_1 and u_2 from the ground field. The linear functions in $GF(p^n)$ are of the form

$$f(x) \quad = \quad \sum_{k=0}^{n-1} u_k \cdot x^{p^k} \tag{40.1-4}$$

where the u_k are again in the ground field. Linear functions can be computed using lookup tables. In $GF(2^n)$ these are all functions of the form

$$f(x) \quad = \quad \sum_{k=0}^{n-1} u_k \cdot x^{2^k} \tag{40.1-5}$$

40.1.2 Squaring

Squaring (and raising to any power 2^k) is a linear operation in GF(2^n). The linearity can be used to accelerate the computation of squares. Write

$$\begin{aligned} \left(u_0 + u_1\, x + u_2\, x^2 + \ldots + u_{n-1}\, x^{n-1}\right)^2 \quad &= \quad u_0 + u_1\, x^2 + u_2\, x^4 + \ldots + u_{n-1}\, x^{2\,(n-1)} \\ &=: \quad u_0\, s_0 + u_1\, s_1 + u_2\, s_2 + \ldots u_{n-1}\, s_{n-1} \end{aligned} \tag{40.1-6}$$

One has to precompute and store the values $s_i = x^{2\,i} \bmod C$ for $i = 0, 1, 2, \ldots, n-1$. For successive square computations it is only necessary to add (that is, XOR) those s_i corresponding to nonzero u_i. For example, with $n = 13$ and the polynomial modulus $C = x^{13} + x^4 + x^3 + x^1 + 1$ one obtains the table

```
 S0 =  ............1
 S1 =  ...........1..
 S2 =  ..........1...
 S3 =  .........1....
 S4 =  ........1.....
 S5 =  .......1......
 S6 =  ......1.......
 S7 =  .....11.11....
 S8 =  ....11.11.....
 S9 =  ...11.11......
S10 =  ..11.11.......
S11 =  .1.11....11.11
S12 =  .11....1.11.1.
```

The squares $s_0, s_1 \ldots s_{\lfloor (n-1)/2 \rfloor}$ have the trivial form $s_i = x^{2\,i}$ which can be used to further accelerate the computation.

40.1.3 Computation of the trace

The *trace* of an element u in GF(2^n) is defined as

$$\mathrm{Tr}(a) \quad := \quad a + a^2 + a^4 + a^8 + \ldots + a^{2^{n-1}} \quad = \sum_{j=0}^{n-1} a^{2^j} \tag{40.1-7}$$

The trace of any element is either zero or one. The trace of zero is always zero. The trace of one in GF(2^n) equals one for odd n, else zero, that is, $\mathrm{Tr}(1) = n \bmod 2$. Exactly half of the elements have trace one. The trace of the sum of two elements is the sum of the traces of the elements:

$$\mathrm{Tr}(a+b) \quad = \quad \mathrm{Tr}(a) + \mathrm{Tr}(b) \tag{40.1-8}$$

The trace of the square of an element is the trace of that element:

$$\mathrm{Tr}(a^2) \quad = \quad \mathrm{Tr}(a) \tag{40.1-9}$$

With u zero or one (an element of the ground field GF(2)) one has

$$\mathrm{Tr}(u \cdot a) \quad = \quad u \cdot \mathrm{Tr}(a) \tag{40.1-10}$$

Thereby, for u_1 and u_1 from the ground field,

$$\mathrm{Tr}(u_1 \cdot a + u_2 \cdot b) \quad = \quad u_1 \cdot a + u_2 \cdot b \tag{40.1-11}$$

That is, the Tr function is linear.

A fast algorithm to compute the trace uses the *trace vector*, a precomputed table $t_i = \mathrm{Tr}(x^i)$ for $i = 0, 1, 2, \ldots, n-1$, and the linearity of the trace

$$\mathrm{Tr}(u) \quad = \quad \sum_{i=0}^{n-1} u_i \, t_i \tag{40.1-12}$$

where the u_i are zero or one. Thereby, when the polynomial fits into a single binary word, precompute the trace vector `tv` whose bits are the traces of the powers of x and later compute the trace of an element via [FXT: `gf2n_fast_trace()` in bpol/gf2n-trace.h]:

```
inline ulong gf2n_fast_trace(ulong a, ulong tv)
// Fast trace computation of the trace of a
//  using the precalculated table tv.
{ return  parity( a & tv ); }  // scalar product over GF(2)
```

Given the trace vector it is also easy to find elements of trace zero or one by simply taking the lowest unset or set bit of the vector, respectively. There are polynomials such that the trace vector contains just one nonzero bit, see section 859.

40.1.4 Inverse and square root

The number of elements in GF(Q) equals Q. For any element $a \in \mathrm{GF}(Q)$ one has

$$a^Q \quad = \quad a \tag{40.1-13}$$

and thereby $a^{Q-1} = 1$. So we can compute the *inverse* a^{-1} of a nonzero element a as

$$a^{-1} \quad = \quad a^{Q-2} \tag{40.1-14}$$

We have see this technique of inversion by exponentiation is section 37.6.4 on page 746 which is the special case $GF(p)$.

All elements except zero are invertible in a field. That is, the number of invertible elements (units) in GF(Q) equals $|\mathrm{GF}(Q)^*| = Q - 1 = p^n - 1$

Every element a of GF(2^n) has a unique *square root* s that can easily be computed as

$$s \;=\; a^{Q/2} \;=\; a^{2^{n-1}} \tag{40.1-15}$$

It can be computed by squaring the element $n-1$ times. But the square root is a linear function, so we can again apply table lookup methods.

A method that uses the precomputed value $\sqrt{x}$ is described in [111]: for an element $a = \sum_k a_k\, x^k$ we have

$$\sqrt{a} \;=\; \sum_{k\ \mathrm{even}} a_k\, x^{k/2} + \sqrt{x} \sum_{k\ \mathrm{odd}} a_k\, x^{(k-1)/2} \tag{40.1-16}$$

The only nontrivial operation is the multiplication with $\sqrt{x}$.

40.1.5 Order and primitive roots

The *order* of an element a is the least positive exponent r so that $a^r = 1$. The maximal order of an element in GF(2^n) equals $2^n - 1 = Q - 1$. An element of maximal order is called a *generator* (or *primitive root*) as its powers 'generate' all nonzero elements in the field. The order of an given element a in GF(2^n) can be computed like

```
function order(a, n)
{
    if a==0 then  return 0  // a not a unit

    h := 2**n - 1  // number of units
    e := h
    {np, p[], k[]} := factorization(h)  // h==product(i=0..np-1, p[i]**k[i])
    for i:=0 to np-1
    {
        f := p[i]**k[i]
        e := e / f
        g1 := a**e  // modulo polynomial
        while g1!=1
        {
            g1 := g1**p[i]  // modulo polynomial
            e := e * p[i]
            p[i] := p[i] - 1
        }
    }
    return e
}
```

The C++ implementation is [FXT: `gf2n_order()` in bpol/gf2n-order.cc]. We have seen a very similar algorithm in section 38.4.2 on page 815. Indeed, with just minimal changes we obtain a pari/gp implementation to compute the order of an element g in GF(2^n):

```
polorder(p, g='x) =
/* Order of g modulo p (p irreducible over GF(2)) */
{
  local(g1, te, tp, tf, tx);
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
    );
  );
  return( te );
}
```

The only difference to the original algorithm is that g can have other values than (the polynomial) x. The function uses the following global variables that must be set up before call:

```
nn_ = 0;  /* max order  =  2^n-1 */
np_ = 0;  /* number of primes in factorization */
vp_ = []; /* vector of primes */
vf_ = []; /* vector of factors (prime powers) */
vx_ = []; /* vector of exponents */
```

To check whether g is a primitive root modulo p use the function

```
polmaxorder_q(p, g='x) =
/* Whether order of g modulo p is maximal  (p irreducible over GF(2)) */
/* Early-out variant */
{
  local(g1, te, tp, tf, tx, ct);
  p *= Mod(1,2);
  te = nn_;
  for(i=1, np_,
    tf = vf_[i];  tp = vp_[i];  tx = vx_[i];
    te = te / tf;
    g1 = Mod(g, p)^te;
    ct = 0;
    while ( 1!=g1,
      g1 = g1^tp;
      te = te * tp;
      ct = ct + 1;
    );
    if ( ct<tx,  return(0) );
  );
  return(1);
}
```

Let a be an invertible element (that is, $a \neq 0$), then a can be written as a power of a generator: $a = g^k$. Then for the order r of a we have

$$r(g^k) \quad = \quad \frac{N}{\gcd(k, N)} \tag{40.1-17}$$

where $N = 2^n - 1$. For N a (Mersenne-) prime the order of all invertible elements except 1 is N.

40.1.6 Implementation

A C++ class for computations in the fields GF(2^n) with n not greater than `BITS_PER_LONG` is [FXT: `class GF2n` in bpol/gf2n.h]:

```
class GF2n
// Implementation of binary finite fields GF(2**n)
//   with arithmetic operations on it.
{
public:
    ulong v_;
```

The static (that is, class global) elements support the computations:

```
public:
    static ulong n_;  // the 'n' in GF(2**n)
    static ulong c_;  // polynomial modulus
    static ulong h_;  // auxiliary bitmask for computations
    static ulong mm_; // 2**n - 1 == max order (a Mersenne number)
    static ulong g_;  // a generator (element of maximal order)
    static ulong tv_; // trace vector
    static ulong sqr_tab[BITS_PER_LONG]; // table for fast squaring
    static factorization mfact_; // factorization of max order
    static char* pc_; // chars to print zero and one: e.g.  "01" or ".1"
    [--snip--]
    static GF2n zero; // zero (neutral element wrt. addition) in GF(2**n)
    static GF2n one;  // one (neutral element wrt. multiplication) in GF(2**n)
    static GF2n tr1e; // an element with trace == 1
```

Note all data is public, making many methods '`get_something()`' unnecessary. You can also modify to the data. Expect funny results if you do. The constructors from other types are '`explicit`' to avoid surprises:

```
public:
    explicit GF2n()  { ; }
    explicit GF2n(const ulong i) : v_(i & mm_)  { ; }
    GF2n(const GF2n &g) : v_(g.v_)  { ; }
    ~GF2n()  { ; }
```

Before doing anything one has to call the initializing function [FXT: `GF2n::init()` in bpol/gf2n.cc]:

```
// if INIT_ASSERT is defined, asserts are C asserts,
// else init() returns false if one of the tests fail:
#define INIT_ASSERT

bool // static
GF2n::init(ulong n, ulong c/*=0*/, bool normalq/*=0*/, bool trustme/*=0*/)
// Initialize class GF(2**n) for 0<n<=BITS_PER_LONG.
// If an irreducible polynomial c is supplied it is used as modulus,
// else a primitive polynomial of degree n is used.
// Irreducibility of c is asserted for deg(c)<BITS_PER_LONG.
// When normalq is set a primitive normal polynomial is used,
// if in addition c is supplied, normality of c is asserted.
// When trustme is set the asserts are omitted.
{
  [--snip--]
        if ( n_ < BITS_PER_LONG )  // test only works for polynomials that fit into words
        {
            if ( ! trustme )
            {
#ifdef INIT_ASSERT
                jjassert( bitpol_irreducible_q(c_, h_) );
#else
                if ( ! bitpol_irreducible_q(c_, h_) )  return false;
#endif
            }
        }
  [--snip--]
}
```

```
      n = 4  GF(2^n)
      c = 1..11 == x^4 + x + 1  (polynomial modulus)
      mm= .1111   == 15  =  3 * 5  (maximal order)
      h = .1... (aux. bitmask)
      g = ...1. (element of maximal order)
      tv= .1... (traces of x^i)
    tr1e= .1... (element with trace=1)

          k  :    f:=g**k  Tr(f)    ord(f)     f*f    sqrt(f)
        ...1 :      ...1     0         1      ...1     ...1
        ..1. :      ..1.     0        15      .1..     .1.1
        .1.. :      .1..     0        15      ..11     ..1.
        .1.1 :      1...     1         5      11..     1.1.
        .11. :      ..11     0        15      .1.1     .1..
        .111 :      .11.     0         3      .111     .111
        1... :      11..     1         5      1111     1...
        1..1 :      1.11     1        15      1..1     111.
        1.1. :      .1.1     0        15      ..1.     ..11
        1.11 :      1.1.     1         5      1...     1111
        11.. :       .111    0         3      .11.     .11.
        11.1 :      111.     1        15      1.11     11.1
        111. :      1111     1         5      1.1.     11..
                    11.1     1        15      111.     1..1
                    1..1     1        15      11.1     1.11
```

Figure 40.1-A: Powers of the generator $g = x$ in GF(2^4) when a primitive polynomial is used as a modulus.

The class defines all the standard operators like the binary operators '`+`' and '`-`' (which are the same operation in GF(2^n)), '`*`' and '`/`', the comparison operators '`==`' and '`!=`', also the computation of inverse, powering, order, and trace. The algorithms used for the arithmetic operations are described in section 38.3 on page 805. We give the method for the inverse and the arithmetic shortcut operators as examples:

```
    GF2n inv()  const
```

```
    n = 4  GF(2^n)
    c = 11111 [normal] [NON-primitive]
      == x^4 + x^3 + x^2 + x + 1  (polynomial modulus)
    mm= .1111   == 15  =  3 * 5  (maximal order)
    h = .1... (aux. bitmask)
    g = ...11 (element of maximal order)
    tv= .111. (traces of x^i)
 tr1e= ...1. (element with trace=1)

       k  :    f:=g**k  Tr(f)    ord(f)     f*f    sqrt(f)
     ...1 :      ...1     0         1       ...1    ...1
     ..1. :      ..11     1        15       .1.1    1..1
     ..1. :      .1.1     1        15       111.    ..11
     ..11 :      1111     1         5       1...    .1..
     .1.. :      111.     1        15       1..1    .1.1
     .1.1 :      11.1     0         3       11..    11..
     .11. :      1...     1         5       ..1.    1111
     .111 :      .111     0        15       1.1.    1.11
     1... :      1..1     1        15       ..11    111.
     1..1 :      .1..     1         5       1111    ..1.
     1.1. :      11..     0         3       11.1    11.1
     1.11 :      1.11     0        15       .111    .11.
     11.. :      ..1.     1         5       .1..    1...
     11.1 :      .11.     0        15       1.11    1.1.
     111. :      1.1.     0        15       .11.    .111
```

Figure 40.1-B: Powers of a generator in GF(2^4) when a non-primitive polynomial is used as a modulus.

```
{
    GF2n z;
    z.v_= bitpolmod_inverse(v_, GF2n::c_);
    return z;
}
[--snip--]

friend inline GF2n & operator += (GF2n &z, const GF2n &f)
{ z.v_ ^= f.v_;  return z; }

friend inline GF2n & operator -= (GF2n &z, const GF2n &f)
{ z.v_ ^= f.v_;  return z; }

friend inline GF2n & operator *= (GF2n &z, const GF2n &f)
{ z.v_ = bitpolmod_mult(z.v_, f.v_, GF2n::c_, GF2n::h_);  return z; }

friend inline GF2n & operator /= (GF2n &z, const GF2n &f)
{ z *= f.inv();  return z; }
```

As a simple demonstration of the usage, the program [FXT: gf2n/gf2n-demo.cc] prints the successive powers of a primitive root, and their squares and square roots. By default computations in GF(2^4) are shown, both for a primitive polynomial modulus (figure 40.1-A), and for a non-primitive polynomial modulus (figure 40.1-B).

40.2 Minimal polynomials

The *minimal polynomial* of an element a in GF(2^n) is defined as the polynomial of least degree which has a as a root. The minimal polynomial can be computed as the product

$$p_a(x) \;:=\; \prod_{k=0}^{r-1} \left(x - a^{2^k}\right) \tag{40.2-1}$$

where r is the smallest positive integer so that $a^{2^r} = a$. The minimal polynomial of any element is irreducible and its degree is a divisor of n.

By construction the zeros of the polynomial are $a, a^2, a^4, a^8, \ldots, a^{2^{r-1}}$. Note that $\left(a^{2^{r-1}}\right)^2 = a$, that is, $p_a = p_{a^2} = p_{a^4} = \ldots = p_{a^{2^{r-1}}}$. The elements $a^2, a^4, \ldots$ are called the *conjugates* of a. For the field $GF(p^n)$ the minimal polynomial has the form $\prod_{k=0}^{r-1} \left(x - a^{p^k}\right)$.

```
n = 6  GF(2^n)
c = 1....11 == x^6 + x + 1  (polynomial modulus)
mm= .111111   == 63  =  3^2 * 7  (maximal order)

 k          :    f:=g**k  ord(f) Tr(f)  p:=minpoly(f)  deg(p)
 0 = ......  :     .....1     1    0       ......11       1
 1 = .....1  :     ....1.    63    0       .1....11       6
 2 = ....1.  :     ...1..    63    0       .1....11       6
 3 = ....11  :     ..1...    21    0       .1.1.111       6 N
 4 = ...1..  :     .1....    63    0       .1....11       6
 5 = ...1.1  :     1.....    63    1       .11..111       6
 6 = ...11.  :     ....11    21    0       .1.1.111       6 N
 7 = ...111  :     ...11.     9    0       .1..1..1       6 N
 8 = ..1...  :     ..11..    63    0       .1....11       6
 9 = ..1..1  :     .11...     7    0       ....11.1       3
10 = ..1.1.  :     11....    63    1       .11..111       6
11 = ..1.11  :     1...11    63    1       .11.11.1       6
12 = ..11..  :     ...1.1    21    0       .1.1.111       6 N
13 = ..11.1  :     ..1.1.    63    0       .1.11.11       6
14 = ..111.  :     .1.1..     9    0       .1..1..1       6 N
15 = ..1111  :     1.1...    21    1       .111.1.1       6 N
16 = .1....  :     .1..11    63    0       .1....11       6
17 = .1...1  :     1..11.    63    1       .11..111       6
18 = .1..1.  :     ..1111     7    0       ....11.1       3
19 = .1..11  :     .1111.    63    0       .1.11.11       6
20 = .1.1..  :     1111..    63    1       .11..111       6
21 = .1.1.1  :     111.11     3    1       .....111       2
22 = .1.11.  :     11.1.1    63    1       .11.11.1       6
23 = .1.111  :     1.1..1    63    1       .111..11       6
24 = .11...  :     .1...1    21    0       .1.1.111       6 N
25 = .11..1  :     1...1.    63    1       .11.11.1       6
26 = .11.1.  :     ...111    63    0       .1.11.11       6
27 = .11.11  :     ..111.     7    0       ....1.11       3
28 = .111..  :     .111..     9    0       .1..1..1       6 N
29 = .111.1  :     111...    63    1       .111..11       6
30 = .1111.  :     11..11    21    1       .111.1.1       6 N
31 = .11111  :     1..1.1    63    1       .11....1       6
32 = 1.....  :     ..1..1    63    0       .1....11       6
33 = 1....1  :     .1..1.    21    0       .1.1.111       6 N
34 = 1...1.  :     1..1..    63    1       .11..111       6
35 = 1...11  :     ..1.11     9    0       .1..1..1       6 N
36 = 1..1..  :     .1.11.     7    0       ....11.1       3
37 = 1..1.1  :     1.11..    63    1       .11.11.1       6
38 = 1..11.  :     .11.11    63    0       .1.11.11       6
39 = 1..111  :     11.11.    21    1       .111.1.1       6 N
40 = 1.1...  :     1.1111    63    1       .11..111       6
41 = 1.1..1  :     .111.1    63    0       .1.11.11       6
42 = 1.1.1.  :     111.1.     3    1       .....111       2
43 = 1.1.11  :     11.111    63    1       .111..11       6
44 = 1.11..  :     1.11.1    63    1       .11.11.1       6
45 = 1.11.1  :     .11..1     7    0       ....1.11       3
46 = 1.111.  :     11..1.    63    1       .111..11       6
47 = 1.1111  :     1..111    63    1       .11....1       6
48 = 11....  :     ..11.1    21    0       .1.1.111       6 N
49 = 11...1  :     .11.1.     9    0       .1..1..1       6 N
50 = 11..1.  :     11.1..    63    1       .11.11.1       6
51 = 11..11  :     1.1.11    21    1       .111.1.1       6 N
52 = 11.1..  :     .1.1.1    63    0       .1.11.11       6
53 = 11.1.1  :     1.1.1.    63    1       .111..11       6
54 = 11.11.  :     .1.111     7    0       ....1.11       3
55 = 11.111  :     1.111.    63    1       .11....1       6
56 = 111...  :     .11111     9    0       .1..1..1       6 N
57 = 111..1  :     11111.    21    1       .111.1.1       6 N
58 = 111.1.  :     111111    63    1       .111..11       6
59 = 111.11  :     1111.1    63    1       .11....1       6
60 = 1111..  :     111..1    21    1       .111.1.1       6 N
61 = 1111.1  :     11...1    63    1       .11....1       6
62 = 11111.  :     1....1    63    1       .11....1       6
```

Figure 40.2-A: Minimal polynomials of the powers of a generator in GF(2^6). Polynomials of degree $n = 6$ that are non-primitive are marked with an 'N'. The trace of an element equals the coefficient of x^{n-1} of its minimal polynomial.

From the definition it can be seen that the coefficients of the minimal polynomial lie in GF(2^n), however, all of them are either zero or one. Thereby the computation has to be carried out using the arithmetic in GF(2^n) but the final result is a binary polynomial [FXT: bpol/gf2n-minpoly.cc]:

```
ulong
gf2n_minpoly(GF2n a, ulong &bp)
// Compute the minimal polynomial p(x) of a \in GF(2**n).
// Return the degree of p().
// The polynomial p() is written to bp
{
    GF2n p[BITS_PER_LONG+1];
    ulong n = GF2n::n_;
    for (ulong k=0; k<=n; ++k)  p[k] = 0;
    p[0] = 1;
    ulong d;
    GF2n s = a;
    for (d=1; d<=n; ++d)
    {
        for (ulong k=d; 0!=k; --k)  p[k] = p[k-1];
        p[0] = 0;
        for (ulong k=0; k<d; ++k)  p[k] += p[k+1] * s;
        s = s.sqr();
        if ( s == a )  break;
    }

    // Here all coefficients are either zero or one,
    //  so we can fill them into a binary word:
    ulong p2 = 0;
    for (ulong j=0; j<=d; ++j)  p2 |= (p[j].v_ << j);
    bp = p2;

    return  d;
}
```

A version of the routine that does not depend on the class `GF2n` is given in [FXT: bpol/bitpolmod-minpoly.h]. The program [FXT: gf2n/gf2n-minpoly-demo.cc] prints the minimal polynomials for the powers of a primitive element g, see figure 40.2-A. The polynomials (of maximal degree) that are non-primitive are marked by an '`N`'. The minimal polynomials for g^k are non-primitive or of degree smaller than n whenever $\gcd(k, 2^n - 1) \neq 1$.

Let C be a irreducible polynomial of degree n and a an element so that $r = \mathrm{ord}_C(a)$ is the order of a modulo C. Then the order of x modulo the minimal polynomial of a is also r. Thereby a primitive polynomial can be determined from an irreducible polynomial C and a generator g modulo C by computing the minimal polynomial of g.

With a primitive polynomial (and the generator $g =$'x') the minimal polynomial of an element $f = g^k$ is primitive if k is a Lyndon word and $\gcd(k, n) = 1$. With a fast algorithm for the generation of Lyndon words one can therefore generate all primitive polynomials as shown in section 38.6 on page 824.

40.3 Computation of the trace vector via Newton's formula

Let $C(x)$ be a polynomial with n roots a_0, a_1, ..., a_{n-1}

$$C(x) \;=\; \prod_{k=0}^{n-1} (x - a_k) \;=\; \sum_{k=0}^{n} c_k\, x^k \tag{40.3-1}$$

We define (following [231, sec.32])

$$s_k \;:=\; a_0^k + a_1^k + \ldots + a_{n-1}^k \tag{40.3-2}$$

Then, for $m = 1, \ldots, n$, we have *Newton's formula*:

$$m\, c_{n-m} \;=\; -\sum_{j=0}^{m-1} s_{m-j}\, c_{n-j} \tag{40.3-3}$$

Now let $C = c_0 + c_1\,x + c_2\,x^2 + \ldots + c_n\,x^n$ be an irreducible polynomial with coefficients in GF(p). Its roots are a, (and the conjugates) a^p, a^{p^2}, a^{p^3}, ..., $a^{p^{n-1}}$. Let $t_0 = n$ and $t_i = \mathrm{Tr}(a^i)$ (computationally x is a root of C, so $t_i = \mathrm{Tr}(x^i)$). Note that $t_0, \ldots, t_{n-1}$ are the elements of the trace vector, see relation 40.1-12 on page 853. Using $c_n = 1$ (monic polynomial C) and $t_j = s_j$ we rewrite Newton's formula as

$$
\begin{aligned}
t_1 &= -1\,c_{n-1} && \text{(40.3-4a)}\\
t_2 &= -c_{n-1}\,t_1 - 2\,c_{n-2} && \text{(40.3-4b)}\\
t_3 &= -c_{n-1}\,t_2 - c_{n-2}\,t_1 - 3\,c_{n-3} && \text{(40.3-4c)}\\
t_4 &= -c_{n-1}\,t_3 - c_{n-2}\,t_2 - c_{n-3}\,t_1 - 4\,c_{n-4} && \text{(40.3-4d)}\\
t_5 &= -c_{n-1}\,t_4 - c_{n-2}\,t_3 - c_{n-3}\,t_2 - c_{n-4}\,t_1 - 5\,c_{n-5} && \text{(40.3-4e)}\\
&\vdots\\
t_k &= -c_{n-1}\,t_{k-1} - c_{n-2}\,t_{k-2} - \ldots - c_{n-k-1}\,t_1 - k\,c_{n-k} && \text{(40.3-4f)}
\end{aligned}
$$

To compute the trace vector for the field GF(p^n), make the assignments in the given order, and finally compute $t_0 = n \bmod p$. The computation does not involve any polynomial modular reduction so the method can be worthwhile even for the determination of the trace of just one element.

With binary finite fields, the components with even subscripts can be computed as $t_{2k} = t_k$. During the computation we set $t_0 = 0$ and correct the value at the end of the routine. An implementation of the implied algorithm is [FXT: bpol/gf2n-trace.cc]:

```
ulong
gf2n_trace_vector_x(ulong c, ulong n)
// Return vector of traces of powers of x, where
//  x is a root of the irreducible polynomial C.
// Must have:  n == degree(C)
{
    c &= ~( 2UL<<(n-1) );  // remove coefficient c[n]

    ulong t = 1;  // set t[0]=1, will be corrected at the end
    for (ulong k=1; k<n; ++k)
    {
        if ( k & 1 )  // k odd: use recursion
        {
            ulong cv = c >> (n-k);  // polynomials coefficients [n-1]..[n-k]
            cv &= t;                // vector (j=1, k,  c[n-j]*t[k-j])
            cv = parity(cv);        //    sum (j=1, k,  c[n-j]*t[k-j])
            t |=  (cv<<k);
        }
        else  // k even:  copy t[k/2] to t[k]
        {
            t |= ( (t>>(k/2)) & 1 ) << k;
        }
    }

    // correct t[0]:
    t ^= ((n+1)&1);  // change low bit if n is even

    return t;
}
```

The routine involves n computations of the parity. The complexity of the equivalent routine for large n has complexity $O(n^2)$ (n computations of sums with $\sim n$ summands).

For a binary polynomial C of odd degree n and all nonzero coefficients c_i at odd indices i we obtain $t_0 = 1$ and $t_i = 0$ for all $i \neq 0$, thereby the trace of any element is just the value of its lowest bit. In [45] it is shown that for $n \equiv \pm 3 \bmod 8$ the first nonzero coefficient c_k (with $k < n$) must appear at a position $k \geq n/3$.

With even degree and all nonzero odd coefficients c_i, c_j, c_j, ... at positions $i, j, k, \cdots < n/2$ the only nonzero components of the trace vector are t_{n-i}, t_{n-j}, t_{n-k}, ... Thereby polynomials of even degree with just one nonzero coefficient c_k where $k < n/2$ lead to only t_{n-k} being nonzero. A special case are trinomials $C = x^n + x^k + 1$ of even degree n and $k < n/2$ (k must be odd else C is reducible). The trace vector for all-ones polynomials ($C = \sum_{k=0}^{n} x^k$, see section 38.4.3.9 on page 821) the only zero component

of the trace vector is t_0 (the degree must be even, else C is reducible). A detailed discussion of the properties of the trace vector is given in [6].

The following variant of the algorithm, suggested by Richard Brent [priv.comm.], shows that the computation is equivalent to a division of power series. Let R be the reciprocal polynomial of C, then (see [61, p.135])

$$\log\left(R(x)\right) \quad = \quad -\sum_{j=1}^{\infty} t_j\, x^j/j \tag{40.3-5}$$

Differentiating both sides gives

$$\frac{R'(x)}{R(x)} \quad = \quad -\sum_{j=1}^{\infty} t_j\, x^{j-1} \tag{40.3-6}$$

Using Newton's method for the inversion we obtain a computational cost of $\gamma\, M(n)$ where $M(n)$ is the cost for the multiplication of two power series up to order x^n and γ is a constant. The constant γ equals three if the division is performed by one inversion, which is two multiplications with the second order Newton iteration, and one final multiplication with $R'(x)$. For large n the multiplications should be done by either one of the splitting schemes suggested in [46] or by FFT methods such as given in [209].

40.4 Solving quadratic equations

We want to solve, in $\mathrm{GF}(2^n)$, the equation

$$a\, x^2 + b\, x + c \quad = \quad 0 \tag{40.4-1}$$

The fact that extracting a square root of an arbitrary element in $\mathrm{GF}(2^n)$ is easy does not enable us to solve the given equation. The formula $r_{0,1} = \left(-b \pm \sqrt{b^2 - 4ac}\right)/(2a)$ that works fine for real and complex numbers is of no help here: how should we divide by two?

Instead we transform the equation into a special form: divide by a: $x^2 + (b/a)\, x + (c/a) = 0$, substitute $x = z\,(b/a)$ to get $z^2\,(b/a)^2 + (b/a)^2\, z + (c/a) = 0$, and divide by $(b/a)^2$ to obtain

$$z^2 + z + C \quad = \quad 0 \quad \text{where} \quad C = \frac{a\, c}{b^2} \tag{40.4-2}$$

If r_0 is one solution of this equation then $r_1 = r_0 + 1$ is the other one: $z\,(z+1) = C$. The equation does not necessarily have a solution at all, the trace of C must be zero because we have $\mathrm{Tr}(C) = \mathrm{Tr}(z^2 + z) = \mathrm{Tr}(z^2) + \mathrm{Tr}(z) = \mathrm{Tr}(z) + \mathrm{Tr}(z) = 0$ for all $z \in \mathrm{GF}(2^n)$.

The function [FXT: `gf2n_solve_quadratic()` in bpol/gf2n-solvequadratic.cc] transforms the equation into the reduced form:

```
bool gf2n_solve_quadratic(GF2n a, GF2n b, GF2n c, GF2n& r0, GF2n& r1)
// Solve  a*x^2 + b*x + c == 0
// Return  whether solutions exist.
// If so, the solutions are written to r0 and r1.
{
    GF2n cc = a*c;
    cc /= (b.sqr());  // cc = (a*c)/(b*b)
    GF2n r;
    bool q = gf2n_solve_reduced_quadratic(cc, r);
    if ( !q )  return  false;

    GF2n s = b / a;
    r0 = r * s;
    r1 = (r+GF2n::one) * s;

    return  true;
}
```

```
       n = 5  GF(2^n)
       c = 1..1.1 == x^5 + x^2 + 1  (polynomial modulus)
       mm= .11111   == 31 (prime)   (maximal order)
       h = .1.... (aux. bitmask)
       g = ....1. (element of maximal order)
       tv= ..1..1 (traces of x^i)
     tr1e= .....1 (element with trace=1)

        k:   f:=g**k  Tr(f)   RootOf(z^2+z=f)
        0:    ....1    1
        1:    ...1.    0      .1..1
        2:    ..1..    0      .1.11
        3:    .1...    1
        4:    1....    0      .1111
        5:    ..1.1    1
        6:    .1.1.    1
        7:    1.1..    0      ..1..
        8:    .11.1    0      11111
        9:    11.1.    1
       10:    1...1    1
       11:    ..111    1
       12:    .111.    1
       13:    111..    1
       14:    111.1    0      1....
       15:    11111    0      11..1
       16:    11.11    0      1..1.
       17:    1..11    1
       18:    ...11    1
       19:    ..11.    0      ...1.
       20:    .11..    1
       21:    11...    1
       22:    1.1.1    1
       23:    .1111    0      1.11.
       24:    1111.    1
       25:    11..1    0      11.11
       26:    1.111    1
       27:    .1.11    0      111.1
       28:    1.11.    0      .11.1
       29:    .1..1    0      1.1..
       30:    1..1.    0      ..11.
```

Figure 40.4-A: Solutions of the equation $z^2 + z = f$ for all elements $f \in \mathrm{GF}(2^5)$ with trace zero.

The following function checks whether the reduced equation has a solution and if so, returns true and writes one solution to the variable `r`:

```
bool
gf2n_solve_reduced_quadratic(GF2n c, GF2n& r)
// Solve  z^2 + z == c
// Return  whether solutions exist.
// If so, one solutions is written to r.
// The other solution is r+1.
{
    if ( 1==c.trace() )  return  false;

    GF2n t( GF2n::tr1e );
    GF2n z( GF2n::zero );
    GF2n u( t );
    for (ulong j=1; j<GF2n::n_; ++j)
    {
        GF2n u2 = u.sqr();
        z = z.sqr();  z += u2*c;   // z = z*z + c*u*u
        u = u2 + t;                // u = u*u + t
    }
    r = z;

    return  true;
}
```

Figure 40.4-A shows the solutions to the reduced equations $x^2 + x = f$ for all elements f with trace zero [FXT: gf2n/gf2n-solvequadratic-demo.cc].

The implementation of the algorithm takes advantage of a precomputed element with trace one. At the

```
------- k=1: -------
    u=t^2 + t
    z=c*t^2
  z^2=c^2*t^4
z^2+z=c^2*t^4 + c*t^2

------- k=2: -------
    u=t^4 + t^2 + t
    z=(c^2 + c)*t^4 + c*t^2
  z^2=(c^4 + c^2)*t^8 + c^2*t^4
z^2+z=(c^4 + c^2)*t^8 + c*t^4 + c*t^2

------- k=3: -------
    u=t^8 + t^4 + t^2 + t
    z=(c^4 + c^2 + c)*t^8 + (c^2 + c)*t^4 + c*t^2
  z^2=(c^8 + c^4 + c^2)*t^16 + (c^4 + c^2)*t^8 + c^2*t^4
z^2+z=(c^8 + c^4 + c^2)*t^16 + c*t^8 + c*t^4 + c*t^2
     =(c^8 + c^4 + c^2)*t    + c*(t^8 + t^4 + t^2)    [using t^16=t]
     =( c + trace(c)  )*t    + c*( t + trace(t) )     [using x^8+x^4+x^2=x+trace(x)]
     =( c + 0  )*t    + c*( t + 1 )                   [using trace(c)=0, trace(t)=1]
     =                          c                     [ z^2+z==c ]
```

Figure 40.4-B: Solving the reduced quadratic equation $z^2 + z = c$ in GF(2^4).

end of step $k \geq 1$ we have

$$u_k = \sum_{j=0}^{k} t^{2^k} \tag{40.4-3a}$$

$$z_k = \sum_{j=0}^{k-1} \left[t^{2^k} \sum_{i=0}^{k-1} c^{2^i} \right] \tag{40.4-3b}$$

Figure 40.4-B shows (for GF(2^4)) that this expression is the solution sought.

Routines for the solution of quadratic equations that do not depend on the class `GF2n` are given in [FXT: bpol/bitpolmod-solvequadratic.cc].

For GF(2^n) with n odd the solution of the reduced quadratic equation $z^2 + z = A$ can be computed via the *half-trace* of A which is defined as

$$H(A) = A + A^4 + A^{16} + \ldots + A^{4^{(n-1)/2}} \tag{40.4-4}$$

We have $H(A)^2 + H(A) = \mathrm{Tr}(A) + A$, so $H(A)$ is a solution of the reduced quadratic if $\mathrm{Tr}(A) = 0$. The half-trace of an element A in the field with field polynomial C can be computed via [FXT: `gf2n_half_trace()` in bpol/gf2n-trace.cc]:

```
ulong
gf2n_half_trace(ulong a, ulong c, ulong h)
{
    ulong t = a;
    ulong d = h;
    while ( d>>=2 )
    {
        t = bitpolmod_square(t, c, h);
        t = bitpolmod_square(t, c, h);
        t ^= a;
    }
    return  t;
}
```

40.5 Representation by matrices *

With a primitive polynomial modulus $C(x)$ a representation of the elements of GF(2^n) as matrices can be obtained from the powers of the primitive element ('x') in a surprisingly simply way: in the table of

```
n = 4  GF(2^n)
c = 1..11 == x^4 + x + 1  (polynomial modulus)

  k:  f:=g**k
  0:   ...1
  1:   ..1.
  2:   .1..        1...1..11.1.111
  3:   1...        .1..11.1.1111..
  4:   ..11        ..1..11.1.1111.
  5:   .11.        ...1..11.1.1111
  6:   11..
  7:   1.11
  8:   .1.1          M_0 =    M_1 =    M_2 =    M_3 =    M_4 =     M_5 =
  9:   1.1.          1...     ...1     ..1.     .1..     1..1      ..11
 10:   .111          .1..     1..1     ..11     .11.     11.1      1.1.
 11:   111.          ..1.     .1..     1..1     ..11     .11.      11.1
 12:   1111          ...1     ..1.     .1..     1..1     ..11      .11.
 13:   11.1
 14:   1..1
```

Figure 40.5-A: Powers of the primitive element in GF(2^4) with field polynomial $x^4 + x + 1$ (left), the list of powers rotated counter clockwise by 90 degree (top right), and the matrices obtained by taking 4 consecutive columns of the list (bottom right).

---

powers of a generator take row k through $k+n-1$ to obtain the columns of matrices M_k, see figure 40.5-A. Now one has $M_k = M_1^k$ so one can use the matrices to represent the elements of GF(2^n).

The matrix $M := M_1$ is the *companion matrix* of the polynomial modulus g. The companion matrix of a polynomial $p(x) = x^n - \sum_{i=0}^{n-1} c_i\, x^i$ of degree n is defined as the $n \times n$ matrix

$$M \quad = \quad \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & c_0 \\ 1 & 0 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & 0 & \cdots & 0 & c_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & c_{n-2} \\ 0 & 0 & 0 & \cdots & 1 & c_{n-1} \end{bmatrix} \tag{40.5-1}$$

For polynomials $p(x) = \sum_{i=0}^{n} a_i\, x^i$ set $c_i := -a_i/a_n$.

The *characteristic polynomial* $c(x)$ of a $n \times n$ matrix M is defined as

$$c(x) \quad := \quad \det(x\, E_n - M) \tag{40.5-2}$$

where E_n is the $n \times n$ unit matrix. The roots of characteristic polynomial are the eigenvalues of the matrix. The characteristic polynomial of the companion matrix of a polynomial $p(x)$ equals $p(x)$. If $p(x)$ is the characteristic polynomial of a matrix M then $p(M) = 0$ (non-proof: set $x = M$ in relation 40.5-2, see [76] for a proof).

```
  k:    [ p_k(x) ]^d          k:    [ p_k(x) ]^d
--------------------        --------------------
  0:    [    11 ]^4           7:    [ 1..11 ]^1
  1:    [ 11..1 ]^1           8:    [ 11..1 ]^1
  2:    [ 11..1 ]^1           9:    [ 11111 ]^1
  3:    [ 11111 ]^1          10:    [   111 ]^2
  4:    [ 11..1 ]^1          11:    [ 1..11 ]^1
  5:    [   111 ]^2          12:    [ 11111 ]^1
  6:    [ 11111 ]^1          13:    [ 1..11 ]^1
                             14:    [ 1..11 ]^1
```

Figure 40.5-B: Characteristic polynomials of the powers of the generator x with the field GF(2^4) and the polynomial $x^4 + x + 1$.

Let $c_k(x)$ be the characteristic polynomial of the matrix $M_k = M^k$ and $p_k(x)$ the minimal polynomial of the element $g^k \in$ GF(2^n). Then

$$c_k(x) \quad = \quad [p_k(x)]^d \quad \text{where} \quad d = n/r \tag{40.5-3}$$

where r is the smallest positive integer so that $M_k^{2^r} = M_k$ For example, for the primitive modulus $C(x) = x^4 + x + 1$ (as above) the sequence of characteristic polynomials of the powers of the generator 'x' are shown in figure 40.5-B.

The trace of the matrix M^k is the d-th power of the *polynomial trace* of the minimal polynomial of g^k. The polynomial trace of $p(x) = x^n - (c_{n-1}\,x^{n-1} + \cdots + c_1\,x + c_0)$ equals c_{n-1} as can be seen from relation 40.5-1 on the facing page.

By construction, picking the first column of M_k gives the vector of the coefficients of the polynomial x^k modulo $C(x)$:

$$M^k\,[1, 0, 0, \ldots, 0]^T \quad \equiv \quad x^k \pmod{C(x)} \tag{40.5-4}$$

Finally, the characteristic polynomial of an element $a \in \mathrm{GF}(2^n)$ in polynomial representation can be written as

$$p_a(x) \quad := \quad \prod_{k=0}^{n-1} \left(x - a^{2^k}\right) \tag{40.5-5}$$

Compare to relation 40.2-1 on page 857 for minimal polynomials.

40.6 Representation by normal bases

As a vector space over GF(2), we so far used the n basis vectors $x^0, x^1, x^2, x^3, \ldots, x^{n-1}$ to represent an element in $\mathrm{GF}(2^n)$. The arithmetic operations were the polynomial operations modulo an irreducible polynomial modulus C.

For certain irreducible polynomials it is possible to use the *normal basis* $x^1, x^2, x^4, x^8, \ldots, x^{2^{n-1}}$ to represent elements of $\mathrm{GF}(2^n)$. These polynomials are called *normal polynomials* or *N-polynomials*. To check whether a polynomial C is normal, compute $r_k = x^{2^k} \bmod C$ for $1 \le k \le n$, compute the nullspace of the matrix M whose k-th row is r_k. If the nullspace is empty (that is, $M \cdot v = 0$ implies $v = 0$) then the polynomial is normal.

The normality of a polynomial is equivalent to the fact that its roots are linearly independent (see section 38.4 on page 808 for the equivalence of computations modulo a polynomial and computations with linear combinations of its roots).

Addition and subtraction with a normal basis is again a simple XOR. Squaring an element can be achieved by a cyclic shift by one position. Note that $(x^{2^{n-1}})^2 = x^1$. Taking the square root is a cyclic shift in the other direction.

In normal basis representation the element one is the all-ones word. Thereby adding one is equivalent to complementing the binary word.

The trace can be computed easily with normal bases, it equals the parity of the binary word.

40.6.1 Multiplication

Multiplication of two elements is achieved via a *multiplication matrix* M. Given two elements $a, b \in \mathrm{GF}(2^n)$ in normal basis representation their product $p = a \cdot b$ is computed as follows: let a_k be the coefficient (zero or one) of x^{2^k} where $0 \le k \le n-1$. Then, for the first component p_0 of the product,

$$p_0 \quad = \quad a^T \cdot M \cdot b \tag{40.6-1}$$

and in general

$$p_k \quad = \quad \left(a^{-2^{k-1}}\right)^T \cdot M \cdot b^{-2^{k-1}} \tag{40.6-2}$$

That is, all components of the product are computed like the first but with a and b cyclically shifted.

A routine that checks whether a given polynomial c is normal and, if so, computes the multiplication matrix M, is [FXT: `bitpol_normal_q()` in bpol/bitpol-normal.cc] which proceeds as follows:

1. If the polynomial c is reducible, return false.
2. Compute the matrix A whose k-th row equals $x^{2^k} \bmod c$. If A is not invertible then (the nullspace is not empty and) c is not normal, so return false.
3. Set $D := A \cdot C^T \cdot A^{-1}$ where C is the companion matrix of c.
4. Compute the multiplication matrix M where $M_{i,j} := D_{j',i'}$, $i' := -i \bmod n$ and $j' := j - i \bmod n$. Return (true and) the matrix M.

Examples of the intermediate results for two different field polynomials are given in figure 40.6-A.

```
Normal poly:  c=11111 =^= 4,3,2,1,0

    A=          A^-1=
      .1..        1111
      ..1.        1...
      1111        .1..
      ...1        ...1

    C^T=        D=A*C^T*A^{-1}=
      .1..        .1..
      ..1.        ...1
      ...1        1111
      1111        ..1.

    Multiplication matrix:  M=
      ..1.
      ..11
      11..
      .1.1
```

```
Normal poly:  c=111.11 =^= 5,4,2,1,0

A=            A^-1=
  .1...         11111
  ..1..         1....
  ....1         .1...
  .1.1.         1..1.
  1.111         ..1..

C^T=          D=A*C^T*A^{-1}=
  .1...         .1...
  ..1..         1..1.
  ...1.         ...11
  ....1         .11..
  111.1         ..1.1

Multiplication matrix:  M=
  .1...
  1..1.
  ...11
  .11..
  ..1.1
```

Figure 40.6-A: Matrices that occur with the computation of the multiplication matrix for the field polynomials $c = 1 + x + x^2 + x^3 + x^4$ (left) and $c = 1 + x + x^2 + x^4 + x^5$ (right).

A C++ function implementing the multiplication algorithm is [FXT: `normal_mult()` in bpol/normal-mult.cc]:

```
ulong
normal_mult(ulong a, ulong b, const ulong *M, ulong n)
// Multiply two elements (a and b in GF(2^n))
//  in normal basis representation.
// The multiplication matrix has to be supplied in M.
{
    ulong p = 0;
    for (ulong k=0; k<n; ++k)
    {
        ulong v = bitmat_mult_Mv(M, n, b);
        v = parity( v & a );  // dot product
        p ^= ( v << k );
        a = bit_rotate_right(a, 1, n);
        b = bit_rotate_right(b, 1, n);
    }
    return  p;
}
```

We note that the algorithm is much more attractive for hardware implementations than for software, see [94].

40.6.2 Solving the reduced quadratic equation

```
Normal poly:  c=1111.1 == x^5 + x^4 + x^3 + x^2 + 1

   k =       :   f=g**k  Tr(f)   x^2+x==f
   0 = ....1 :    11111    1
   1 = ...1. :    ...1.    1
   2 = ...11 :    .111.    1
   3 = ..1.1 :    1.111    .      x=.11.1
   4 = ..11. :    111..    1
   5 = ..111 :    .11.1    1
   6 = .1... :    .1...    1
   7 = .1..1 :    111.1    .      x=.1.11
   8 = .1.1. :    .1111    .      x=..1.1
   9 = .1.11 :    ..11.    .      x=...1.
  10 = .11.. :    11..1    1
  11 = .11.1 :    11...    .      x=.1...
  12 = .111. :    11.1.    1
  13 = .1111 :    1.1..    .      x=.11..
  14 = 1.... :    1....    1
  15 = 1...1 :    ..111    1
  16 = 1..1. :    11.11    .      x=.1..1
  17 = 1..11 :    1.11.    1
  18 = 1.1.. :    1111.    .      x=.1.1.
  19 = 1.1.1 :    ...11    .      x=....1
  20 = 1.11. :    .11..    .      x=..1..
  21 = 1.111 :    .1.1.    .      x=..11.
  22 = 11... :    1..11    1
  23 = 11..1 :    .1.11    1
  24 = 11.1. :    1...1    .      x=.1111
  25 = 11.11 :    ..1.1    .      x=...11
  26 = 111.. :    1.1.1    1
  27 = 111.1 :    1..1.    .      x=.111.
  28 = 1111. :    .1..1    .      x=..111
```

Figure 40.6-B: Solving the reduced quadratic equation $x^2 + x = f$ for powers $f = g^k$ of the generator $g = x$. The equation is solvable if the trace is zero, that is, the number of bits in the normal representation is even. The (primitive) field polynomial is $1 + x^2 + x^3 + x^4 + x^5$.

The reduced quadratic equation $x^2 + x = f$ has two solutions if $\mathrm{Tr}(f) = 0$. If so, one solution $x = [x_0, x_1, \ldots, x_{n-1}]$ can be obtained as $x_k = \sum_{j=0}^{k} f_k$ where $f = [f_0, f_1, \ldots, f_{n-1}]$. This follows from observing that

$$x^2 + x \quad = \quad [x_0 + x_{n-1},\, x_0 + x_1,\, x_1 + x_2,\, \ldots,\, x_{n-2} + x_{n-1},\, x_{n-1} + x_0] \tag{40.6-3}$$

Now equate $x^2 + x = f$ and set $x_{n-1} = 0$ (setting $x_{n-1} = 1$ gives the complement which is also a solution). In C++ this translates to (see page 29) [FXT: `normal_solve_reduced_quadratic()` in bpol/normal-solvequadratic.cc]

```
ulong
normal_solve_reduced_quadratic(ulong c)
// Solve x^2 + x = c
// Must have:  trace(c)==0
// Return one solution x, the other solution equals 1+x,
// that is, the complement of x.
{
    return  inverse_rev_gray_code(c);
}
```

The highest bit of the result is zero exactly if the equation is solvable. The reversed Gray code is given in section 1.15.6 on page 41.

The program [FXT: gf2n/normalbasis-demo.cc] prints the powers of x in normal basis representation, see figure 40.6-B. By default a primitive normal polynomial from [FXT: bpol/normal-primpoly.cc] is used. If more than one argument is given the arguments should give the nonzero coefficients of the polynomial modulus.

40.6.3 The number of binary normal bases *

n :	A_n	n :	A_n	n :	A_n	n :	A_n
1:	1	11:	93	21:	27783	31:	28629151
2:	1	12:	128	22:	95232	32:	67108864
3:	1	13:	315	23:	182183	33:	97327197
4:	2	14:	448	24:	262144	34:	250675200
5:	3	15:	675	25:	629145	35:	352149525
6:	4	16:	2048	26:	1290240	36:	704643072
7:	7	17:	3825	27:	1835001	37:	1857283155
8:	16	18:	5376	28:	3670016	38:	3616800768
9:	21	19:	13797	29:	9256395	39:	5282242875
10:	48	20:	24576	30:	11059200	40:	12884901888

Figure 40.6-C: The number A_n of degree-n binary normal polynomials up to $n = 40$.

n :	B_n	n :	B_n	n :	B_n	n :	B_n
1:	1	11:	87	21:	23579	31:	28629151
2:	1	12:	52	22:	59986	32:	33552327
3:	1	13:	315	23:	178259	33:	78899078
4:	1	14:	291	24:	103680		
5:	3	15:	562	25:	607522		
6:	3	16:	1017	26:	859849		
7:	7	17:	3825	27:	1551227		
8:	7	18:	2870	28:	1815045		
9:	19	19:	13797	29:	9203747		
10:	29	20:	11255	30:	5505966		

Figure 40.6-D: The number B_n of degree-n binary primitive normal polynomials up to $n = 33$.

The number A_n of degree-n binary normal polynomials up to $n = 40$ is given in figure 40.6-C. A table of the values A_n for $1 \le n \le 130$ and their factorizations is given in [FXT: data/num-normalpoly.txt]. The sequence A_n is sequence A027362 of [214].

The number B_n of degree-n binary primitive normal polynomials up to $n = 30$ is given in figure 40.6-D. This is sequence A107222 of [214].

40.6.3.1 Computation via exhaustive search

For small degrees all normal polynomials can be generated by selecting from the irreducible polynomials those that are normal. Using the mechanism that generate all irreducible polynomials via Lyndon words that is described in section 38.6 on page 824, the computation is a matter of minutes for $n < 25$.

The program [FXT: gf2n/all-normalpoly-demo.cc] prints all normal polynomials of a given degree n and marks those that are primitive with a '`P`'. The Output for $n = 9$ is

```
 1:  c=11..11...1  P
 2:  c=11...1..11  P
 3:  c=11111...11  P
 4:  c=11.11.1.11  P
 5:  c=111.....1.1  P
 6:  c=111...1111  P
 7:  c=11.....1
 8:  c=11.111..11  P
 9:  c=1111.1.1.1  P
10:  c=1111..1.11  P
11:  c=11.1.11.11  P
12:  c=11.1...1.1
13:  c=1111111.11  P
14:  c=1111...111  P
15:  c=11...1.1.1  P
16:  c=11.1..1111  P
17:  c=11...11111  P
```

```
18:  c=111.111..1  P
19:  c=1111.11..1  P
20:  c=11..111.11  P
21:  c=11.11.11.1  P
```

We can compute the number of normal (=: A_n) and primitive normal (=: B_n) binary polynomials for a small degrees n using that program. The table of the values B_n in figure 40.6-D was produced with the mentioned program, the computation up to $n = 30$ takes about 90 minutes. As noted in [123], no formula for the number of primitive normal polynomials is presently known. The proof that primitive normal bases exist for all finite extension fields was given 1987 in [168].

40.6.3.2 Cycles in the De Bruijn graph

Quite surprisingly, it turns out that A_n equals the number of cycles in the De Bruijn graph (see sections 19.2.2 on page 359 and 39.4 on page 838). Thereby for n a power of two the number A_n equals the number of binary De Bruijn sequences of length $2\,n$. No isomorphism between both objects (paths and polynomials) is presently known.

40.6.3.3 Invertible circulant matrices

Moreover, A_n equals the number of invertible circulant $n \times n$ matrices over GF(2).

```
      arg 1: 6 == n  [  n x n - matrices]  default=6
      arg 2: 2 == wh  [What to do: 0==>just count  1==>print words
                                   2==>also print matrix]  default=2

v0 = 1..... [I]      v0 = 1.11.. [I]
      M =                  M =
     1.....               1.11..
     .1....               .1.11.
     ..1...               ..1.11
     ...1..               1..1.1
     ....1.               11..1.
     .....1               .11..1

v0 = 111... [S]      v0 = 11111. [I]      v0 = 11.1.. [I]
      M =                  M =                  M =
     111...               11111.               11.1..
     .111..               .11111               .11.1.
     ..111.               1.1111               ..11.1
     ...111               11.111               1..11.
     1...11               111.11               .1..11
     11...1               1111.1               1.1..1

   n=6   #invertible=4   #singular=1
```

Figure 40.6-E: The length-6 Lyndon words of odd weight and the corresponding circulant matrices. Singular matrices are marked with '`[S]`', invertible matrices with '`[I]`'.

This is demonstrated in [FXT: gf2n/bitmat-circulant-demo.cc] whose output for $n = 6$ is shown in figure 40.6-E. The search uses the Lyndon words as periodic words would trivially lead to singular matrices. Further, Lyndon words with an even number of bits can be skipped as the vector $[1, 1, 1, \ldots, 1]$ is in the nullspace of the corresponding matrices.

If the set $\{\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots, \alpha^{2^{n-1}}\}$ is a normal basis of GF(2^n) we say that α generates the normal basis. Considering the rows of a circulant matrix as some element β in a normal basis representation then the following rows are $\beta^2, \beta^4, \beta^8, \ldots, \beta^{2^{n-1}}$ and the matrix is invertible if β generates a normal basis. If α generates a normal basis then an element $\beta = \sum_{i=0}^{n-1} a_i\, \alpha^{2^i}$ generates a normal basis exactly if the polynomial $\sum_{i=0}^{n-1} a_i\, x^i$ is relatively prime to $x^n - 1$. Thereby, with a fast algorithm to generate Lyndon words, determine all elements that generate normal bases if one such element is known as follows: select the Lyndon words with an odd number of ones and test whether $\gcd(L(x), x^n - 1) = 1$ where $L(x)$ is the binary polynomial corresponding to the Lyndon word. If n is a power of two, then $x^n - 1 = (x - 1)^n$ and all Lyndon words with an odd number of ones are coprime to $x^n - 1$.

```
L(x) = 111.11.  W(x) = 1....11        L(x) = 1.1.1..  W(x) = 1..1111
       M =             M^-1 =                M =             M^-1 =
       111.11.         1....11               1.1.1..         1..1111
       .111.11         11....1               .1.1.1.         11..111
       1.111.1         111....               ..1.1.1         111..11
       11.111.         .111...               1.1.1.1         1111.1.
       .11.111         ..111..               .1.1.1.1        11111..
       1.11.11         ...111.               1.1..1.         .11111.
       11.11.1         ....111               .1.1..1         ..11111
```

Figure 40.6-F: The inverse of a $n \times n$ circulant matrix over GF(2) can be found by computing the inverse $W(x)$ of its first row as a polynomial $L(x)$ modulo $x^n - 1$.

If the Lyndon word under consideration is taken as a polynomial $L(x)$ over GF(2) then the corresponding matrix is invertible exactly if $gcd(L(x), x^n - 1) = 1$. The first row inverse of a circulant matrix over GF(2) can be found by computing $W(x) = L(x)^{-1} \bmod x^n - 1$ where $L(x)$ is the binary polynomial with coefficients one where the Lyndon word has a one. As the inverse of a circulant matrix is also circulant, the remaining rows are cyclic shifts of $W(x)$. Two examples with $n = 7$ are shown in figure 40.6-F.

40.6.3.4 Factorization of $x^n - 1$

The factorization of the polynomial $x^n - 1$ over GF(2) can be used for the computation of A_n. The file [FXT: data/polfactdeg.txt] supplies the necessary information:

```
# Structure of the factorization of x^n-1 over GF(2):
 1:  [1]  [1*1]
 2:  [2]  [1*1]
 3:  [1]  [1*1 + 1*2]
 4:  [4]  [1*1]
 5:  [1]  [1*1 + 1*4]
 6:  [2]  [1*1 + 1*2]
 7:  [1]  [1*1 + 2*3]
 8:  [8]  [1*1]
 9:  [1]  [1*1 + 1*2 + 1*6]
10:  [2]  [1*1 + 1*4]
11:  [1]  [1*1 + 1*10]
12:  [4]  [1*1 + 1*2]
13:  [1]  [1*1 + 1*12]
14:  [2]  [1*1 + 2*3]
15:  [1]  [1*1 + 1*2 + 3*4]
16: [16]  [1*1]
17:  [1]  [1*1 + 2*8]
```

An entry: `n: [e] [m1*d1 + m2*d2 + ... ]` says that $(x^n - 1) = P(x)^e$ and $P(x)$ factors into `m1` different irreducible polynomials of degree `d1`, `m2` different irreducible polynomials of degree `d2` and so on. As an example, for $n = 6$ we have

$$x^6 - 1 \;=\; \left[x^3 - 1\right]^2 \;=\; \left[(x+1)\left(x^2 + x + 1\right)\right]^2 \tag{40.6-4}$$

x^6 is the square ($e = 2$) of a product of one irreducible polynomial of degree one and one of degree two. Therefore we have the entry: `6: [2] [1*1 + 1*2]`. Another example, $n = 15$,

$$x^{15} - 1 \;=\; \left[(x+1)\left(x^2 + x + 1\right)\left(x^4 + x + 1\right)\left(x^4 + x^3 + 1\right)\left(x^4 + x^3 + x^2 + x + 1\right)\right]^1 \tag{40.6-5}$$

corresponding to the entry `15: [1] [1*1 + 1*2 + 3*4]`.

Now one has for the number of normal polynomials:

$$A_n \;=\; \frac{2^n}{n} \prod_i \left(1 - \frac{1}{2^{d_i}}\right)^{m_i} \tag{40.6-6}$$

Note that the quantity e does not appear in the formula. For example, with $n = 6$ and $n = 15$ we obtain

$$A_6 \;=\; \frac{2^6}{6} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 \;=\; \frac{64}{6} \cdot \frac{1}{2} \cdot \frac{3}{4} \;=\; 4 \tag{40.6-7a}$$

$$A_{15} \;=\; \frac{2^{15}}{15} \cdot \left(1 - \frac{1}{2^1}\right)^1 \cdot \left(1 - \frac{1}{2^2}\right)^1 \cdot \left(1 - \frac{1}{2^4}\right)^3 \;=\; 675 \tag{40.6-7b}$$

40.6.3.5 Efficient computation

It is actually possible to compute the number of degree-n normal binary polynomials without explicitly factorizing $x^n - 1$. We have

$$x^n - 1 \quad = \quad \prod_{d \backslash n} Y_d(x) \tag{40.6-8}$$

where $Y_d(x)$ is the d-th cyclotomic polynomial (see section 38.7 on page 826). We further know that $Y_d(x)$ factors into $\varphi(d)/r$ polynomials of degree r where $r = \mathrm{ord}_d(2)$ is the order of 2 modulo d. Let $a_n := A_n / \left(\frac{2^n}{n}\right)$, then a_n can for odd n be computed as

$$a_n \quad = \quad \prod_{d \backslash n} \left(1 - \frac{1}{2^r}\right)^{\varphi(d)/r} \tag{40.6-9}$$

The following pari/gp code works for any prime p:

```
p=2 /* global */
num_normal_p(n)=
{
    local( r, i, pp );
    pp = 1;
    fordiv (n, d,
        r = znorder(Mod(p,d));
        i = eulerphi(d)/r;
        pp *= (1 - 1/p^r)^i;
    );
    return( pp );
}
```

The number A_n can be computed (for arbitrary n) as $A_n = a_q \left(\frac{2^n}{n}\right)$ where q odd, and $n = q\,2^t$:

```
num_normal(n)=
{
    local( t, q, pp );
    t = 1;  q = n;
    while ( 0==(q%p), q/=p; t+=1; );
    /* here: n==q*p^t */
    pp = num_normal_p(q);
    pp *= p^n/n;
    return( pp );
}
```

The quantity t is not used in the computation. The implementation is quite efficient: the computation of A_n for all $n \le 10,000$ takes less than three seconds. The computation of A_n for $n = 1234567 = 127 \cdot 9721$ (A_n is a number with 371,636 decimal digits) takes about 200 milliseconds.

40.6.4 Dual normal bases

Let $A = \{a_0, a_1, a_2, \cdots, a_{n-1}\}$ be a basis of GF(2^n). A basis $B = \{b_0, b_1, b_2, \cdots, b_{n-1}\}$ is said to be the *dual basis* (or *complementary basis*) of A if

$$\mathrm{Tr}\,(a_k\, b_j) \quad = \quad \delta_{k,j} \qquad \text{for} \quad 0 \le k, j < n \tag{40.6-10}$$

A basis that is its own dual is called *self-dual*. We treat only normal basis here. If α is a root of a normal polynomial C then $A = \{\alpha, \alpha^2, \alpha^4, \cdots, \alpha^{2^{n-1}}\}$ is a normal basis.

A necessary condition for C to be normal is that $\gcd(T, x^n - 1) = 1$ where

$$T \quad = \quad t_0 + t_1\, x + t_2\, x^2 + \ldots + t_{n-1}\, x^{n-1} \tag{40.6-11}$$

with $t_k = \mathrm{Tr}(\alpha\, \alpha^{2^k})$. The polynomial T can be computed via [FXT: bpol/normalpoly-dual.cc]

```
            C              T              C*       D = T^-1 (mod x^n-1)

     1:  11..11...1  P  .11....111     1111.1.1.1  .1.1111.11
     2:  11...1..11  P  .1..11..11     111....1.1  ..111111.1
     3:  11111...11  P  .1.1111.11     11..111.11  .11....111
     4:  11.11.1.11  P  ..1.11.1.1     11111111.1  .111..1111
     5:  111....1.1  P  ..111111.1     11...1..11  .1..11..11
     6:  111...1111  P  ..111111.1     11...1.1.1  .1..11..11
     7:  11.......1     .11....111     1111.11..1  .1.1111.11
     8:  11.111..11  P  .........1  S  11.111..11  .........1
     9:  1111.1.1.1  P  .1.1111.11     11..11...1  .11....111
    10:  1111..1.11  P  .111..1111     11.1.11.11  ..1.11.1.1
    11:  11.1.11.11  P  ..1.11.1.1     1111..1.11  .111..1111
    12:  11.1..1..1     .........1  S  11.1..1..1  .........1
    13:  1111111.11  P  .111..1111     11.11.1.11  ..1.11.1.1
    14:  1111...111  P  .111..1111     11.11.11.1  ..1.11.1.1
    15:  11...1.1.1  P  .1..11..11     111...1111  ..111111.1
    16:  11.1..1111  P  .........1  S  11.1..1111  .........1
    17:  11...11111  P  .1..11..11     111.111..1  ..111111.1
    18:  111.111..1  P  ..111111.1     11...11111  .1..11..11
    19:  1111.11..1  P  .1.1111.11     11.......1  .11....111
    20:  11..111.11  P  .11....111     11111...11  .1.1111.11
    21:  11.11.11.1  P  ..1.11.1.1     1111...111  .111..1111
```

Figure 40.6-G: All normal polynomials C of degree 9 and their polynomials T (left), their duals C^* and $D = T^{-1}$ (right). Primitive polynomials C are marked with '`P`', self-dual $C = C^*$ are marked with '`S`'.

```
ulong
gf2n_ntrace(ulong c, ulong deg)
// Return vector V of traces v[k]=trace(ek), where
//   ek = x*x^(2^k), k=0..deg-1, and
//   x is a root of the irreducible polynomial C.
// Must have:  deg == degree(C)
{
    const ulong tv = gf2n_trace_vector_x(c, deg); // traces of x^k
    ulong rt = 2UL;  // root of C
    const ulong h = 1UL << (deg-1);  // aux
    ulong v = 0;
    for (ulong k=0; k<deg; ++k)
    {
        ulong ek = bitpolmod_times_x(rt, c, h);  // == x*x^(2^k)
        ulong tk = gf2n_fast_trace(ek, tv);      // == sum(ek[i]*tk[i])
        v |= (tk<<k);
        rt = bitpolmod_square(rt, c, h);
    }
    return  v;
}
```

Now if C is normal then T has an inverse $D \equiv T^{-1} \bmod x^n - 1$. Write

$$D \;=\; d_0 + d_1\,x + d_2\,x^2 + \ldots + d_{n-1}\,x^{n-1} \tag{40.6-12a}$$

then

$$\beta \;=\; d_0\,\alpha + d_1\,\alpha^2 + d_2\,\alpha^4 + \ldots + d_{n-1}\,\alpha^{2^{n-1}} \tag{40.6-12b}$$

is a root of a normal polynomial so $B = \{\beta, \beta^2, \beta^4, \cdots, \beta^{2^{n-1}}\}$ is the dual (normal) basis of A. The following routine computes T, D, B, and finally the minimal polynomial C^* of β:

```
ulong
gf2n_dual_normal(ulong c, ulong deg, ulong ntc/*=0*/, ulong *ntd/*=0*/)
// Return the minimal polynomial CS for the dual (normal) basis
//  with the irreducible normal polynomial C.
// Return zero if C is not normal.
// Must have:  deg == degree(C).
// If ntc is supplied it must be equal to gf2n_ntrace(c, deg).
// If ntd is nonzero, ntc^-1 (mod x^deg-1) is written to it.
{
    if ( 0==ntc )  ntc = gf2n_ntrace(c, deg);
    const ulong d = bitpolmod_inverse(ntc, 1 | (1UL<<deg) );  // ntc=d^-1 (mod x^deg-1)
```

```
    if ( 0==d )  return 0;  // C not normal
    if ( 0!=ntd )  *ntd = d;

    const ulong h = 1UL << (deg-1);  // aux
    ulong alpha = 2UL;  // 'x', a root of C
    ulong beta = 0;     // root of the dual polynomial
    for (ulong m=d; m!=0; m>>=1)
    {
        if ( m & 1 )  beta ^= alpha;
        alpha = bitpolmod_square(alpha, c, h);
    }

    ulong cs;  // minimal polynomial of beta
    bitpolmod_minpoly(beta, c, deg, cs);

    return cs;
}
```

Figure 40.6-G shows the normal bases of degree 9 and their duals, it was created with the program [FXT: gf2n/normalpoly-dual-demo.cc]. A (complicated) expression for the number of self-dual normal bases is given in [143].

$n:$	S_n	$n:$	S_n	$n:$	S_n	$n:$	S_n	$n:$	S_n
1:	0	9:	3	17:	17	25:	205	33:	3267
2:	1	10:	4	18:	48	26:	320	34:	4352
3:	1	11:	3	19:	27	27:	513	35:	
4:	0	12:	0	20:	0	28:	0	36:	
5:	1	13:	5	21:	63	29:	565	37:	
6:	2	14:	8	22:	96	30:	1920	38:	
7:	1	15:	15	23:	89	31:	961	39:	
8:	0	16:	0	24:	0	32:	0	40:	

$n:$	Z_n	$n:$	Z_n	$n:$	Z_n	$n:$	Z_n	$n:$	Z_n
1:	0	9:	2	17:	17	25:	200	33:	2660
2:	1	10:	3	18:	25	26:	215	34:	2917
3:	1	11:	3	19:	27	27:	428	35:	
4:	0	12:	0	20:	0	28:	0	36:	
5:	1	13:	5	21:	57	29:	562	37:	
6:	1	14:	4	22:	60	30:	997	38:	
7:	1	15:	11	23:	87	31:	961	39:	
8:	0	16:	0	24:	0	32:	0	40:	

Figure 40.6-H: Number of self-dual normal basis (S_n) and self-dual primitive normal basis (Z_n).

40.7 Conversion between normal and polynomial representation

If the field polynomial C is normal then conversion between the representations in polynomial and normal basis can be achieved as follows: Let Z be the $n \times n$ matrix whose k-th column equals $x^{2^k} \bmod C$ where n is the degree of C. If a is the polynomial representation then the normal representation is $b = Z^{-1} \cdot a$ (both a and b shall be column vectors).

The implementation [FXT: `class GF2n` in bpol/gf2n.h] allows the conversion to the normal representation if the field polynomial is normal. In the initializer [FXT: `GF2n::init()` in bpol/gf2n.cc] the matrix Z (`n2p_tab[]`) and Z^{-1} (`p2n_tab[]`) are computed with the lines

```
    // conversion to and from normal representation:
    for (ulong k=0,s=2;  k<n_;  ++k)
    {
```

```
 k = bin(k):   f= g**k   == (normal)    trace(f)
 0 = ....0 :      ...1  ==  11111       1
 1 = ...1. :      ..1.  ==  ...1.       1
 2 = ...10 :      .1..  ==  .111.       1
 3 = ..1.. :      1...  ==  ..1..       1
 4 = ..1.1 :      111.  ==  1.111       .
 5 = ..110 :      .111  ==  111..       1
 6 = .1... :      .11.  ==  .11.1       1          P2N=
 7 = .1..1 :      111.  ==  .1...       1            11...
 8 = .1.1. :      ..1.  ==  111.1       .            1.11.
 9 = .1.10 :      .1.1  ==  .1111       .            1..11
10 = .1100 :      1.1.  ==  ..11.       .            1..1.
11 = .11.. :      1.1.  ==  11..1       1            1....
12 = .11.1 :      1.11  ==  11...       .
13 = .1110 :      1..1  ==  11.1.       1
14 = .1111 :      11.1  ==  1.1..       .           N2P=
15 = 1.... :      .1.1  ==  1....       1             ....1
16 = 1...1 :      1.11  ==  ..111       1             1..1
17 = 1..1. :      1...  ==  11.11       .             .1.1.
18 = 1..10 :      11111  ==  1.11.       1             ..11
19 = 1.1.. :      ...11  ==  1111.       .             ..11.
20 = 1.1.1 :      ..11.  ==  ...11       .
21 = 1.110 :      .11..  ==  .11..       .
22 = 1.111 :      11...  ==  .1.1.       .
23 = 11... :      .11.1  ==  1..11       1
24 = 11..1 :      11.1.  ==  .1.11       1
25 = 11.10 :      .1..1  ==  1...1       .
26 = 11.11 :      1..1.  ==  ..1.1       .
27 = 111.. :      11..1  ==  1.1.1       1
28 = 111.1 :      .1111  ==  1..1.       .
29 = 1110. :      1111.  ==  .1..1       .
```

Figure 40.7-A: Conversion between normal- and polynomial representation with the (primitive) polynomial $c = 1 + x^2 + x^3 + x^4 + x^5$. The conversion matrices are given as `P2N` and `N2P`.

```
        n2p_tab[k] = s;
        s = bitpolmod_square(s, c_, h_);
    }
    bitmat_transpose(n2p_tab, n_, n2p_tab);
    is_normal_ = bitmat_inverse(n2p_tab, n_, p2n_tab);
```

The last line records whether the field polynomial is normal which is the case exactly if Z is invertible.

The functions [FXT: bpol/gf2n.cc]

```
ulong // static
GF2n::p2n(ulong f)
{
    return  bitmat_mult_Mv(p2n_tab, n_, f);
}
ulong // static
GF2n::n2p(ulong f)
{
    return  bitmat_mult_Mv(n2p_tab, n_, f);
}
```

allow conversions between the normal and polynomial representations. The method

```
    ulong get_normal()  const  { return p2n(v_); }
```

provides a convenient way to obtain the normal representation of a given element.

This is demonstrated in [FXT: gf2n/gf2n-normal-demo.cc] where both the polynomial and the normal representation are given, see figure 40.7-A.

If the last argument of the initialization routine of the C++ class `GF2n`, `init(n, c, normalq)`, is set then a (primitive) normal polynomial will be used as field polynomial. A list of primitive normal polynomials is given in [FXT: bpol/normal-primpoly.cc].

40.8 Optimal normal bases (ONB)

The number of nonzero terms in the multiplication matrix determine the complexity (operation count) for the multiplication with normal bases. It turns out that for certain values of n there are normal bases of GF(2^n) whose multiplication matrices have at most two nonzero entries in each row (and column). Such bases are called *optimal normal bases* (ONB).

Optimal normal bases are especially interesting for hardware implementations because of both the highly regular structure of the multiplication algorithm and the minimal complexity with ONBs.

40.8.1 Type-1 optimal normal bases

A so-called *type-1 optimal normal basis* exists for n when $p := n + 1$ is prime and 2 is a primitive root modulo p (and for $n = 0$ and $n = 1$). The sequence of such n starts

```
0, 1, 2, 4, 10, 12, 18, 28, 36, 52, 58, 60, 66, 82, 100, 106, 130, 138, 148,
162, 172, 178, 180, 196, 210, 226, 268, 292, 316, 346, 348, 372, 378, 388,
418, 420, 442, 460, 466, 490, 508, ...
```

This is entry A071642 of [214]. One has always $n \equiv 2$ or $n \equiv 4$ modulo 8. A list of the corresponding primes is given in figure 39.6-B on page 842. The field polynomial corresponding to a type-1 ONB is the all-ones polynomial

$$c \;=\; \frac{x^p - 1}{x - 1} \;=\; 1 + x + x^2 + x^3 + \ldots + x^n \tag{40.8-1}$$

These polynomials are non-primitive for all $n \neq 2$.

```
Normal poly:  c=11111111111

A=              A^-1=
  .1........      1111111111
  ..1.......      1.........
  ....1.....      .1........
  ........1.      ........1.
  .....1....      ..1.......
  1111111111      ....1.....
  .........1      .........1
  .......1..      .......1..
  ...1......      ...1......
  ......1...      ......1...

C^T=            D=A*C^T*A^{-1}=     Multiplication matrix:  M=
  .1........      .1........             .....1....
  ..1.......      .........1             ....11....
  ...1......      ....1.....             ...1...1..
  ....1.....      ......1...             ..1.....1.
  .....1....      .........1             .......1.1
  ......1...      1111111111             11........
  .......1..      ....1.....             .1.......1
  ........1.      ...1......             ..1.1.....
  .........1      ..1.......             ...1..1...
  1111111111      .......1..             ....1....1
```

Figure 40.8-A: Matrices that occur with the computation of the multiplication matrix for the field polynomial $c = 1 + x + \ldots + x^{10}$.

The multiplication matrices are sparse: there is exactly one entry in the first row and column and exactly two in the other rows and columns. That is, the multiplication matrices for GF(n) with optimal normal basis have exactly $2n-1$ nonzero entries. For example, with $n = 10$ we obtain the matrix shown (together with the intermediate results) in figure 40.8-A. The equivalent data for $n = 4$ is shown in figure 40.6-A on page 866.

```
 0: [0]  1
 1: [1]  11
 2: [1]  111
 3: [1]  11.1  <--= x^4 + x^3 + 1
 4: [0]  111.1
 5: [1]  11.111
 6: [1]  111..11
 7: [0]  11.1...1
 8: [0]  111.1...1
 9: [1]  11.111..11
10: [0]  111..11.111
11: [1]  11.1...111.1
12: [0]  111.1....11.1
13: [0]  11.111.....111
14: [1]  111..11......11
15: [0]  11.1...1.......1
16: [0]  111.1...1.......1
17: [0]  11.111..11......11
18: [1]  111..11.111.....111
19: [0]  11.1...111.1....11.1
20: [0]  111.1....11.1...111.1
21: [0]  11.111.....111..11.111
22: [0]  111..11......11.111..11
23: [1]  11.1...1.......111.1...1
24: [0]  111.1...1........11.1...1
25: [0]  11.111..11.........111..11
26: [1]  111..11.111..........11.111
27: [0]  11.1...111.1...........111.1
28: [0]  111.1....11.1...........11.1
29: [1]  11.111.....111.............111
30: [1]  111..11......11.............11
31: [0]  11.1...1.......1.............1
32: [0]  111.1...1.......1.............1
33: [1]  11.111..11......11.............11
34: [0]  111..11.111.....111............111
35: [1]  11.1...111.1....11.1...........11.1
36: [0]  111.1....11.1...111.1...........111.1
37: [0]  11.111.....111..11.111..........11.111
38: [0]  111..11......11.111..11.........111..11
39: [1]  11.1...1.......111.1...1........11.1...1
40: [0]  111.1...1........11.1...1.......111.1...1
```

Figure 40.8-B: The polynomials p_k up to $k = 40$ as binary strings. The entry in the second column is '`[1]`' is the polynomial is irreducible (a field polynomial for a type-2 optimal normal basis).

40.8.2 Type-2 optimal normal bases

A *type-2 optimal normal basis* exists for n if $p := 2\,n + 1$ is prime and either

- $n \equiv 1$ or $n \equiv 2$ modulo 4 and the order of 2 modulo p equals $2\,n$.
- $n \equiv 3$ mod 4 and the order of 2 modulo p equals n.

A type-2 basis exists for the following $n \leq 200$:

```
1, 2, 3, 5, 6, 9, 11, 14, 18, 23, 26, 29, 30, 33, 35, 39, 41, 50,
51, 53, 65, 69, 74, 81, 83, 86, 89, 90, 95, 98, 99, 105, 113, 119,
131, 134, 135, 146, 155, 158, 173, 174, 179, 183, 186, 189, 191, 194
```

The corresponding normal polynomial (see figure 40.8-B) of degree n can be computed via the recursion

$$p_0 := 1, \qquad p_1 := x + 1 \tag{40.8-2a}$$

$$p_k := x\,p_{k-1} + p_{k-2} \tag{40.8-2b}$$

Compare to the recursion that transforms a linear hybrid cellular automaton into a binary polynomial relation 39.7-1 on page 844: the type-2 ONBs correspond to the most trivial LHCA defined by the rule having a single one as the lowest bit of the rule word.

As with type-1 ONBs, the multiplication matrices are sparse. The polynomials and multiplication matrices for $n = 6$ and $n = 9$ are

```
6,5,4,1,0      9,8,6,5,4,1,0
.1....         .1.....
i...i.         i...i...
..ii1.         ....1..i
.i1..i         .ii....i.
.ii...         ...i..ii.
...i.i         ...i.i...
                ..i..1...
                ...i....i
```

The intermediate values with the computation of the multiplication matrix for $n = 5$ are shown in figure 40.6-A on page 866.

We note a relation of the polynomials p_k to the *Fibonacci polynomials* defined by

$$f_0 := 0, \qquad f_1 := 1 \tag{40.8-3a}$$

$$f_k := x\, f_{k-1} + f_{k-2} \tag{40.8-3b}$$

We have

$$p_k^2 = f_{2k+1} \tag{40.8-4}$$

40.9 Gaussian normal bases

The *type-t Gaussian normal basis* (GNB) generalize the optimal normal basis. The type-1 and type-2 GNBs are the corresponding ONBs. The multiplication matrices for type-t GNBs for $t > 2$ have more nonzero entries than the ONBs.

A type-t GNB exists for n when $p := t\,n + 1$ is prime and $\gcd(n, t\,n/r_2) = 1$ where r_2 is the order of 2 modulo p. Implementation of the test using pari/gp:

```
gauss_test(n, t)=
{ /* test whether a type-t Gaussian normal basis exists for GF(2^n) */
    local( p, r2, g, d );
    p = t*n + 1;
    if ( !isprime(p), return( 0 ) );
    if ( p<=2, return( 0 ) );
    r2 = znorder( Mod(2, p) );
    d = (t*n)/r2;
    g = gcd( d, n);
    return ( if ( 1==g, 1, 0) );
}
```

For n divisible by 8 no GNB exist. If a type-t GNB exists it is unique.

40.9.1 Computation of the multiplication matrix

An algorithm that computes the multiplication matrix for a type-t GNB proceeds as follows (The algorithm uses a vector $F[1, 2, \ldots, p-1]$):

1. Set $p = t\,n + 1$ (this is a prime), and compute an element r of order t modulo p.
2. For $k = 0, 1, \ldots, t-1$ do the following: set $j = r^k$ and for $i = 0, 1, \ldots, n-1$ set $F[j\,2^i] = i$.
3. Set the multiplication matrix M to zero.
4. For $i = 1, 2, \ldots, p-2$ add one to $M_{F[p-i],F[i+1]}$.
5. If t is odd set $h = n/2$ and do the following: for $i = 0, 1, \ldots, h-1$ increment $M_{i,h+i}$ and $M_{h+i,i}$.

Implementation in pari/gp:

```
gauss_nb(n, t)=
{ /* return multiplier matrix for type-t Gaussian normal basis */
  /* returned matrix is over Z and has to be multiplied by Mod(1,2) */
    local(p, r, F, w, x, nh, m, ir, ic);
    if ( 0==gauss_test(n, t), print("No type-",t, " GNB for n=",n); return(0));
    p = t*n + 1;
    r = znprimroot(p);  r = r^(n);  /* r has order t */
    F = vector(p-1);
    w = Mod(1, p);
    for (k=0, t-1,
        j = lift(w);
        for (i=0, n-1,
            F[j] = i;
            j+=j;  if (j>=p, j-=p);  /* 2*j mod p */
        );
        w *= r;
    );

    m = matrix(n, n);
    for (i=1, p-2,
        ir = F[p-i];  ic = F[i+1];
        m[ ir+1, ic+1 ] += 1;
    );

    if ( 1==(t%2),
        nh = n/2; /* odd t ==> even n */
        for (i=0, nh-1,
            ir = i;  ic = nh + i;
            ir += 1;  ic += 1;
            m[ir, ic] += 1;
            m[ic, ir] += 1;
        );
    );

    return ( m );
}
```

```
        n=7, t=4                              n=12, t=3

     M=        M*Mod(1,2)=               M=            M*Mod(1,2)=
   .1..2..       .1.....           ....1.1...1.      ....1.1...1.
   1.1..11       1.1..11           ...1.111....      ...1.111....
   .1.111.       .1.111.           ...1...12...      ...1...1....
   ..12.1.       ..1..1.           .11..1...1..      .11..1...1..
   2.1...1       ..1...1           1......1.11.      1......1.11.
   .111..1       .111..1           .1.1.....1.1      .1.1.....1.1
   .1..111       .1..111           11....2.....      11..........
                                   .11.1.....1.      .11.1.....1.
                                   ..2......1.1      .........1.1
                                   ...111..1...      ...111..1...
                                   1...1..1...1      1...1..1...1
                                   .....1..1.11      .....1..1.11
```

Figure 40.9-A: Multiplication matrices over $\mathbb{Z}$ and $GF(2)$ for Gaussian normal bases with $n = 7$, $t = 4$ (left), and $n = 12$, $t = 3$ (right). Dots denote zeros.

The implementation computes M with entries in $\mathbb{Z}$ and has to be reduced modulo 2 before usage. Figure 40.9-A gives two examples.

The file [FXT: data/gauss-normalbasis.txt] lists for each $2 \le n \le 1032$ the smallest ten values of t so that there is a type-t GNB. Note that different values of t do not necessary lead to different multiplication matrices, especially for small values of n. For example, the modulo 2 reduced multiplication matrices for $n = 6$ and the 10 smallest values of t are:

```
 t=2:    t=3:    t=6:    t=10:    t=11:   t=23:   t=27:   t=30:   t=35:    t=55:
.1....  ..11.1  .1....  .1....   ..11.1  .1.11.  .1.11.  .1....  .1.11.   .1.11.
1...1.  ..11..  1.11.1  1.11.1   ..11..  1.111.  1.111.  1...1.  1.111.   1.111.
...11.  11..11  .1...1  .1...1   11..11  .1..1.  .1..1.  ...11.  .1..1.   .1..1.
..1..1  11....  .1..1.  .1..1.   11....  11....  11....  ..1..1  11....   11....
.11...  ..1..1  ...1.1  ...1.1   ..1..1  111..1  111..1  .11...  111..1   111..1
...1.1  1.1.11  .11.11  .11.11   1.1.11  ....11  ....11  ...1.1  ....11   ....11
```

40.9.2 Determination of the field polynomial

We give two algorithms for the determination of the field polynomial when n and t are given, the first uses computations with complex numbers, the second uses polynomial modular arithmetic over GF(2).

40.9.2.1 Algorithm with complex numbers

```
n=4 t=1:  p=5
  a(1)=2  w(1)=(+0.309016994374947 + 0.951056516295153 I)
  a(2)=4  w(2)=(-0.809016994374947 + 0.587785252292473 I)
  a(3)=8  w(3)=(+0.309016994374947 - 0.951056516295153 I)
  a(4)=6  w(4)=(-0.809016994374947 - 0.587785252292473 I)
z(x)=x^4 + x^3 + x^2 + x + 1
p(x)=x^4 + x^3 + x^2 + x + 1

n=4 t=3:  p=13
  a(1)=18  w(1)=(+0.65138781886599 + 0.52241580345640 I)
  a(2)=10  w(2)=(-1.15138781886599 + 1.72542218842200 I)
  a(3)=20  w(3)=(+0.65138781886599 - 0.52241580345640 I)
  a(4)=14  w(4)=(-1.15138781886599 - 1.72542218842200 I)
z(x)=x^4 + x^3 + 2*x^2 - 4*x + 3
p(x)=x^4 + x^3 + 1

n=4 t=7:  p=29
  a(1)=40  w(1)=(+1.09629120178362 - 2.64399848798350 I)
  a(2)=22  w(2)=(-1.59629120178362 - 0.50918758384404 I)
  a(3)=44  w(3)=(+1.09629120178362 + 2.64399848798350 I)
  a(4)=30  w(4)=(-1.59629120178362 + 0.50918758384404 I)
z(x)=x^4 + x^3 + 4*x^2 + 20*x + 23
p(x)=x^4 + x^3 + 1
```

Figure 40.9-B: Numerical values with the computation of the field polynomial for $n = 4$ and types $t \in \{1, 3, 7\}$. Note that the final result is identical for the types $t = 3$ and $t = 7$.

The normal polynomial corresponding to a type-t Gaussian basis can be computed with the following algorithm.

1. Set $p = t\,n + 1$ and determine r so that the order of r modulo p equals t.
2. For $1 \le k \le n$ compute $w_k = \sum_{j=0}^{t-1} \exp(a_k\,\pi\,i/p)$ where $a_k = 2^k\,r^j \bmod 2p$.
3. Let $z(x) = \prod_{k=1}^{n} (x - w_k)$, this is a polynomial with real integer coefficients.
4. Return the polynomial with coefficients reduced modulo 2.

The computation of the polynomial $z(x)$ uses complex (inexact) arithmetics. That its coefficients should be close to real integers can be used as a check.

The following pari/gp routine computes the complex polynomial. In order to keep the arguments for the exponential function small the values a_k are computed modulo $2\,p$, the periodicity of $\exp(.\,\pi\,i/p)$.

```
gauss_zpoly(n, t)=
{ /* return field polynomial for type-t Gaussian normal basis
     as polynomial over the complex numbers */

    local(p, r, wk, tk1, tk, a, zp);
    p = n*t + 1;
    r = znprimroot(p);  r = r^n;  \\ r has order t (mod p)
    r = Mod(lift(r), 2*p);

    zp = 1;
    tk1 = Mod(2,2*p);  tk = Mod(1,2*p);
    for (k=1, n,
        tk *= tk1;  \\ == Mod(2,2*p)^k;
        wk = 0;
        a = tk;
        for (j=0, t-1,
            wk += exp(1.0*I*Pi*lift(a)/p);
            a *= r;
        );
```

```
n=11 t=2:  p=23
  a(1)=44   w(1)=(+1.92583457469559 - 5.69206140 E-19 I)
  a(2)=42   w(2)=(+1.70883880909297 - 2.71050543 E-19 I)
  a(3)=38   w(3)=(+0.92013007546230 - 2.98155597 E-19 I)
  a(4)=30   w(4)=(-1.15336064422973 + 2.43945488 E-19 I)
  a(5)=14   w(5)=(-0.66975922434197 + 1.08420217 E-19 I)
  a(6)=28   w(6)=(-1.55142258140884 + 3.25260651 E-19 I)
  a(7)=10   w(7)=(+0.40691202610526 - 6.60685698 E-20 I)
  a(8)=20   w(8)=(-1.83442260301090 + 1.89735380 E-19 I)
  a(9)=40   w(9)=(+1.36510628643730 - 8.13151629 E-20 I)
  a(10)=34  w(10)=(-0.13648482672934 + 3.68459331 E-20 I)
  a(11)=22  w(11)=(-1.98137189207266 + 4.33680868 E-19 I)
z(x)=x^11 + x^10 - 10*x^9 - 9*x^8 + 36*x^7 + 28*x^6 - 56*x^5 \
      - 35*x^4 + 35*x^3 + 15*x^2 - 6*x - 1
p(x)=x^11 + x^10 + x^8 + x^4 + x^3 + x^2 + 1

n=11 t=6:  p=67
  [--snip--]
z(x)=x^11 + x^10 - 30*x^9 - 63*x^8 + 220*x^7 + 698*x^6 - 101*x^5 \
      - 1960*x^4 - 1758*x^3 - 35*x^2 + 243*x - 29
p(x)=x^11 + x^10 + x^8 + x^5 + x^2 + x + 1

n=11 t=8:  p=89
  [--snip--]
z(x)=x^11 + x^10 - 40*x^9 - 19*x^8 + 482*x^7 + 84*x^6 - 2185*x^5 \
      + 102*x^4 + 3152*x^3 - 781*x^2 + 57*x - 1
p(x)=x^11 + x^10 + x^8 + x^5 + x^2 + x + 1

n=11 t=18:  p=199
  [--snip--]
z(x)=x^11 + x^10 - 90*x^9 - 115*x^8 + 2349*x^7 + 943*x^6 - 26327*x^5 \
      + 21284*x^4 + 102168*x^3 - 217794*x^2 + 148930*x - 30647
p(x)=x^11 + x^10 + x^8 + x^7 + x^6 + x^5 + 1
```

Figure 40.9-C: Numerical values with the computation of the field polynomial for $n = 11$ and types $t \in \{2, 6, 8, 18\}$. The final results are identical for $t = 6$ and $t = 8$.

```
        zp *= (x-wk);
    );
    return ( zp );
}
```

The final step uses pari/gp's function `round()` which rounds all coefficients of its polynomial argument:

```
gauss_poly(n, t)=
{ /* return field polynomial for type-t Gaussian normal basis */
    local(pp, zp);
    zp = gauss_zpoly(n, t);
    pp = round(real(zp));  /* rounds all coefficients */
    pp *= Mod(1,2);  /* coefficients modulo 2 */
    return( pp );
}
```

The results for type-1 bases can be verified using relation 40.8-1 on page 875, results with type-2 bases with relations 40.8-2a... 40.8-2b on page 876. The intermediate values occurring with the computation for $n = 4$ and the types $t \in \{1, 3, 7\}$ are shown in figure 40.9-B. The values for $n = 11$ and the types $t \in \{2, 6, 8, 18\}$ are shown in figure 40.9-C.

40.9.2.2 Algorithm working in GF(2)

The following algorithm is a variation of what is given in [234].

1. Set $p = t\,n + 1$ and determine r so that the order of r modulo p equals t.
2. Set $M(x) = \sum_{k=0}^{p-1} x^k$. All computations are done modulo M.
3. If t equals 1 return M.
4. Set $F_0 = 1$ (modulo M).

```
  n=4  t=3:  p=13  \\ integer computation

     r= Mod(3, 13)  ord(r)=3 ==  t
     M= x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

     ------- k=1
     Z= x^9 + x^3 + x
     F= x^9 + x^3 + 2*x
     ------- k=2
     Z= x^6 + x^5 + x^2
     F= x^11 + x^10 + x^9 + x^8 + 2*x^7 + 2*x^6 + x^5 + x^4 + 2*x^3 + 3*x^2 + x
     ------- k=3
     Z= -x^11 - x^9 - x^8 - x^7 - x^6 - x^5 - x^3 - x^2 - x - 1
     F= -x^11 - x^10 - 2*x^9 + x^7 + 2*x^6 + x^3 - 1
     ------- k=4
     Z= x^11 + x^8 + x^7
     F= x^4 - x^3 + 2*x^2 + 4*x + 3

   ==> x^4 - x^3 + 2*x^2 + 4*x + 3  ==   x^4 + x^3 + 1 (mod 2)
```

```
  n=4  t=3:  p=13  \\ computation over GF(2)

     r= Mod(3, 13)  ord(r)=3 ==  t
     M= x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1

     ------- k=1
     Z= x^9 + x^3 + x
     F= x^9 + x^3 + 2*x
     ------- k=2
     Z= x^6 + x^5 + x^2
     F= x^11 + x^10 + x^9 + x^8 + x^5 + x^4 + x^2 + x
     ------- k=3
     Z= x^11 + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x^2 + x + 1
     F= x^11 + x^10 + x^7 + x^3 + 1
     ------- k=4
     Z= x^11 + x^8 + x^7
     F= x^4 + x^3 + 1

   ==>  x^4 + x^3 + 1
```

Figure 40.9-D: Computation of the field polynomial for $n = 4$ and $t = 3$ with polynomials over the integers (top) and polynomials over GF(2) (bottom).

5. For $1 \leq k \leq n$:
   (a) Set $Z_k = \sum_{j=0}^{t-1} x^{a(k,j)}$ (modulo M) where $a(k, j) = 2^k\, r^j \bmod p$.
   (b) Set $F_k = (x + Z_k)\; F_{i-1}$ (modulo M).
6. Return F_n with coefficients reduced modulo 2.

The intermediate quantities in the computation for $n = 4$ and $t = 3$ are shown at the top of figure 40.9-D. The result is a polynomial over the integers identical to the one computed with the algorithm that uses complex numbers. When all polynomials are taken over GF(2) the computation proceeds as shown at the bottom of figure 40.9-D. Implementation in pari/gp:

```
gauss_poly2(n, t)=
{  /* return field polynomial for type-t Gaussian normal basis */
    local(p, M, r, F, t21, t2, Z);

    p = t*n + 1;
    r = znprimroot(p)^n;  \\ element of order t mod p
    M = sum(k=0, p-1, 'x^k);  \\ The polynomial modulus
    M *= Mod(1,2);  \\ ... over GF(2)
    if ( 1==t,  return( M ) );  \\ for type 1

    F = Mod(1, M);
    t21 = Mod(2,p);  t2 = Mod(1,p);
    for (k=1, n,
        Z = sum(j=0, t-1, Mod('x^lift(t2*r^j), M) );
        F = ('x+Z)*F;
        t2 *= t21;
    );
    return ( lift(F) );
}
```

While the algorithm avoids inexact arithmetic the polynomial modulus M is of degree $p-1 = n\,t$ which can be large for large t. In practice the computation with complex numbers is much faster. It finishes in less than a second for $n = 620$ and $t = 3$ (and a working precision of 150 decimal digits) while the exact method needs about two minutes.

Appendix A

Machine used for benchmarking

The machine used for performance measurements is a AMD64 (Athlon64) clocked at 2.2 GHz with dual channel double data rate (DDR) clocked at 200 MHz ('800 MHz'). It has 512 kB (16-way associative) second level cache and separate first level caches for data and instructions, each 64 kB (and 2-way associative). Cache lines are 64 bytes (8 words, 512 bits). The memory controller is integrated in the CPU.

The CPU has 16 general purpose (64 bit) registers that are addressable as byte, 16 bit word, 32 bit word, or 64 bit (full) word. These are used for integer operations and for passing integer function arguments. There are 16 (128 bit, SSE) registers that are used for floating point operations and for passing floating point function arguments. The SSE registers are SIMD registers. Additionally, there are 8 (legacy, x87) FPU registers.

The performance-wise interesting information reported by the CPUID instruction is:

```
Vendor: AuthenticAMD
Name: AMD Athlon(tm) 64 Processor 3400+
  Family: 15,  Model: 15,  Stepping: 0
Level 1 cache (data):  64 kB,  2-way associative.
  64 bytes per line,  lines per tag: 1.
Level 1 cache (instr):  64 kB,  2-way associative.
  64 bytes per line,  lines per tag: 1.
Level 2 cache:  512 kB,  16-way associative
  64 bytes per line,  lines per tag: 1.

Max virtual addr width: 48
Max physical addr width: 40
Features:
  lm: Long Mode (64-bit mode)
  mtrr: Memory Type Range Registers
  tsc: Time Stamp Counter
  fpu: x87 FPU
  3dnow: AMD 3DNow! instructions
  3dnowext: AMD Extensions to 3DNow!
  mmx: Multimedia Extensions
  mmxext: AMD Extensions to MMX
  sse: Streaming SIMD Extensions
  sse2: Streaming SIMD Extensions-2
  cmov: CMOV instruction (plus FPU FCMOVCC and FCOMI)
  cx8: CMPXCHG8 instruction
  clflush: CLFLUSH instruction
  fxsr: FXSAVE and FXRSTOR instructions
```

Special instructions as SIMD, prefetch and non-temporal moves are not used unless explicitly noted.

See [126] for a comparison of instruction latencies and throughput for various x86 CPU cores. You do want to study the cited document *before* buying an x86-based system.

Appendix B

The pseudo language Sprache

Many algorithms in this book are given in a pseudo language called Sprache. Sprache is meant to be immediately understandable for everyone who ever had contact with programming languages like C, FORTRAN, Pascal or Algol. Sprache is hopefully self explanatory. The intention of using Sprache instead of completely relying an mathematical formulas (like tensor formalism) or algorithm description by words is to minimize the work it takes to translate the given algorithm to one's favorite programming language. It should be mere syntax adaptation.

By the way, 'Sprache' is the German word for language.

```
// a comment:
// comments are useful.

// assignment:
t := 2.71

// parallel assignment:
{s, t, u} := {5, 6, 7}
// same as:
s := 5
t := 6
u := 7

{s, t} := {s+t, s-t}
// same as (avoiding the temporary):
temp := s + t
t := s - t
s := temp

// if conditional:
if a==b then  a:=3

// with block
if a>=3 then
{
    // do something ...
}

// a function returns a value:
function plus_three(x)
{
    return  x + 3
}

// a procedure works on data:
procedure increment_copy(f[],g[],n)
// real f[0..n-1] input
// real g[0..n-1] result
{
    for k:=0 to n-1
    {
        g[k] := f[k] + 1
    }
}

// for loop with stepsize:
```

```
for i:=0 to n step 2  // i:=0,2,4,6,...
{
    // do something
}

// for loop with multiplication:
for i:=1 to 32 mul_step 2
{
    print i, ", "
}
```

will print `1, 2, 4, 8, 16, 32,`

```
// for loop with division:
for i:=32 to 8 div_step 2
{
    print i, ", "
}
```

will print `32, 16, 8,`

```
// while loop:
i:=5
while i>0
{
    // do something 5 times...
    i := i - 1
}
```

The usage of `foreach` emphasizes that no particular order is needed in the array access (so parallelization is possible):

```
procedure has_element(f[],x)
{
    foreach t in f[]
    {
        if t==x then return TRUE
    }
    return FALSE
}
```

Emphasize type and range of arrays:

```
real     a[0..n-1],    // has n elements (floating point reals)
complex  b[0..2**n-1]  // has 2**n elements (floating point complex)
mod_type m[729..1728]  // has 1000 elements (modular integers)
integer  i[]           // has ? elements (integers)
```

Arithmetical operators: `+, -, *, /, %` and `**` for powering. Arithmetical functions: `min(), max(), gcd(), lcm(), ...`

Mathematical functions: `sqr(), sqrt(), pow(), exp(), log(), sin(), cos(), tan(), asin(), acos(), atan(), ...`

Bitwise operators: `~, &, |, ^` for bit-wise complement, AND, OR, XOR, respectively. Bit shift operators: `A<<3` shifts (the integer) `A` 3 bits to the left `A>>1` shifts `A` 1 bits to the right.

Comparison operators: `==, !=, <, > ,<=, >=`

There is no operator '`=`' in Sprache, only '`==`' (for testing equality) and '`:=`' (assignment operator).

A well-known constant: `PI` $= 3.14159265\ldots$

The complex square root of minus one in the upper half plane: `I` $= \sqrt{-1}$

Boolean values: `TRUE` and `FALSE`

Logical operators: `NOT, AND, OR, XOR`

Modular arithmetic: `x := a * b mod m` shall do what it says, `i := a**(-1) mod m` shall set `i` to the modular inverse of `a`.

Appendix C

The pari/gp language

We give a short introduction to the pari/gp language.

From the manual page of pari [189] (slightly edited):

```
NAME       gp - PARI calculator

SYNOPSIS   gp [-emacs] [-f] [-test] [-q] [-s stacksize] [-p primelimit]

DESCRIPTION
       Invokes the PARI-GP calculator. This is an advanced programmable calcu-
       lator, which computes symbolically as  long  as  possible,  numerically
       where  needed,  and  contains  a  wealth  of number-theoretic functions
       (elliptic curves, class field theory...).  Its basic data types are

       integers, real numbers, exact rational numbers,  algebraic  numbers,
       p-adic numbers, complex numbers,
       modular integers,
       polynomials and rational functions,
       power series,
       binary quadratic forms,
       matrices, vectors, lists,
       character strings,
       and recursive combinations of these.
```

Interactive usage

To use pari/gp interactively, just type `gp` at your command line prompt. A startup message like the following will appear:

```
                          GP/PARI CALCULATOR Version 2.3.1 (released)
                       amd64 running linux (x86-64 kernel) 64-bit version
                compiled: Oct 11 2006, gcc-3.3.5 20050117 (prerelease) (SUSE Linux)
                          (readline v5.0 enabled, extended help available)

                               Copyright (C) 2000-2006 The PARI Group

PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT ANY
WARRANTY WHATSOEVER.

Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.

parisize = 8000000, primelimit = 500000
?
```

The question mark in the last line is a prompt, the program is waiting for your input.

```
? 1+1
%1 = 2
```

Here we successfully computed one plus one. Next we compute a factorial:

```
? 44!
%2 = 2658271574788448768043625811014615890319638528000000000
```

Integers are of unlimited precision, the practical limit is the amount of physical RAM. For floating point numbers, the precision (number of decimal digits) can be set as follows

```
? default(realprecision,55)
%3 = 55
? sin(1.5)
%4 = 0.997494986604054430941723371141487322706651425922115821 9
```

The history numbers `%N` (where `N` is a number) can be used to recall the result of a prior computation:

```
? %4
%5 = 0.997494986604054430941723371141487322706651425922115821 9
```

The output of the result of a calculation can suppressed using a semicolon at the end of the command. This can be useful for timing purposes:

```
? default(realprecision,10000)
%5 = 10000
? sin(2.5);
? ##
  ***   last result computed in 100 ms.
```

The command `##` gives the time used for the last computation.

The printing format can be set independently of the used precision:

```
? default(realprecision,10000);
? default(format,"g.15");
? sin(2.5)
%6 = 0.598472144103956
```

Command line completion is available, typing `si`, then the tab-key, gives a list of builtin functions whose names start with `si`:

```
? si
  sigma      sign       simplify   sin        sinh       sizebyte   sizedigit
```

You can get the help text by using the question mark, followed by the help topic:

```
? ?sinh
  sinh(x): hyperbolic sine of x.
```

A help overview is invoked by a single question mark

```
? ?
Help topics: for a list of relevant subtopics, type ?n for n in
  0: user-defined identifiers (variable, alias, function)
  1: Standard monadic or dyadic OPERATORS
  2: CONVERSIONS and similar elementary functions
  3: TRANSCENDENTAL functions
  4: NUMBER THEORETICAL functions
  5: Functions related to ELLIPTIC CURVES
  6: Functions related to general NUMBER FIELDS
  7: POLYNOMIALS and power series
  8: Vectors, matrices, LINEAR ALGEBRA and sets
  9: SUMS, products, integrals and similar functions
 10: GRAPHIC functions
 11: PROGRAMMING under GP
 12: The PARI community
```

Select a section by its number:

```
? ?7
  O                 deriv             eval              factorpadic
  intformal         padicappr         polcoeff          polcyclo
  poldegree         poldisc           poldiscreduced    polhensellift
  polinterpolate    polisirreducible  pollead           pollegendre
  polrecip          polresultant      polroots          polrootsmod
  polrootspadic     polsturm          polsubcyclo       polsylvestermatrix
  polsym            poltchebi         polzagier         serconvol
  serlaplace        serreverse        subst             substpol
  substvec          taylor            thue              thueinit
```

You should try both of the following

```
? ??tutorial
  displaying 'tutorial.dvi'.
? ??
  displaying 'users.dvi'.
```

A short overview (which you may want to print) of most functions can be obtained via

```
? ??refcard
  displaying 'refcard.dvi'.
```

A session can be ended by either entering `quit` or just hitting control-d.

Builtin operators and basic functions

There are the 'usual' operators `+`, `-`, `*`, `/`, `^` (powering), and `%` (modulo). The operator `\` gives the integer quotient without remainder. The assignment operator is `=`. C-style shortcuts are available, for example `t+=3` is the same as `t=t+3`.

The increment by one can be abbreviated as `t++`, the decrement as `t--`. [Technical note: these behave as the C-language pre-increment (and pre-decrement), that is the expression evaluates to `t+1`, not `t`. There is no post-increment or post-decrement in pari/gp.]

Comparison operators are `==`, `!=` (alternatively `<>`), `>`, `>=`, `<`, and `<=`. Logical operators are `&&` (and), (or), and `!` (not)

Bit-wise operations for integers are

```
  bitand  bitneg  bitnegimply  bitor  bittest  bitxor
```

and

```
shift(x,n): shift x left n bits if n>=0, right -n bits if n<0.
shiftmul(x,n): multiply x by 2^n (n>=0 or n<0)
```

One can also use the operators `>>` and `<<`, as in the C-language, and the shortcuts `>>=` and `<<=`.

An overview of basic functions is obtained as

```
? ?2
  Col         List        Mat         Mod         Pol         Polrev      Qfb
  Ser         Set         Str         Strchr      Strexpand   Strtex      Vec
  Vecsmall    binary      bitand      bitneg      bitnegimply bitor       bittest
  bitxor      ceil        centerlift  changevar   component   conj        conjvec
  denominator floor       frac        imag        length      lift        norm
  norml2      numerator   numtoperm   padicprec   permtonum   precision   random
  real        round       simplify    sizebyte    sizedigit   truncate    valuation
  variable
```

Here are a few:

```
sign(x): sign of x, of type integer, real or fraction.
max(x,y): maximum of x and y.
min(x,y): minimum of x and y.
abs(x): absolute value (or modulus) of x.

floor(x): floor of x = largest integer<=x.
ceil(x): ceiling of x=smallest integer>=x.
frac(x): fractional part of x = x-floor(x)
```

An overview of sums, products, and some numerical functions:

```
? ?9
  intcirc           intfouriercos     intfourierexp     intfouriersin
  intfuncinit       intlaplaceinv     intmellininv      intmellininvshort
  intnum            intnuminit        intnuminitgen     intnumromb
  intnumstep        prod              prodeuler         prodinf
  solve             sum               sumalt            sumdiv
  suminf            sumnum            sumnumalt         sumnuminit
  sumpos
```

For example:

```
sum(X=a,b,expr,{x=0}): x plus the sum (X goes from a to b) of expression expr.
prod(X=a,b,expr,{x=1}): x times the product (X runs from a to b) of expression.
```

Basic data types

Strings:

```
? a="good day!"
  "good day!"
```

Integers, floating-point numbers (real or complex), and complex integers:

```
? factor(239+5*I)
  [-I 1]
  [1 + I 1]
  [117 + 122*I 1]
```

Exact rationals:

```
? 2/3+4/5
  22/15
```

Modular integers:

```
? Mod(3,239)^77
  Mod(128, 239)
```

Vectors and matrices:

```
? v=vector(5,j,j^2)
  [1, 4, 9, 16, 25]
? m=matrix(5,5,r,c,r+c)
  [2 3 4 5 6]
  [3 4 5 6 7]
  [4 5 6 7 8]
  [5 6 7 8 9]
  [6 7 8 9 10]
```

The vector is a row vector, trying to right-multiply it with the matrix fails:

```
? t=m*v
  ***   impossible multiplication t_MAT * t_VEC.
```

The operator ~ transposes vectors (and matrices), we multiply with the column vector:

```
? t=m*v~
%14 = [280, 335, 390, 445, 500]~
```

The result is a column vector, note the tilde at the end of the line.

Vector indices start with one:

```
? t[1]
%15 = 280
```

Symbolic computations

Univariate polynomials:

```
? (1+x)^7
  x^7 + 7*x^6 + 21*x^5 + 35*x^4 + 35*x^3 + 21*x^2 + 7*x + 1
? factor((1+x)^6+1)
  [x^2 + 2*x + 2 1]
  [x^4 + 4*x^3 + 5*x^2 + 2*x + 1 1]
```

Power series:

```
? (1+x+O(x^4))^7
  1 + 7*x + 21*x^2 + 35*x^3 + O(x^4)
? log((1+x+O(x^4))^7)
  7*x - 7/2*x^2 + 7/3*x^3 + O(x^4)
```

Types can be nested, here we compute modulo the polynomial $1 + x + x^7$ with coefficients over GF(2):

```
? t=Mod(1+x, Mod(1,2)*(1+x+x^7))^77
  Mod(Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2), Mod(1, 2)*x^7 + Mod(1, 2)*x + Mod(1, 2))
? lift(t)  \\ discard modulo polynomial
  Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2)
? lift(lift(t))  \\ discard modulo polynomial, then modulus 2 with coefficient
  x^3 + x + 1
```

Symbolic computations are limited when compared to a computer algebra system: for example, multivariate polynomials cannot (yet) be factored, and there is no symbolic solver for polynomials.

An uninitialized variable evaluates to itself, as a symbol:

```
? hello
  hello
```

To create a symbol, prepend a tick:

```
? w=3
  3
? hello='w  /* the symbol w, not the value of w */
  w
```

Here is a method to create symbols:

```
? sym(k)=eval(Str("A", k))
? t=vector(5, j, sym(j-1))
  [A0, A1, A2, A3, A4]
```

The ingredients are `eval()` and `Str()`:

```
eval(x): evaluation of x, replacing variables by their value.
Str({str}*): concatenates its (string) argument into a single string.
```

Some more trickery to think about:

```
sym(k)=eval(Str("A", k))
t=vector(5, j, sym(j-1));  print("1: t=", t);
{ for (k=1, 5,
    sy = sym(k-1);
    v = 1/k^2;
    /* assign to the symbol that sy evaluates to, the value of v: */
    eval( Str( Str( sy ), "=", Str( v) ) );
 ); }
print("2: t=", t); /* no lazy evaluation with pari/gp */
t=eval(t);   print("3: t=", t);
```

The output of this script is

```
1: t=[A0, A1, A2, A3, A4]
2: t=[A0, A1, A2, A3, A4]
3: t=[1, 1/4, 1/9, 1/16, 1/25]
```

More builtin functions

The following constants and transcendental functions are known by pari:

```
? ?3
  Euler       I           Pi          abs         acos        acosh       agm         arg
  asin        asinh       atan        atanh       bernfrac    bernreal    bernvec     besselh1
  besselh2    besseli     besselj     besseljh    besselk     besseln     cos         cosh
  cotan       dilog       eint1       erfc        eta         exp         gamma       gammah
  hyperu      incgam      incgamc     lngamma     log         polylog     psi         sin
  sinh        sqr         sqrt        sqrtn       tan         tanh        teichmuller theta
  thetanullk  weber       zeta
```

To obtain information about a particular function, use a question mark:

```
? ?sinh
  sinh(x): hyperbolic sine of x.
```

Transcendental functions will also work with complex arguments and symbolically, returning a power series:

```
? sinh(x)
%9 = x + 1/6*x^3 + 1/120*x^5 + 1/5040*x^7 + 1/362880*x^9 \
     + 1/39916800*x^11 + 1/6227020800*x^13 + 1/1307674368000*x^15 + O(x^17)
```

The line break (and the backslash indicating it) was manually entered for layout reasons. The 'precision' (that is default order) of power series can be set by the user:

```
? default(seriesprecision,9);
? sinh(x)
%11 = x + 1/6*x^3 + 1/120*x^5 + 1/5040*x^7 + 1/362880*x^9 + O(x^10)
```

One can also manually give the $O(x^N)$ term:

```
? sinh(x+O(x^23))
%12 = x + 1/6*x^3 + 1/120*x^5 + 1/5040*x^7 + \
   [--snip--] \
  + 1/121645100408832000*x^19 + 1/5109094217170944000*x^21 + O(x^23)
```

Functions operating on matrices are (type `mat`, then hit the tab-key)

```
  matadjoint        matalgtobasis     matbasistoalg     matcompanion
  matdet            matdetint         matdiagonal       mateigen
  matfrobenius      mathess           mathilbert        mathnf
  mathnfmod         mathnfmodid       matid             matimage
  matimagecompl     matindexrank      matintersect      matinverseimage
  matisdiagonal     matker            matkerint         matmuldiagonal
  matmultodiagonal  matpascal         matrank           matrix
  matrixqz          matsize           matsnf            matsolve
  matsolvemod       matsupplement     mattranspose
```

Builtin number theoretical functions are

```
? ?4
  addprimes     bestappr      bezout        bezoutres     bigomega      binomial
  chinese       content       contfrac      contfracpnqn  core          coredisc
  dirdiv        direuler      dirmul        divisors      eulerphi      factor
  factorback    factorcantor  factorff      factorial     factorint     factormod
  ffinit        fibonacci     gcd           hilbert       isfundamental ispower
  isprime       ispseudoprime issquare      issquarefree  kronecker     lcm
  moebius       nextprime     numbpart      numdiv        omega         precprime
  prime         primepi       primes        qfbclassno    qfbcompraw    qfbhclassno
  qfbnucomp     qfbnupow      qfbpowraw     qfbprimeform  qfbred        qfbsolve
  quadclassunit quaddisc      quadgen       quadhilbert   quadpoly      quadray
  quadregulator quadunit      removeprimes  sigma         sqrtint       zncoppersmith
  znlog         znorder       znprimroot    znstar
```

Functions related to polynomials and power series are

```
? ?7
  O                 deriv             eval              factorpadic
  intformal         padicappr         polcoeff          polcyclo
  poldegree         poldisc           poldiscreduced    polhensellift
  polinterpolate    polisirreducible  pollead           pollegendre
  polrecip          polresultant      polroots          polrootsmod
  polrootspadic     polsturm          polsubcyclo       polsylvestermatrix
  polsym            poltchebi         polzagier         serconvol
  serlaplace        serreverse        subst             substpol
  substvec          taylor            thue              thueinit
```

Plenty to explore!

Control structures for programming

Some loop constructs available are

`while(a,seq)`: while a is nonzero evaluate the expression sequence seq. Otherwise 0.

`until(a,seq)`: evaluate the expression sequence seq until a is nonzero.

`for(X=a,b,seq)`: the sequence is evaluated, X going from a up to b.

`forstep(X=a,b,s,seq)`: the sequence is evaluated, X going from a to b in steps of s (can be a vector of steps)

`forprime(X=a,b,seq)`: the sequence is evaluated, X running over the primes between a and b.

`fordiv(n,X,seq)`: the sequence is evaluated, X running over the divisors of n.

The expression `seq` is a list of statements:

```
for ( k=1, 10,  stat1; stat2; stat3; )  /* last semicolon optional */
for ( k=1, 10,  stat1; )
for ( k=1, 10,  ; )  /* zero statements (do nothing, ten times) */
```

(The comments enclosed in `/* */` were manually added.)

The loop-variable is local to the loop:

```
? for(k=1,10, ; )  /* do nothing, ten times */
? k
  k /* not initialized in global scope ==> returned as symbol */
```

A global variable of the same name is not changed:

```
? k=7
  7
? for(k=1,3, print("  k=",k))
  k=1
  k=2
  k=3
? k
  7 /* global variable k not modified */
```

For the sake of clarity, avoid using global and loop-local variables of the same name.

A loop can be aborted with the statement `break()`. The n enclosing loops are aborted by `break(n)`. With `next()`, the next iteration of a loop is started (and the statements until the end of the loop are skipped). With `break(n)` the same is done for the n-th enclosing loop.

And yes, there is an `if` statement:

```
if(a,seq1,seq2): if a is nonzero, seq1 is evaluated, otherwise seq2. seq1 and seq2
    are optional, and if seq2 is omitted, the preceding comma can be omitted also.
```

To have more than one statement in the branches use semicolons between the statements:

```
if ( a==3, /* then */
    b=b+1;
    c=7;
    , /* else */
    b=b-1;
    c=0;
   );
```

Non-interactive usage (scripts)

Usually one will create scripts that are fed into gp (at the command line):

```
gp -q < myscript.gp
```

The option `-q` suppresses the startup message and the history numbers `%N`.

If the script contains just the line

```
exp(2.0)
```

the output would be

```
  7.389056098930650227230427460575007813 1
```

To also see the commands in the output, add a `default(echo,1);` to the top of the file. The output will then be

```
? exp(2.0)
  7.389056098930650227230427460575007813 1
```

You should use comments in your scripts, there are two types of them:

```
\\ a line comment, started with backslashes
/* a block comment
   can stretch over several lines, as in the C-language */
```

Comments are not visible in the output. With the script

```
default(echo, 1);
\\ sum of square numbers:
s=0;  for (k=1, 10, s=s+k*k);  s
```

the output would be

```
? default(echo,1);
? s=0;for(k=1,10,s=s+k*k);s
  385
```

Note that all whitespaces are suppressed in the output.

A command can be broken into several lines if it is inclosed into a pair of braces:

```
{ for (k=1, 10,
       s=s+k*k;
       print(k,":  s=", s);
  ); }
```

This is equivalent to the one-liner

```
 for (k=1, 10,  s=s+k*k;  print(k,":  s=", s);  );
```

User-defined Functions

Now we define a function:

```
powsum(n, p)=
{ /* return the sum 1^p+2^p+3^p+...+n^p */
    local(t);
    t = 0;
    for (k=1, n,
         t = t+k^p);  \\ '^' is the powering operator
    return( t );
}
```

The statement `local(t);` makes sure that no global variable named `t` (if it exists) would be changed by the function. It must be the first statement in the function. The variable `k` in the `for()`-loop is automatically local and should not be listed with the locals. Note that each statement is terminated with a semicolon. The output would be

```
? powsum(n,p)=local(t);t=0;for(k=1,n,t=t+k^p);return(t);
? powsum(10,2)
  385
```

Note how the function definition is changed to a one-liner in the output.

If you have to use global variables, list them at the beginning of your script as follows:

```
global(var1, var2, var3);
```

Any attempt to use the listed names as names of function arguments or local variables in functions will trigger an error.

Arguments are passed by value. There is no mechanism for passing by reference, global variables can be a workaround for this.

Arguments can have defaults, as in

```
powsum(n, p=2)= /* etc */
```

Calling the function as either `powsum(9)` or `powsum(9,)` would compute the number of the first 9 squares. Defaults can appear anywhere in the argument list, as in

```
abcsum(a, b=3, c)= return( a+b+c );
```

So `abcsum(1,,1)` would return 5.

All arguments are implicitly given the default zero, so the sequence of statements

```
foo(a, b, c)= print(a,":",b,":",c);
foo(,,)
foo()
foo
```

will print three times `0:0:0`. This feature is rarely useful and does lead to obscure errors. It will hopefully be removed in future versions of pari/gp.

Bibliography

— A —

[1] Milton Abramowitz, Irene A. Stegun, (eds.): **Handbook of Mathematical Functions**, National Bureau of Standards, 1964, third printing, (1965). 665, 667, 669, 674

[2] Ramesh C. Agarwal, James W. Cooley: **New algorithms for digital convolution**, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-25, pp.392-410, (October-1977). 525

[3] Simon Joseph Agou, Marc Deléglise, Jean-Louis Nicolas: **Short Polynomial Representations for Square Roots Modulo** p, Designs, Codes and Cryptography, vol.28, pp.33-44, (2003). 751

[4] Manindra Agrawal, Neeraj Kayal, Nitin Saxena: **PRIMES is in P**, Annals of Math., vol.160, no.2, pp.781-793, (September-2004). Online at `http://www.math.princeton.edu/~annals/`. 770

[5] Omran Ahmadi, Alfred Menezes: **Irreducible polynomials of maximum weight**, Utilitas Mathematica, vol.72, pp.111-123, (2007). Online at `http://www.math.uwaterloo.ca/~ajmeneze/research.html`. 822

[6] Omran Ahmadi, Alfred Menezes: **On the number of trace-one elements in polynomial bases for** $\mathbb{F}_{2^n}$, Designs, Codes and Cryptography, vol.37, no.3, pp.493-507, (December-2005). Online at `http://citeseer.ist.psu.edu/677930.html`. 861

[7] W. R. Alford, Andrew Granville, Carl Pomerance: **There are infinitely many Carmichael numbers**, Annals of Mathematics, vol.139, pp.703-722, (1994). 753

[8] Jean-Paul Allouche, Jeffrey Shallit: **The ubiquitous Prouhet-Thue-Morse sequence**, In: C. Ding, T. Helleseth, H. Niederreiter, (eds.), Sequences and Their Applications: Proceedings of SETA'98, pp.1-16, Springer-Verlag, (1999). Online at `http://www.cs.uwaterloo.ca/~shallit/papers.html`. 40, 694

[9] Jean-Paul Allouche, Michael Cosnard: **The Komornik-Loreti constant is transcendental**, Amer. Math. Monthly, vol.107, pp.448-449, (2000). Online at `http://www.lri.fr/~allouche/`. 694

[10] Jean-Paul Allouche, Jeffrey Shallit: **Automatic Sequences**, Cambridge University Press, (2003). 334

[11] Advanced Micro Devices (AMD) Inc.: **AMD Athlon Processor, x86 code optimization guide**, Publication no.22007, Revision K, (February-2002). Online at `http://developer.amd.com/`. 19

[12] Advanced Micro Devices (AMD) Inc.: **AMD64 Architecture Programmer's Manual. Volume 3: General-Purpose and System Instructions**, Publication no.24594, Revision 3.11, (December-2005). Online at `http://developer.amd.com/`. 25

[13] Advanced Micro Devices (AMD) Inc.: **Software Optimization Guide for AMD64 Processors**, Publication no.25112, Revision 3.06, (September-2005). Online at `http://developer.amd.com/`. 7

[14] Ray Andraka: **A survey of CORDIC algorithms for FPGA based computers**, In: Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98), ACM, pp.191-200, (1998). Online at `http://www.andraka.com/papers.htm`. 634

[15] George E. Andrews, Richard Askey, Ranjan Roy: **Special functions**, Cambridge University Press, (1999). 578, 664, 670

[16] Francisco Argüello, Emilio L. Zapata: **Fast Cosine Transform based on the Successive Doubling Method**, Electronics Letters, vol.26, no.19, pp.1616-1618, (September-1990). Online at `http://citeseer.ist.psu.edu/400246.html`. 501

[17] Jörg Arndt, Christoph Haenel: **Pi – Unleashed**, Springer-Verlag, (2000). (translation of: **Pi. Algorithmen, Computer, Arithmetik**, 1998). 583

[18] Jörg Arndt: **Arctan relations for Pi**, Online at `http://www.jjj.de/arctan/arctanpage.html`. 596

[19] Jörg Arndt: **FXT, a library of algorithms**. Online at `http://www.jjj.de/fxt/`. xi

[20] Jörg Arndt: **hfloat, a library for high precision computations**. Online at `http://www.jjj.de/hfloat/`. xi, 498

[21] Mikhail J. Atallah, Samuel S. Wagstaff, Jr.: **Watermarking With Quadratic Residues**, In: Proc. of IS-T/SPIE Conf. on Security and Watermarking of Multimedia Contents, SPIE vol.3657, pp.283-288, (1999). Online at `http://homes.cerias.purdue.edu/~ssw/water.html`. 751

— B —

[22] John C. Baez: **The Octonions**, Bull. Amer. Math. Soc., vol.39, pp.145-205, (2002). Online at `http://math.ucr.edu/home/baez/octonions/`. 788

[23] David H. Bailey: **FFTs in External or Hierarchical Memory**, Journal of Supercomputing, vol.4, no.1, pp.23-35, (March-1990). Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`. 406

[24] David H. Bailey, P. N. Swarztrauber: **The Fractional Fourier Transform and Applications**, SIAM Review, vol.33, no.3, pp.389-404, (September-1991). Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`. 425

[25] David H. Bailey, Jonathan M. Borwein, Peter B. Borwein, Simon Plouffe: **The Quest for Pi**, Mathematical Intelligencer, vol.19, no.1, pp.50-57, (January-1997). Online at `http://crd.lbl.gov/~dhbailey/dhbpapers/`. 585, 586

[26] David H. Bailey, Richard E. Crandall: **On the Random Character of Fundamental Constant Expansions**, Experimental Mathematics, vol.10, no.2, pp.175-190, (June-2001). Online at `http://www.expmath.org/`. 602

[27] Robert Baillie, S. Wagstaff, Jr.: **Lucas Pseudoprimes**. Mathematics of Computation, vol.35, no.152, pp.1391-1417, (October-1980). 768

[28] Dominique Roelants van Baronaigien: **A Loopless Gray-Code Algorithm for Listing k-ary Trees**, Journal of Algorithms, vol.35, pp.100-107, (2000). 308

[29] F. L. Bauer: **An Infinite Product for Square-Rooting with Cubic Convergence**, The Mathematical Intelligencer, vol.20, pp.12-13, (1998). 653

[30] M. Beeler, R. W. Gosper, R. Schroeppel: **HAKMEM**, MIT AI Memo 239, (29-February-1972). Retyped and converted to html by Henry Baker, April-1995. Online at `http://home.pipeline.com/~hbaker1/`. 57, 62, 154, 155, 641, 692

[31] Albert H. Beiler: **Recreations in the Theory of Numbers**, Dover Publications, (1964). 784, 787

[32] Hacène Belbachir, Farid Bencherif: **Linear Recurrent Sequences and Powers of a Square Matrix**, INTEGERS, vol.6, (2006). Online at `http://www.integers-ejcnt.org/vol6.html`. 637

[33] A. A. Bennett: **The four term Diophantine arccotangent relation**, Annals of Math., vol.27, no.1, pp.21-24, (September-1925). 594

[34] Lennart Berggren, Jonathan Borwein, Peter Borwein, (eds.): **Pi: A Source Book**, Springer, (1997). 583

[35] Bruce C. Berndt, S. Bahrgava, Frank G. Garvan: **Ramanujan's Theories of Elliptic Functions to Alternative Bases**, Trans. Amer. Math. Soc., vol.347, no.11, pp.4163-4244, (1995). Online at `http://www.math.ufl.edu/~frank/publist.html`. 580

[36] Bruce C. Berndt: **Flowers which we cannot yet see growing in Ramanujan's garden of hypergeometric series, elliptic functions and q's**, In: J. Bustoz, M. E. H. Ismail, S. K. Suslov, (eds.): Special Functions 2000: Current Perspective and Future Directions Kluwer, Dordrecht, pp.61-85, (2001). Online at `http://www.math.uiuc.edu/~berndt/publications.html` 580

[37] Daniel J. Bernstein: **Multidigit multiplication for mathematicians**, draft, (August-2001). Online at `http://cr.yp.to/djb.html`. 534

[38] Daniel J. Bernstein: **Pippenger's exponentiation algorithm**, draft, (18-January-2002). Online at `http://cr.yp.to/djb.html`. 539

[39] Daniel J. Bernstein: **Computing Logarithm Intervals with the Arithmetic-Geometric-Mean Iteration**, draft, (2003). Online at `http://cr.yp.to/djb.html`. 598

[40] Pedro Berrizbeitia, T. G. Berry: **Generalized Strong Pseudoprime tests and applications**, Journal of Symbolic Computation, no.11, (1999). 758

[41] James R. Bitner, Gideon Ehrlich, Edward M. Reingold: **Efficient generation of the binary reflected Gray code and its applications**, Communications of the ACM, vol.19, no.9, pp.517-521, (September-1976). 196, 310

[42] Ian F. Blake, Shuhong Gao, Ronald C. Mullin: **Explicit Factorization of $x^{2^k}+1$ over $\mathbb{F}_p$ with Prime $p \equiv 3 \bmod 4$**, Applicable Algebra in Engineering, Communication and Computation 4, pp.89-94, (1993). Online at `http://www.math.clemson.edu/~sgao/pub.html`. 774

[43] Ian F. Blake, Shuhong Gao, Robert J. Lambert: **Construction and Distribution Problems for Irreducible Trinomials over Finite Fields**, In: D. Gollmann, (ed.): Applications of finite fields, pp.19-32, (1996). Online at `http://www.math.clemson.edu/~sgao/pub.html`. 820

[44] Leo I. Bluestein: **A linear filtering approach to the computation of the discrete Fourier transform**, IEEE Transactions on Audio and Electroacoustics, vol.18, pp.451-455, (December-1970). 423

[45] Antonia W. Bluher: **A Swan-like theorem**, Finite Fields and Their Applications, vol.12, pp.128-138, (28-June-2006). 860

[46] Marco Bodrato: **Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0**, Lecture Notes in Computer Science, vol.4547, pp.116-133, (2007). 803, 861

[47] Marco Bodrato, Alberto Zanoni: **What About Toom-Cook Matrices Optimality?**, Technical Report 605, Centro Vito Volterra, Università di Roma Tor Vergata, (October-2006). Online at `http://bodrato.it/papers/WhatAboutToomCookMatricesOptimality.pdf`. 526, 529

[48] Marco Bodrato, Alberto Zanoni: **Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices**, Proceedings of the 2007 international symposium on Symbolic and algebraic computation, pp.17-24, (2007). 527, 529

[49] J. M. Borwein, P. B. Borwein: **Cubic and higher order algorithms for π**, Canad. Math. Bull., vol.27, pp.436-443, (1984). 587

[50] J. M. Borwein, P. B. Borwein: **More quadratically converging Algorithms for π**, Mathematics of Computation, vol.46, no.173, pp.247-253, (January-1986). 583

[51] J. M. Borwein, P. B. Borwein: **An explicit cubic iteration for π**, BIT Numerical Mathematics, vol.26, no.1, (March-1986). 588

[52] J. M. Borwein, P. B. Borwein: **Pi and the AGM**, Wiley, (1987). 576, 583, 584, 585, 589, 597, 670

[53] J. M. Borwein, P. B. Borwein: **On the Mean Iteration** $(a,b) \leftarrow \left(\frac{a+3b}{4}, \frac{\sqrt{ab}+b}{2}\right)$, Mathematics of Computation, vol.53, no.187, pp.311-326, (July-1989). 578, 589

[54] J. M. Borwein, P. B. Borwein: **A cubic counterpart of Jacobi's Identity and the AGM**, Transactions of the American Mathematical Society, vol.323, no.2, pp.691-701, (February-1991). 578, 587

[55] J. Borwein, P. Borwein, F. Garvan: **Hypergeometric Analogues of the Arithmetic-Geometric Mean Iteration**, Constr. Approx., vol.9, no.4, pp.509-523, (1993). Online at `http://www.math.ufl.edu/~frank/publist.html`. 578, 581

[56] J. M. Borwein, P. B. Borwein, F. G. Garvan: **Some cubic modular identities of Ramanujan**, Transactions of the American Mathematical Society, vol.343, no.1, pp.35-47, (May-1994). Online at `http://citeseer.ist.psu.edu/borwein94some.html`. 585

[57] J. M. Borwein, F. G. Garvan: **Approximations to π via the Dedekind eta function**, Organic mathematics (Burnaby, BC, 1995), pp.89-115, CMS Conf. Proc., vol.20, Amer. Math. Soc., Providence, RI, (1997). Online at `http://www.math.ufl.edu/~frank/publist.html`. 587, 588

[58] Florian Braun, Marcel Waldvogel: **Fast Incremental CRC Updates for IP over ATM Networks**, In: Proceedings of the 2001 IEEE Workshop on High Performance Switching and Routing, (2001). Online at `http://citeseer.ist.psu.edu/braun01fast.html`. 80

[59] Richard P. Brent: **Algorithms for minimization without derivatives**, Prentice-Hall, (1973) (out of print). 564

[60] Richard P. Brent: **Fast multiple-precision evaluation of elementary functions**, Journal of the ACM (JACM), vol.23, no.2, pp.242-251, (April-1976). Online at `http://www.rpbrent.com/`. 583

[61] Richard P. Brent: **On computing factors of cyclotomic polynomials**, Mathematics of Computation, vol.61, no.203, pp.131-149, (July-1993). Online at `http://www.rpbrent.com/`. 768, 861

[62] Richard P. Brent: **On the periods of generalized Fibonacci recurrences**, Mathematics of Computation, vol.63, no.207, pp.389-401, (July-1994). Online at `http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`. 639

[63] Richard P. Brent: **Computing Aurifeuillian factors**, in: Computational Algebra and Number Theory, Mathematics and its Applications, vol.325, Kluwer Academic Publishers, Boston, pp.201-212, (1995). Online at `http://www.rpbrent.com/`. 768

[64] Richard P. Brent, Alfred J. van der Poorten, Herman J. J. te Riele: **A comparative Study of Algorithms for Computing Continued Fractions of Algebraic Numbers**, In: Lecture Notes in Computer Science, vol.1122, Springer-Verlag, Berlin, pp.35-47, (1996). Online at `http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`. 682

[65] Richard P. Brent, Samuli Larvala, Paul Zimmermann: **A Fast Algorithm for Testing Irreducibility of Trinomials mod 2 and some new primitive trinomials of degree 3021377**, Mathematics of Computation, vol.72, pp.1443-1452, 2003. Online at `http://www.rpbrent.com/`. 817

[66] Richard P. Brent: **Primality Testing**, Slides of talk to the Oxford University Invariant Society, (25-November-2003). Online at `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/talks.html`. 770

[67] John Brillhart, Derrick H. Lehmer, John L. Selfridge: **New primality criteria and factorizations of** $2^m \pm 1$, Mathematics of Computation, vol.29, no.130, pp.620-647, (April-1975). 762

[68] J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, S. S. Wagstaff, Jr.: **Factorizations of** $b^n \pm 1 \quad b = 2, 3, 5, 6, 10, 11$ **up to high powers**, Contemporary Mathematics, Volume 22, second edition, American Mathematical Society, (1988). Online at `http://www.ams.org/`. 764, 770, 817

[69] Bette Bultena, Frank Ruskey: **An Eades-McKay Algorithm for Well-Formed Parenthesis Strings**, Information Processing Letters 68, pp.255-259, (1998). Online at `http://www.cs.uvic.ca/~ruskey/Publications/`. 306

[70] Ronald Joseph Burthe, Jr.: **Further Investigations with the Strong Probable Prime Test**, Mathematics of Computation, vol.65, no.213, pp.373-381, (January-1996). 757

— C —

[71] K. Cattel, S. Zhang, X. Sun, M. Serra, J. C. Muzio, D. M. Miller: **One-Dimensional Linear Hybrid Cellular Automata: Their Synthesis, Properties, and Applications in VLSI Testing**, tutorial paper. Online at `http://citeseer.ist.psu.edu/260274.html`. 847

[72] Kevin Cattel, Shujian Zhang: **Minimal Cost One-Dimensional Linear Hybrid Cellular Automata of Degree Through 500**, Journal of Electronic Testing: Theory and Applications, vol.6, pp.255-258, (1995). Online at `http://citeseer.ist.psu.edu/cattell95minimal.html` 843

[73] Kevin Cattell, Frank Ruskey, Joe Sawada, Micaela Serra, C. Robert Miers: **Fast Algorithms to Generate Necklaces, Unlabeled Necklaces, and Irreducible Polynomials over** GF(2), Journal of Algorithms, vol.37, pp.267-282, (2000). Online at `http://www.cis.uoguelph.ca/~sawada/pub.html`. 339, 824

[74] Heng Huat Chan: **On Ramanujan's cubic transformation formula for** ${}_2F_1(\frac{1}{3}, \frac{2}{3}; 1; z)$, Mathematical Proceedings of the Cambridge Philosophical Society, vol.124, pp.193-204, (September-1998). 578

[75] Heng Huat Chan, Kok Seng Chua, Patrick Solé: **Quadratic iterations to** π **associated with elliptic functions to the cubic and septic base**, Trans. Amer. Math. Soc., vol.355, pp.1505-1520, (2003). 578, 588, 589

[76] Busiso P. Chisala: **A Quick Cayley-Hamilton**, The American Mathematical Monthly, vol.105, no.9, pp.842-844, (November-1998). 864

[77] D. V. Chudnovsky, G. V. Chudnovsky: **The computation of classical constants**, Proceedings of the National Academy of Sciences of the United States of America, vol.86, no.21, pp.8178-8182, (1-November-1989). 611

[78] Fan Chun, Persi Diaconis, Ron Graham: **Universal cycles for combinatorial structures**, Discrete Mathematics, vol.11, pp.43-59, (1992). Online at `http://www-stat.stanford.edu/~cgates/PERSI/coauthor.html#Graham`. 839

[79] Jaewook Chung, M. Anwar Hasan: **Asymmetric Squaring Formulae**, Centre for Applied Cryptographic Research, University of Waterloo, Ontario, Canada, (3-August-2006). Online at `http://www.cacr.math.uwaterloo.ca/tech_reports.html`. 526, 527, 528, 532

[80] Frédérick Chyzak, Peter Paule, Otmar Scherzer, Armin Schoisswohl, Burkhard Zimmermann: **The construction of orthonormal wavelets using symbolic methods and a matrix analytical approach for wavelets on the interval**, Experimental Mathematics, vol.10, no.1, pp.67-86, (2001). 519

[81] Mathieu Ciet, Jean-Jacques Quisquater, Francesco Sica: **A Short Note on Irreducible Trinomials in Binary Fields**, (2002). Online at `http://www.dice.ucl.ac.be/crypto/publications/2002/poly.ps`. 819

[82] Douglas W. Clark, Lih-Jyh Weng: **Maximal and Near-Maximal Shift Register Sequences: Efficient Event Counters and Easy Discrete Logarithms**, IEEE Transactions on Computers, vol.43, no.5, pp.560-568, (May-1994). 819

[83] Henri Cohen: **A Course in Computational Algebraic Number Theory**, Springer-Verlag, (1993). Online errata list at `http://www.math.u-bordeaux.fr/~cohen/`. 14, 592, 750, 751, 762, 770, 829, 832, 847

[84] Henri Cohen: **On the equation** $a^m + b^n = c^p$, Note, (January-1998). Online at `http://www.math.u-bordeaux.fr/~cohen/fermatgen`. 786

[85] Henri Cohen, Fernando Rodriguez Villegas, Don Zagier: **Convergence acceleration of alternating series**, Experimental Mathematics, vol.9, no.1, pp.3-12, (2000). Online at `http://www.expmath.org/`. 621

[86] Henri Cohen: **Analysis of the flexible window powering method**, Preprint. Online at `http://www.math.u-bordeaux.fr/~cohen/`. 539

[87] Louis Comtet: **Advanced Combinatorics**, revised edition, D. Reidel Publishing, (1974). 268

[88] Stephen A. Cook: **On the minimum computation time of functions**, PhD thesis, Department of Mathematics, Harvard University, (1966). Online at `http://www.cs.toronto.edu/~sacook/`. 525

[89] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: **Introduction to Algorithms**, MIT Press, second edition, (2001). 115, 149, 524, 553

[90] Richard Crandall, Barry Fagin: **Discrete Weighted Transforms and Large-Integer Arithmetic**, Mathematics of Computation, vol.62, no.205, pp.305-324, (January-1994). 412

[91] Richard E. Crandall, Ernst W. Mayer, Jason S. Papadopoulos: **The twenty-fourth Fermat number is composite**, Mathematicians of Computation, vol.72, no.243, pp.1555-1572, (6-December-2002). 762

[92] Richard Crandall, Carl Pomerance: **Prime Numbers: A Computational Perspective**, second edition, Springer, (2005). 770

[93] R. Creutzburg, M. Tasche: **Parameter Determination for Complex Number-Theoretic Transforms Using Cyclotomic Polynomials**, Mathematics of Computation, vol.52, no.185, pp.189-200, (January-1989). 775

— D —

[94] Ricardo Dahab, Darrel Hankerson, Fei Hu, Men Long, Julio Lopez, Alfred Menezes: **Software Multiplication using Normal Bases**, IEEE Transactions on Computers, vol.55, pp.974-984, (2006). 866

[95] Ivan Damgård, Peter Landrock, Carl Pomerance: **Average case error estimates for the strong probable prime test**, Mathematics of Computation, vol.61, no.203, pp.177-194, (July-1993). 757

[96] Ivan B. Damgård, Gudmund Skovbjerg Frandsen: **An extended quadratic Frobenius primality test with average and worst case error estimates**, IEEE Transactions on Computers, vol.19, pp.783-793, (February-2003). Online at `http://www.brics.dk/RS/03/9/index.html`. 758

[97] Leonard Eugene Dickson: **History of the Theory of Numbers, vol.I, Divisibility and Primality**, Carnegie Institute of Washington, 1919, unaltered reprint of the AMS, (2002). 658

[98] Leonard Eugene Dickson: **History of the Theory of Numbers, vol.II, Diophantine Analysis**, Carnegie Institute of Washington, 1919, unaltered reprint of the AMS, (2002). 315

[99] Geoffrey Dixon: **Division Algebras, Galois Fields, Quadratic Residues**, arXiv:hep-th/9302113, (23-February-1993). Online at `http://arxiv.org/abs/hep-th/9302113`. 790

[100] P. Duhamel, H. Hollmann: **Split radix FFT algorithm**, Electronics Letters, vol.20, no.1, pp.14-16, (5-January-1984). 394

[101] Pierre Duhamel: **Implementation of "split-radix" FFT algorithms for complex, real and real-symmetric data**, IEEE Transactions on Acoustics, Speech and Signal Processing, vol.34 pp.285-295, (1986). 401

[102] Richard Durstenfeld: **Algorithm 235: random permutation**, Communications of the ACM, vol.7, no.7, p.420, (July-1964). 113

[103] Jacques Dutka: **On Square Roots and Their Representations**, Archive for History of Exact Sciences, vol.36, no.1, pp.21-39, (March-1986). 653

— E —

[104] Peter Eades, Brendan McKay: **An algorithm for generating subsets of fixed size with a strong minimal change property**, Information Processing Letters, vol.19, p.131-133, (19-October-1984). 173

[105] H.-D. Ebbinghaus, H. Hermes, F. Hirzebruch, M. Koecher, K. Mainzer, J. Neukirch, A. Prestel, R. Remmert: **Zahlen**, second edition, (english translation: **Numbers**), Springer-Verlag, (1988). 56, 788, 791

[106] Karen Egiazarian, Jaako Astola: **Discrete Orthogonal Transforms Based on Fibonacci-type recursions**, Note, Signal Processing Lab, Tampere University, Finland. 480

[107] Gideon Ehrlich: **Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations**, Journal of the ACM, vol.20, no.3, pp.500-513, (July-1973). 239

[108] Doron Zeilberger ('Shalosh B.EKHAD'): **Forty "Strange" Computer-Discovered Hypergeometric Series Evaluations**, (12-October-2004). Online at `http://www.math.rutgers.edu/~zeilberg/pj.html`. 665

— F —

[109] Steven R. Finch: **Mathematical Constants**, Cambridge University Press, (2003). 317, 670

[110] N. J. Fine: **Infinite Products for k-th Roots**, The American Mathematical Monthly, vol.84, no.8. pp.629-630, (October-1977). 653

[111] K. Fong, D. Hankerson, J. López, A. Menezes: **Field inversion and point halving revisited**, Technical Report, CORR 2003-18, Department of Combinatorics and Optimization, University of Waterloo, Canada, (2003). Online at `http://citeseer.ist.psu.edu/fong03field.html`. 854

[112] The Free Software Foundation (FSF): **GCC, the GNU Compiler Collection**. Online at `http://gcc.gnu.org/`. 21

[113] The Free Software Foundation (FSF): **Other built-in functions provided by GCC**, section 5.44 of the documentation for GCC version 3.4.4. Online at `http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/Other-Builtins.html`. 21

— G —

[114] Yves Gallot: **Cyclotomic polynomials and prime numbers**, Note, revised version, (5-January-2001). Online at `http://perso.wanadoo.fr/yves.gallot/papers/cyclotomic.html` 768

[115] Shuhong Gao: **Normal Bases over Finite Fields**, Thesis, University of Waterloo, Ontario, Canada, (1993). Online at `http://www.math.clemson.edu/~sgao/pub.html`. 774

[116] Shuhong Gao, Daniel Panario: **Tests and Constructions of Irreducible polynomials over Finite Fields**, In: F. Cucker, M. Shub, (eds.): Foundations of Computational Mathematics, Springer, pp.346-361, (1997). Online at `http://www.math.clemson.edu/~sgao/pub.html`. 811, 812

[117] Frank Garvan: **Cubic modular identities of Ramanujan, hypergeometric functions and analogues of the arithmetic-geometric mean iteration**, Contemporary Mathematics, vol.166, pp.245-264, (1993). Online at `http://www.math.ufl.edu/~frank/publist.html`. 578

[118] Frank Garvan: **Ramanujan's theories of elliptic functions to alternative bases – a symbolic excursion**, J. Symbolic Computation, vol.20, no.5-6, pp.517-536, (1995). Revised 16-December-2005 edition online at `http://www.math.ufl.edu/~frank/publist.html`. 581

[119] Joachim von zur Gathen, Jürgen Gerhard: **Modern Computer Algebra**, Cambridge University Press, second edition, 2003. 656, 770, 832

[120] Solomon W. Golomb: **Cyclotomic Polynomials and Factorization Theorems**, The American Mathematical Monthly, vol.85, no.9, pp.734-737, (November-1978). 768

[121] Daniel M. Gordon: **A survey of fast exponentiation methods**, Journal of Algorithms, vol.27, no.1, pp.129-146, (1998). Online at `http://citeseer.ist.psu.edu/gordon97survey.html`. 539

[122] Mark Goresky, Andrew Klapper: **Fibonacci and Galois Representations of Feedback with Carry Shift Registers**, IEEE Transactions on Information Theory, vol.48, no.11, pp.2826-2836, (November-2002). Online at `http://www.math.ias.edu/~goresky/`. 842

[123] Johannes Grabmeier, Alfred Scheerhorn: **Finite Fields in AXIOM**, AXIOM Technical Report no. ATR/5, (1992). Online at `http://citeseer.ist.psu.edu/121340.html`. 869

[124] R. L. Graham, D. E. Knuth, O. Patashnik: **Concrete Mathematics**, second printing, Addison-Wesley, New York, (1988). 267, 268, 306, 658, 664, 667, 689

[125] Torbjörn Granlund, Peter L. Montgomery: **Division by Invariant Integers using Multiplication**, SIGPLAN Notices, vol.29, pp.61-72, (June-1994). Online at `http://swox.com/~tege/`. 5

[126] Torbjörn Granlund: **Instruction latencies and throughput for AMD and Intel x86 processors**, (6-September-2005). Online at `http://swox.com/doc/x86-timing.pdf`. 883

[127] Andrew Granville: **It is easy to determine whether a given integer is prime**, Bulletin (New Series) of the AMS, vol.42, no.1, pp.3-38, (2004). Online at `http://www.dms.umontreal.ca/~andrew/AllPublic.html`. 770

[128] S. Gudvangen: **Practical Applications of Number Theoretic Transforms**, NORSIG-99, Norwegian Signal Processing Symposium, (1999). Online at `http://www.norsig.no/norsig99/`. 514

— H —

[129] Bruno Haible, Thomas Papanikolaou: **Fast multiprecision evaluation of series of rational numbers**, Technical Report no. TI-7/97, (18-March-1997). Online at `http://www.informatik.th-darmstadt.de/TI/Veroeffentlichung/TR/Welcome.html`. 615

[130] R. V. L. Hartley: **A More Symmetrical Fourier Analysis Applied to Transmission Problems**, Proceedings of the IRE, vol.30, pp.144-150, (March-1942). 483

[131] B. R. Heap: **Permutations by Interchanges**, Computer Journal, vol.6, pp.293-294, (1963). 234

[132] Gerben J. Hekstra, Ed F. A. Deprettere: **Floating Point Cordic (extended version)**, Technical Report ET/NT 93.15, Delft University of Technology, (8-March-1993). Online at `http://citeseer.ist.psu.edu/hekstra93floating.html`. 634

[133] Nicholas J. Higham: **The Matrix Sign Decomposition and its Relation to the Polar Decomposition**, Linear Algebra and its Applications, 212-213, pp.3-20, (1994). Online at `http://citeseer.ist.psu.edu/higham94matrix.html`. 553

[134] Nicholas J. Higham: **Stable Iterations for the Matrix Square Root**, Numerical Algorithms, vol.15, no.2, pp.227-242, (1997). Online at `http://www.ma.man.ac.uk/~higham/`. 553

[135] Christian W. Hoffmann: π **und das arithmetisch-geometrische Mittel**, Swiss Federal Research Institute WSL, (9-April-2002). Online at `http://www.wsl.ch/staff/christian.hoffmann/pi.pdf`. 588

[136] Alston S. Householder: **Polynomial Iterations to Roots of Algebraic Equations**, Proceedings of the American Mathematical Society, vol.2, no.5, pp.718-719, (October-1951). 561

[137] Alston S. Householder: **The Numerical Treatment of a Single Nonlinear Equation**, McGraw-Hill, (1970). 564, 566, 568, 572

[138] Thomas D. Howell, Jean-Claude Lafon: **The Complexity of the Quaternion Product**, Technical Report TR-75-245, Department of Computer Science, Cornell University, Ithaca, NY, (June-1975). Online at `http://home.pipeline.com/~hbaker1/`. 791

— **I** —

[139] F. M. Ives: **Permutation enumeration: four new permutation algorithms**, Communications of the ACM, vol.19, no.2, pp.68-72, (February-1976). 252

— **J** —

[140] Gerhard Jaeschke: **On strong pseudoprimes to several bases**, Mathematics of Computation, vol.61, no.204, pp.915-926, (October-1993). 756

[141] T. A. Jenkyns: **Loopless Gray Code Algorithms**, Technical Report CS-95-03, Brock University, Canada, (July-1995). Online at `http://citeseer.ist.psu.edu/134169.html`. 196, 204

[142] S. M. Johnson: **Generation of permutations by adjacent transposition**, Mathematics of Computation, vol.17, no.83, pp.282-285, (July-1963). 239

[143] Dieter Jungnickel, Alfred J. Menezes, Scott A. Vanstone: **On the Number of Self-Dual Bases of $GF(q^m)$ Over $GF(q)$**, Proceedings of the American Mathematical Society, vol.109, no.1, pp.23-29, (May-1990). 873

— **K** —

[144] Bahman Kalantari, Jürgen Gerlach: **Newton's Method and Generation of a Determinantal Family of Iteration Functions**, Technical Report DCS-TR 371, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, (1998). Online at `http://citeseer.ist.psu.edu/kalantari98newtons.html`. 567

[145] Dan Kalman: **A Singularly Valuable Decomposition: The SVD of a Matrix**, preprint, College Math Journal, vol.27, no.1, (January-1996). Online at `http://www.american.edu/academic.depts/cas/mathstat/People/kalman/pdffiles/index.html`. 555

[146] K. Karamanos: **From symbolic dynamics to a digital approach**, International Journal of Bifurcation and Chaos, vol.11, no.6, pp.1683-1694, (2001). 700

[147] Anatoly A. Karatsuba, Y. Ofman: **Multiplication of multidigit numbers on automata**, Soviet Physics Doklady, vol.7, no.7, pp.595-596, January-1963. Translated from Doklady Akademii Nauk SSSR, vol.145, no.2, pp.293-294, (July-1962). 524

[148] Richard Kaye: **A Gray Code For Set Partitions**, Information Processing Letters, vol.5, no.6, pp.171-173, (December-1976). Online at `http://www.kaye.to/rick/`. 324

[149] Wilfried Keller: **Fermat factoring status**. Online at `http://www.prothsearch.net/fermat.html`. 763

[150] Louis V. King: **On the Direct Numerical Calculation of Elliptic Functions and Integrals**, Cambridge University Press, (1924). 583, 603

[151] V. Kislenkov, V. Mitrofanov, E. Zima: **How fast can we compute products?**, in ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation, (1999). Online at `http://citeseer.ist.psu.edu/434789.html`. 539

[152] Andrew Klapper, Mark Goresky: **Feedback Shift Registers, 2-Adic Span and Combiners With Memory**, Journal of Cryptology, vol.10, pp.111-147, (1997). Online at `http://www.math.ias.edu/~goresky/EngPubl.html`. 56, 842

[153] Donald E. Knuth: **Structured programming with `go to` statements**, ACM Computing Surveys, vol.6, no.4, (December-1974). 138

[154] Donald E. Knuth: **The Art of Computer Programming**, third edition, Volume 1: Fundamental Algorithms, Addison-Wesley, (1997). Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`. 628, 630

[155] Donald E. Knuth: **The Art of Computer Programming**, third edition, Volume 2: Seminumerical Algorithms, Addison-Wesley, (1997). Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`. 56, 58, 534, 539, 735, 807, 829

[156] Donald E. Knuth: **The Art of Computer Programming**, second edition, Volume 3: Sorting and Searching, Addison-Wesley, (1997). Online errata list at `http://www-cs-staff.stanford.edu/~knuth/`. 115, 659, 739

[157] Donald E. Knuth: **The Art of Computer Programming**, pre-fascicles for Volume 4. Online at `http://www-cs-staff.stanford.edu/~knuth/`. 76, 172, 179, 212, 213, 239, 259, 261, 268, 839

[158] Wolfram Koepf: **Orthogonal Polynomials and Computer Algebra**, In: R. P. Gilbert et al., (eds.): Recent Developments in Complex Analysis and Computer Algebra, Kluwer, pp.205-234, (1999). Online at `http://www.mathematik.uni-kassel.de/~koepf/Publikationen/index.html`. 580

[159] Kenji Koike, Hironori Shiga: **A three terms Arithmetic-Geometric mean**, Journal of Number Theory, vol.124, pp.123-141, (2007). 578

[160] Ramanujachary Kumanduri, Cristina Romero: **Number theory with computer applications**, Prentice-Hall, (1998). 681, 751, 754, 778, 779

— L —

[161] Clement W. H. Lam, Leonard H. Soicher: **Three new combination algorithms with the minimal change property**, Communications of the ACM, vol.25, no.8, pp.555-559, (August-1982). 172

[162] Susan Landau, Neil Immermann: **The Similarities (and Differences) between Polynomials and Integers**, International Conference on Number Theoretic and Algebraic Methods in Computer Science (1993), pp.57-59, version of (8-August-1996). Online at `http://citeseer.ist.psu.edu/landau94similarities.html`. 534

[163] Glen G. Langdon, Jr.: **An algorithm for generating permutations**, Communications of the ACM, vol.10, no.5, pp.298-299, (May-1967). 261

[164] Benoit Leblanc, Evelyne Lutton, Jean-Paul Allouche: **Inverse problems for finite automata: a solution based on Genetic Algorithms**, Lecture Notes in Computer Science vol.1363, pp.157-166, (1998). Online at `http://www.lri.fr/~allouche/bibliorecente.html`. 334

[165] D. H. Lehmer: **On arccotangent relations for π**, American Mathematical Monthly, vol.45, pp.657-664, (1938). 594

[166] Charles E. Leiserson, Harald Prokop, Keith H. Randall: **Using de Bruijn Sequences to Index a 1 in a Computer Word**, MIT Lab for Computer Science, Cambridge, MA, (June-1998). Online at `http://citeseer.ist.psu.edu/leiserson98using.html`. 15

[167] Arjen K. Lenstra: **Integer Factoring**, Designs, Codes and Cryptogaphy, vol.19, pp.101-128, (2000). Online at `http://modular.fas.harvard.edu/edu/Fall2001/124/misc/arjen_lenstra_factoring.pdf`. 770

[168] H. W. Lenstra, Jr., R. J. Schoof: **Primitive Normal Bases for Finite Fields**, Mathematics of Computation, vol.48, no.177, pp.217-231, (January-1987). 869

[169] Rudolf Lidl, Harald Niederreiter: **Introduction to finite fields and their applications**, Cambridge University Press, revised edition, (1994). 817

[170] J. H. van Lint, R. M. Wilson: **A Course in Combinatorics**, Cambridge University Press, (1992). 354, 839

[171] W. Lipski, Jr.: **More on permutation generation methods**, Computing, vol.23, no.4, pp.357-365, (December-1979). 236, 238

[172] Lisa Lorentzen, Haakon Waadeland: **Continued Fractions and Applications**, North-Holland, (1992). 600, 689

[173] Ya Yan Lu: **Computing the Logarithm of a Symmetric Positive Definite Matrix**, Applied Numerical Mathematics: Transactions of IMACS, vol.26, no.4, pp.483-496, (1998). Online at `http://citeseer.ist.psu.edu/331019.html`. 600

— M —

[174] Robert S. Maier: **A generalization of Euler's Hypergeometric Transform**, arXiv:math.CA/0302084, (8-April-2003). Online at `http://arxiv.org/abs/math.CA/0302084`. 667, 669

[175] Robert S. Maier: **Algebraic Hypergeometric Transformations of Modular Origin**, arXiv:math.NT/0501425, (25-July-2005). Online at `http://arxiv.org/abs/math.NT/0501425`. 578, 582

[176] Kei Makita, Yasuyuki Nogami, Tatsuo Sugimura: **makita-primedeg-irred-via-aop.pdf**, Electronics and Communications in Japan (Part III: Fundamental Electronic Science), vol.88, no.7, pp.23-32, (2005). 814

[177] H. Malvar: **Fast computation of the discrete cosine transform through fast Hartley transform**, Electronics Letters, vol.22, no.7, pp.352-353, (1986). 501

[178] Henrique S. Malvar: **Fast Computation of the discrete cosine transform and the discrete Hartley transform**, IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35, no.10, pp.1484-1485, (October-1987). 501

[179] M. D. McIlroy: **A Killer Adversary for Quicksort**, Software Practice and Experience, vol.29, p.1-4, 1999. 117

[180] Z. A. Melzak: **Bypasses: a simple approach to complexity**, Wiley, (1983). 317

[181] Helmut Meyn, Werner Götz: **Self-reciprocal Polynomials Over Finite Fields**, Séminaire Lotharingien de Combinatoire, B21d, 8 pp, (1989). Online at `http://www.emis.de/journals/SLC/`. 815

[182] Peter L. Montgomery: **A Survey of Modern Integer Factorization Algorithms**, CWI Quarterly, vol.7, no.4, pp.337-365, (1994). Online at `http://citeseer.ist.psu.edu/montgomery94survey.html`. 770

[183] James A. Muir, Douglas R. Stinson: **Minimality and Other Properties of the Width-w Nonadjacent Form**, Technical report CORR 2004-08, Centre for Applied Cryptographic Research (CACR) at the University of Waterloo, Canada, (2004). Online at `http://www.cacr.math.uwaterloo.ca/techreports/2004/tech_reports2004.html`. 60

[184] Jean-Michel Muller: **Elementary Functions: algorithms and implementation**, Birkhäuser, (1997). 627, 634

[185] David R. Musser: **Introspective Sorting and Selection Algorithms**, Software – Practice and Experience, vol.8, pp.983-993, (1997). 117

— **N** —

[186] H. J. Nussbaumer: **Fast Fourier Transform and Convolution Algorithms**, second edition, Springer-Verlag, (1982). 418

— **O** —

[187] C. D. Olds: **Continued Fractions**, The Mathematical Association of America, (1963). 689, 779

[188] Michael Orchard: **Fast bit-reversal algorithms based on index representations in** GF(2^n), IEEE International Symposium on Circuits and Systems, (1989), vol.3, pp.1903-1906, (1989). 837

— **P** —

[189] The PARI Group (PARI): **PARI/GP, version 2.3.1**, Online at `http://pari.math.u-bordeaux.fr/`. 817, 887

[190] Michael S. Paterson, Larry J. Stockmeyer: **On the number of nonscalar multiplications necessary to evaluate polynomials**, SIAM J. Comput., vol.2, no.1, pp.60-66, (March-1973). 617

[191] W. H. Payne, F. M. Ives: **Combination Generators**, ACM Transactions on Mathematical Software (TOMS), vol.5, no.2, pp.163-172, (June-1979). 172

[192] Carl Pomerance, J. L. Selfridge, S. Wagstaff, Jr.: **The Pseudoprimes to** $25 \cdot 10^9$. Mathematics of Computation, vol.35, no.151, pp.1003-1026, (July-1980). 756, 768

[193] Peter John Potts: **Computable Real Arithmetic Using Linear Fractional Transformations**, Report, Department of Computing, Imperial College of Science, Technology and Medicine, London, (June-1996). Online at `http://citeseer.ist.psu.edu/potts96computable.html`. 665, 669

[194] Helmut Prodinger: **On Binary Representations of Integers with Digits -1, 0, +1**, INTEGERS, vol.0, (14-June-2000). Online at `http://www.integers-ejcnt.org/vol0.html`. 59

— **Q** —

— **R** —

[195] Michael O. Rabin: **Probabilistic algorithms in finite fields**, Technical Report MIT-LCS-TR-213, Massachusetts Institute of Technology, (January-1979). Online at `http://publications.csail.mit.edu/`. 811

[196] Charles M. Rader: **Discrete Fourier Transforms When the Number of Data Samples is Prime**, Proceedings of the IEEE, vol.56, iss.6, pp.1107-1108, (June-1968). 427

[197] Janusz Rajski, Jerzy Tyszer: **Primitive Polynomials Over GF(2) of Degree up to 660 with Uniformly Distributed Coefficients**, Journal of Electronic Testing: Theory and Applications, vol.19, pp.645-657, (2003). 823

[198] David Rasmussen, Carla D. Savage, Douglas B. West: **Gray code enumeration of families of integer partitions**, Journal of Combinatorial Theory, Series A, vol.70, no.2, pp.201-229, (1995). Online at `http://www.csc.ncsu.edu/faculty/savage/papers.html`. 316

[199] Phillip A. Regalia, Sanjit K. Mitra: **Kronecker Products, Unitary Matrices and Signal Processing Applications**, SIAM Review, vol.31, no.4, pp.586-613, (December-1989). 436

[200] Hans Riesel: **Prime Numbers and Computer Methods for Factorization**, Birkhäuser, (1985). 766, 770

[201] Frank Ruskey, Carla Savage, Terry MinYih Wang: **Generating Necklaces**, Journal of Algorithms, vol.13, pp.414-430, (1992). 342

[202] Frank Ruskey, Joe Sawada: **An Efficient Algorithm for Generating Necklaces with Fixed Density**, SIAM J. Comput., vol.29, no.2, pp.671-684, (1999). Online at `http://www.cs.uvic.ca/~ruskey/Publications/`. 346

[203] Frank Ruskey, Aaron Williams: **Generating combinations by prefix shifts**, Lecture Notes in Computer Science, 3595, (2005). Extended Abstract for COCOON 2005. Online at `http://www.cs.uvic.ca/~ruskey/Publications/`. 169

— S —

[204] Eugene Salamin: **Computation of π Using Arithmetic-Geometric Mean**, Mathematics of Computation, vol.30, no.135, pp.565-570, (July-1976). 583

[205] Eugene Salamin: **Application of Quaternions to Computation with Rotations**, Working Paper, Stanford AI Lab, 1979. Edited and TeX-formatted by Henry G. Baker, (1995). Online at `http://home.pipeline.com/~hbaker1/`. 550

[206] Carla Savage: **A Survey of Combinatorial Gray Codes**, SIAM Review, vol.39, no.4, pp.605-629, (1997). Online at `http://www.csc.ncsu.edu/faculty/savage/papers.html`. 161

[207] Joe Sawada: **Generating Bracelets in Constant Amortized Time**, SIAM J. Comput, vol.31, no.1, pp.259-268, (2001). Online at `http://citeseer.ist.psu.edu/470511.html`. 336

[208] Joe Sawada: **A fast algorithm to generate necklaces with fixed content**, Theoretical Computer Science, vol.301, no.1-3, pp.477-489, (May-2003). 346

[209] Arnold Schönhage: **Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2**, Acta Informatica, vol.7, no.4, pp.395-398, (December-1977). 804, 861

[210] E. Schröder: **On Infinitely Many Algorithms for Solving Equations**, translation by G. W. Stewart of: **Ueber unendlich viele Algorithmen zur Auflösung der Gleichungen**, Mathematische Annalen, (1870). Online at `http://citeseer.ist.psu.edu/93609.html`. 564

[211] Robert Sedgewick: **Permutation Generation Methods**, ACM Computing Surveys (CSUR), vol.9, no.2, pp.137-164, (June-1977). Online at `http://www.princeton.edu/~rblee/ELE572Papers/`. 236

[212] Robert Sedgewick: **Algorithms in C++. Parts 1-4: Fundamentals, Data Structures, Sorting, Searching**, third edition, Addison-Wesley, (1998). 115, 137

[213] Jeffrey Shallit, David Swart: **An Efficient Algorithm for Computing the ith Letter of $\varphi^n(a)$**, In: Proc. 10th Annual SODA conference, pp.768-775, (1999). `http://www.cs.uwaterloo.ca/~shallit/Papers/`. 334

[214] N. J. A. Sloane: **The On-Line Encyclopedia of Integer Sequences**. Online at `http://www.research.att.com/~njas/sequences/`. 11, 40, 47, 58, 60, 66, 67, 68, 69, 75, 86, 130, 131, 234, 267, 268, 269, 270, 286, 288, 290, 291, 293, 295, 296, 298, 306, 308, 316, 318, 320, 327, 329, 339, 343, 344, 354, 372, 658, 660, 677, 691, 692, 693, 694, 695, 696, 697, 698, 700, 704, 707, 709, 710, 718, 719, 720, 721, 722, 723, 742, 745, 751, 753, 756, 768, 780, 785, 790, 819, 821, 822, 824, 842, 850, 868, 875

[215] David M. Smith: **Efficient multiple precision evaluation of elementary functions**, Mathematics of Computation, vol.52, pp.131-134, (1989). 617

[216] Jerome A. Solinas: **Generalized Mersenne Numbers**, Technical report CORR 99-39, University of Waterloo, Canada, (1999). Online at `http://www.cacr.math.uwaterloo.ca/`. 737

[217] Hong-Yeop Song: **Examples and Constructions of Hadamard Matrices**, Dept. of Electical and Electronics Engineering, Yonsei University, Korea, (June-2002). Online at `http://calliope.uwaterloo.ca/~ggong/710T4/Song-lecture.ps`. 354

[218] H. Sorensen, D. Jones, C. Burrus, M. Heideman: **On computing the discrete Hartley transform**, IEEE Trans. on Acoustics, Speech and Signal Processing, vol.33, no.5, pp.1231-1238, (October-1985). 493

[219] Henrik V. Sorensen, Douglas L. Jones, Michael T. Heideman, C. Sidney Burrus: **Real-Valued Fast Fourier Transform Algorithms**, IEEE Transactions on Acoustics, Speech and Signal Processing, vol.35, no.6, pp.849-863, (June-1987). 401

[220] Damien Stehlé, Paul Zimmermann: **A Binary Recursive Gcd Algorithm**, INRIA research report RR-5050, (December-2003). 735

[221] Ralf Stephan: **Divide-and-conquer generating functions. Part I. Elementary sequences**, (2003). Online at `http://arxiv.org/abs/math.CO/0307027`. 705

[222] Carl Størmer: **Sur l'application de la théorie des nombres entiers complexes a la solution en nombres rationnels** $x_1\ x_2\ \ldots x_n\ c_1\ c_2\ \ldots c_n\ k$ **de l'équation:** c_1 **arc tg** $x_1 + c_2$ **arc tg** $x_2 + \ldots + c_n$ **arc tg** $x_n = k\frac{\pi}{4}$, Archiv for Mathematik og Naturvidenskab, B.XIX, Nr.3, (vol.19, no.3), pp.1-96, (1896). 590

[223] Richard G. Swan: **Factorization of polynomials over finite fields**, Pacific J. Math., vol.12, no.3, pp.1099-1106, (1962). 819

[224] Paul N. Swarztrauber: **Bluestein's FFT for Arbitrary N on the Hypercube**, Parallel Computing, vol.17, pp.607-617, (1991). Online at `http://www.cisl.ucar.edu/css/staff/pauls/papers/bluestein/bluestein.html`. 423

— T —

[225] Tadao Takaoka, Stephen Violich: **Combinatorial Generation by Fusing Loopless Algorithms**, In: Computing: The Australasian Theory Symposium (CATS2006), Hobart, Australia. Conferences in Research and Practice in Information Technology (CRPIT), vol.51. Barry Jay, Joachim Gudmundsson, (eds.), (2006). 306

[226] M. A. Thornton, D. M. Miller, R. Drechsler: **Transformations Amongst the Walsh, Haar, Arithmetic and Reed-Muller Spectral Domains**, International Workshop on Applications of the Reed-Muller Expansion in Circuit Design (RMW), pp.215-225, (2001). Online at `http://engr.smu.edu/~mitch/publications.html`. 459

[227] John Todd: **A problem on arc tangent relations**, Amer. Math. Monthly, vol.56, no.8, pp.517-528, (October-1949). 594

[228] Mikko Tommila: **apfloat, A High Performance Arbitrary Precision Arithmetic Package**, (1996). Online at `http://www.apfloat.org/`. 775

[229] J. F. Traub: **Iterative Methods for the Solution of Equations**, Chelsea, (1964). 564, 565, 571

[230] H. F. Trotter: **Algorithm 115: Perm**, Communications of the ACM, vol.5, no.8, pp.434-435, (August-1962). 239

[231] H. W. Turnbull: **Theory of Equations**, fifth edition, Oliver and Boyd, Edinburgh, (1952). 859

— U —

— V —

[232] Vincent Vajnovszki: **Generating a Gray Code for P-Sequences**, Journal of Mathematical Modelling and Algorithms, vol.1, pp.31-41, (2002). 306

[233] Raimundas Vidūnas: **Expressions for values of the gamma function**, arXiv:math.CA/0403510, (30-March-2004). Online at `http://arxiv.org/abs/math/0403510`. 576

[234] San C. Vo: **A Survey of Elliptic Cryptosystems, Part I: Introductory**, NASA Advanced Supercomputing (NAS) Division, (August-2003). Online at `http://www.nas.nasa.gov/News/Techreports/2003/2003.html`. 880

— W —

[235] Timothy Walsh: **Generating Gray codes in $O(1)$ worst-case time per word**, In: DMTCS 2003, C. S. Calude et al. (eds.), Lecture Notes in Computer Science, vol.2731, pp.73-88, (2003). 161

[236] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu: **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD**, Cryptology ePrint Archive, Report 2004/199, revised version, (17-August-2004). Online at `http://eprint.iacr.org/2004/199/`. 79

[237] Zhong-De Wang: **New algorithm for the slant transform**, IEEE Transactions Pattern Anal. Mach. Intell. (PAMI), vol.4, no.5, pp.551-555, (September-1982). 455

[238] André Weimerskirch, Christoph Paar: **Generalizations of the Karatsuba Algorithm for Polynomial Multiplication**, (March-2002). Online at `http://citeseer.ist.psu.edu/571363.html`. 799, 802

[239] André Weimerskirch, Christoph Paar: **Generalizations of the Karatsuba Algorithm for Efficient Implementations**, Technical Report, Ruhr-Universität-Bochum, Germany, corrected version, (2003). Online at `http://www.crypto.ruhr-uni-bochum.de/en_publications.html`. 799

[240] Eric Weisstein: **MathWorld**. Online at `http://mathworld.wolfram.com/`. 665, 720

[241] Mark B. Wells: **Generation of Permutations by Transposition**, Mathematics of Computation, vol.15, no.74, pp.192-195, (April-1961). 238

[242] Mark Weston, Vincent Vajnovszki: **Gray codes for necklaces and Lyndon words of arbitrary base**, Proceedings of GASCOM'06, Dijon, France, pp.55-62, (30-May-2006). 340

[243] Michael Roby Wetherfield, Hwang Chien-lih: **Lists of Machin-type (inverse integral cotangent) identities for Pi/4**, Online at `http://machination.mysite.wanadoo-members.co.uk/index.html`. 594

[244] Michael Roby Wetherfield: **The enhancement of Machin's formula by Todd's process**, The Mathematical Gazette, vol.80, pp.333-344, (July-1996). 594

[245] E. T. Whittaker, G. N. Watson: **A Course of Modern Analysis**, Cambridge University Press, fourth edition, (1927), reprinted 1990. 598, 665, 667, 670

[246] Mladen Victor Wickerhauser: **Adapted Wavelet Analysis from Theory to Software**, AK Peters, Ltd., Wellesley, Mass., (1994). 518

[247] Herbert S. Wilf: **generatingfunctionology**, second edition, Academic Press, (1992). Online at `http://www.math.upenn.edu/~wilf/DownldGF.html`. 269

[248] Hugh C. Williams: **Éduard Lucas and primality testing**, Wiley, (1989). 770

— X —

[249] Limin Xiang, Kazuo Ushijima: **On $O(1)$ Time Algorithms for Combinatorial Generation**, The Computer Journal, vol.44, pp.292-302, (2001). 306

— Y —

— Z —

[250] S. Zaks: **A new algorithm for generation of permutations**, BIT 24, pp.196-204, (1984). 233

[251] Paweł Zieliński, Krystyna Ziętak: **The Polar Decomposition – Properties, Applications and Algorithms**, Annals of the Polish Mathematical Society, vol.38, (1995). Online at `http://citeseer.ist.psu.edu/zielinski95polar.html`. 551

[252] Neal Zierler: **On a Theorem by Gleason and Marsh**, Proceedings of the American Mathematical Society, vol.9, no.2, pp.236-237, (April-1958). 817

Index

Symbols

-2, representations with radix -2 56
9^{9^9}, computation 422
E, elliptic function 576
K, elliptic function 575
$O(1)$ algorithms 524
$\circledast$, cyclic convolution 409
$\circledast_{\{v\}}$, weighted convolution 418
$\circledast_{lin}$, linear convolution 412
η-product (eta-product) 661
π computation, AGM vs. binary splitting 615
π, computation 582
$\varphi(n)$, Euler's totient function 741
i^i, computation 603
z^z, series for 676
$\mathcal{W}_v[\,]$, weighted transform 417
2-adic, inverse and square root 55
2D Hilbert curve 51, 154, 333, 712
3D Hilbert curve 333

A

AC (adjacent changes), Gray code 363
`ac_gray_delta()` 364
acceleration of convergence, sumalt algorithm 621
ACF (auto correlation function) 414, 840
acyclic (linear) convolution 412
acyclic (linear) correlation 414
addition, modulo m 731
additive inverse, modulo m 734
adjacency matrix of a graph 355
adjacent changes (AC), Gray code 363
adjacent nodes in a graph 355
AGM
  – (arithmetic-geometric mean), 573
  – 4-th order variant, 574
  – and hypergeometric functions, 578
  – vs. binary splitting, 615
algebra 787
all-ones polynomial 821, 875
all-ones polynomials, trace vector 860
`all_irredpoly (C++ class)` 825
alternating series, and continued fractions 686
alternating series, sumalt algorithm 621
AND-convolution 463
`apply_permutation()` 104, 112
approximations, initial, for iterations 549
arbitrary length FFT 423
arc (edge) of a digraph 355
arctan relations for π 590
arctan, computation by rectangular scheme 618
argument reduction
  – for arctan, 601
  – for cos, 605
  – for exp, 605
  – for log, 601
arithmetic transform 455
arithmetic transform, convolution property 462
arithmetic, modular 731
arithmetic-geometric mean (AGM) 573
array notation 161
array, of bits 152
asm trick, with GCC 4, 503
asymptotics, of an algorithm 523
auto correlation function (ACF) 414, 840
average, of two integers, without overflow 22

B

backtracking 355
base (radix) conversion 616
base field 770
basis functions, Reed-Muller transform 459
Beatty sequence with Φ 721
Bell numbers 131, 320
Ben-Or test for irreducibility 810
Berlekamp's Q-matrix algorithm 827
Bhaskara equation 778
big endian machine 3
binary
  – exponentiation, 537
  – finite field, 771, 851
  – GCD algorithm, 734
  – heap, 148

– powering, 537
– relation, 127
– search, 117
binary splitting
– for rational series, 611
– vs. AGM, 615
– with continued fractions, 685
`binary_necklace (C++ class)` 338
Binet form, of a recurrence 643
binomial coefficient 165
bit combinations 61
bit counting 19
bit subsets, via sparse counting 63
bit-array 152
bit-array, fitting in a word 21
bit-block boundaries, determination 13
bit-reversal 31
bit-reversal permutation 85
bit-subset, testing 6
bit-wise
– reversal, 30
– rotation, 26
– zip, 35
`bit_fibgray (C++ class)` 73
`bit_necklace (C++ class)` 338
`bit_rotate_sgn()` 49
`bit_subset (C++ class)` 63
`bitarray (C++ class)` 152
`bitpol_factor()` 831
`bitpol_mult()` 84
`bitpol_normal_q()` 866
`bitpol_refine_factors()` 828
`bitpol_sreduce()` 830
bitrev permutation 85
`BITS_PER_LONG` 3
blocks of bits, counting 20
blocks of bits, creation 12
blocks, swapping via quadruple reversion 92
blue code 45, 342
blue code, fixed points 47
bracelets, as equivalence classes 129
branches, avoiding them 22
bsearch 117
`bsearch()` 117
`bsearch_approx()` 118
`bsearch_ge()` 118
bubble sort 24
builtins, GCC 21
butterfly diagram, for radix-2 transforms 431
byte-wise Gray code and parity 38
`BYTES_PER_LONG` 3

C

C++ class XYZ *see* XYZ (C++ class)
C2RFT *see* real FFT
C2RFT (complex to real FT) 398
canonical sequence 104
carries with mixed radix counting 210
carry, in multiplication 533
CAT, constant amortized time 162
`catalan (C++ class)` 301
Catalan constant 622
Catalan numbers 306, 565
Cayley numbers 788
Cayley-Dickson construction 788
characteristic polynomial
– of a matrix, 864
– of a recurrence relation, 635
– with Fourier transform, 504
characteristic, of a field 852
Chase's sequence, for combinations 178
Chebyshev polynomials
– and Pell's equation, 781
– and products for the square root, 653
– and recurrence for subsequences, 641
– and square root approximants, 652
– as hypergeometric functions, 672
– definition, 645
– fast computation, 650
– with accelerated summation, 622
Chinese Remainder Theorem (CRT) 747
Chinese Remainder Theorem, for convolution 514
chirp z-transform 423
circuit in a graph 355
circulant matrix 869
Clausen's products 669
CLHCA (a class of cellular automata) 847
`clz` (Count Leading Zeros), GCC builtin 21
co-lexicographic order
– (definition), 161
– for combinations, 166
– for compositions, 183
– for permutations, 221
– for subsets of a multiset, 275
– with bit combinations, 61
colex (co-lexicographic) order 161
`comb_rec (C++ class)` 179
`combination_chase (C++ class)` 179
`combination_colex (C++ class)` 167
`combination_emk (C++ class)` 174
`combination_enup (C++ class)` 176
`combination_lex (C++ class)` 166
`combination_pref (C++ class)` 170
`combination_revdoor (C++ class)` 172
combinations, Gray code, with binary words 69
combinations, of k bits 61
combinatorial Gray code 161
companion matrix 636, 864

comparison function, for sorting 121
compiler, smarter than you thought 25
complement, of a permutation 105
complement-shift sequences 361
complementary basis 871
complementing the sequency 43
complete graph 357
complex numbers, construction 771
complex numbers, mult. via 3 real mult. 772
complex numbers, sorting 122
composite modulus 741
compositeness of an integer, test for 753
composition, of permutations 105
`composition_colex (C++ class)` 183
`composition_colex2 (C++ class)` 184
`composition_ex_colex (C++ class)` 185
compositions 183
computation of π, AGM vs. binary splitting 615
concave sequence 132
conditional search, for paths in a graph 362
conditional swap 23
conference matrix 349
conjugates of an element in GF(2^n) 857
connected permutation 269
connection polynomial 833
constant
– Catalan, 622
– CORDIC scaling, 631, 634
– Fibonacci parity, 719
– Gray code, 707
– Komornik-Loreti, 694
– parity number, 692
– Pell, 723
– Pell Gray code, 726
– Pell palindromic, 723
– period-doubling, 700
– rabbit, 718
– revbin, 706
– Roth's, 696
– ruler, 699
– sum of Gray code digits, 709
– sum-of-digits, 705
– Thue, 696
– weighted sum of Gray code digits, 711
– weighted sum-of-digits, 706
constant amortized time (CAT) 162
continued fraction 680
continued fractions, as matrix products 685
convergent, of a continued fraction 680
conversion, float to int 7
conversion, of the radix (base) 616
convex sequence 132
convolution
– acyclic (linear), 412
– and Chinese Remainder Theorem, 514
– and multiplication, 532
– AND-convolution, 463
– by FFT, without revbin permutations, 411
– by FHT, 493
– cyclic, 409
– cyclic, by FHT, 493
– dyadic, 445
– exact, 514
– linear, 412
– mass storage, 421
– negacyclic, 418, 496
– OR-convolution, 462
– property, of the Fourier transform, 410
– right-angle, 418
– skew circular, 418
– weighted, 418
– XOR-convolution, 445
cool-lex, order for combinations 169
Cooley-Tukey FFT algorithm 378
`copy_reverse_0()` 397
copying one bit 8
CORDIC algorithms 630
`coroutine (C++ class)` 156
coroutines 156
correlation 414
`cos_rot()` 500
cosine transform (DCT) 500
cosine, by rectangular scheme 620
cosine, CORDIC algorithm 630
cosine, in a finite field 775
counting bits of a sparse word 20
counting bits of a word 19
counting sort 134
coupled iteration, for the square root 543
CPU instructions, often missed 84
CRC (cyclic redundancy check) 77
`crc32 (C++ class)` 79
`crc64 (C++ class)` 77
cross correlation 414
CRT (Chinese Remainder Theorem) 747
`ctz` (Count Trailing Zeros), GCC builtin 21
cube root extraction 543
cubic convergence 563
cycle in a graph 355
cycle type, of a permutation 268
cycle-leaders, for the Gray code permutation 341
cycle-leaders, for the Gray permutation 97
`cycles (C++ class)` 107
cycles of a permutation and in-place routines 111
cycles, of a permutation 106
cyclic
– convolution, 409
– convolution, by FFT, 411

– correlation, 414
– distance, with binary words, 28
– period, of a binary word, 28
– permutation (definition), 108
– permutation, random, 113
– permutations, and factorial numbers, 229
– permutations, recursive generation, 264
– redundancy check (CRC), 77
– ring, 743
– XOR, 29
`cyclic_perm (C++ class)` 265
cyclotomic polynomials
– (definition), 655
– and primes, 768
– and primitive binary polynomials, 826

D

Daubechies wavelets 519
De Bruijn sequence, to compute bit position 15
De Bruijn graph 359
De Bruijn sequence 197, 838
De Bruijn sequence, as path in a graph 359
De Bruijn sequences, number of 839
`debruijn (C++ class)` 197
decimation in frequency (DIF) 381
decimation in frequency FFT algorithm 381
decimation in time (DIT) 378
decimation in time FFT algorithm 378
delta sequence 416
delta set 161
delta squared process 571
`deque (C++ class)` 146
deque (double-ended queue) 146
derangement 105, 270
derangement order, for permutations 260
DFT (discrete Fourier transform) 375, 378
DIF (decimation in frequency) 381
difference sets, and correlation 416
digraph 355
`digraph (C++ class)` 356
`digraph::sort_edges()` 366
`digraph_paths (C++ class)` 357
`digraph_paths::print_turns()` 366
directed graph 355
discrete cosine transform (DCT) 500
discrete Fourier transform (DFT) 375, 378
discrete sine transform (DST) 501
DIT (decimation in time) 378
division
– algorithm using only multiplication, 541
– CORDIC algorithm, 632
– exact, by $C = 2^k \pm 1$, 55
– exact, with polynomials over GF(2), 798
divisionless iterations for polynomial roots 560
`divisors (C++ class)` 275
Dobinski's formula, for Bell numbers 321
double-ended queue (deque) 146
dragon curve sequence 709
DST (discrete sine transform) 501
`dsth()` 501
dual basis 871
dyadic convolution 445
`dyadic_convolution()` 446

E

E, elliptic function 576
Eades-McKay sequence, for combinations 173
easy case, with combinatorial generation 163
edge of a graph 355
edge sorting, with graph search 366
element of order n 507
elementary functions, as hypergeometric f. 671
elliptic E 576
elliptic K 575
elliptic functions, as hypergeometric functions 677
endian-ness, of a computer 3
endo (Even Numbers DOwn) order 175
endo order, for mixed radix numbers 216
enup (Even Numbers UP) order 175
enup order for combinations 176
enup order, with permutations 255
equivalence classes 127
equivalence relation 127
equivalence relations, number of 131
`equivalence_classes()` 128
Eratosthenes, prime sieve 738
eta-product 661
Euclidean algorithm 734
Euler numbers 269
Euler's identity, for hypergeometric functions 667
Euler's totient function 741
exact convolution 514
exact division 55
exact division, by $C = 2^k \pm 1$ 55
exact division, with polynomials over GF(2) 798
exponent, of a group 740
exponential convergence 563
exponential function
– bit-wise computation, 627
– by rectangular scheme, 620
– computation via $q = \exp(-\pi\, K'/K)$, 603
– iteration for, 603
– of power series, 607
exponentiation
– algorithms, 537
– modulo m, 734
extension field 770, 851
external algorithms 421

F

factorial number system 222
factorial, binsplit algorithm for 611
factorization of binary polynomials 827
falling factorial basis 222
fast Fourier transform (FFT) 376
fast Hartley transform (FHT) 483
`fcsr (C++ class)` 840
FCSR (feedback carry shift register) 840
feedback carry shift register (FCSR) 840
Fermat numbers 762
Fermat primes 750
`ffact2cyclic()` 267
`ffs` (Find First Set), GCC builtin 21
FFT
– as polynomial evaluation, 534
– radix-2 DIF, 382
– radix-2 DIT, 380
– radix-4 DIF, 390
– radix-4 DIT, 387
– split-radix algorithm, 392
FFT (fast Fourier transform) 376
FFT caching 539
FFT, for multiplication 532
`fft_complex_convolution()` 412
`fft_dif4l()` 391
`fft_dit4_core_p1()` 388
FHT
– convolution by, 493
– DIF step, 487
– DIF, recursive, 487
– DIT, recursive, 484
– radix-2 DIF, 488
– radix-2 DIT, 485
– radix-2 DIT step, 484
– shift operator, 484
FHT (fast Hartley transform) 483
`fht_dif2()` 488
`fht_dif_core()` 384
`fht_real_complex_fft()` 491
Fibbinary numbers 60, 719
Fibonacci
– k-step sequence, 286
– numbers, 286, 288, 718
– parity, 719
– parity constant, 719
– polynomials, 877
– representation, 719
– setup, of a shift register, 836
– words, 282
– words, Gray code, 73, 283
– words, shifts-order, 200
Fibonacci-Haar transform 478
Fibonacci-Walsh transform 479
FIFO (first-in, first-out), queue 144
filter, for wavelet transforms 516
finite field 771
finite-state machine 154
fixed point, of a function 563
fixed points, of the blue code 47
FKM algorithm 336
four step FFT 406
Fourier shift operator 380
Fourier transform (FT) 375
Fourier transform, convolution property 410
fractional Fourier transform 425
FT (Fourier transform) 375
full path in a graph 355

G

Galois Field 851
Galois setup, of a shift register 836
Gauss' transformation 667
Gaussian normal basis 877
GCC, builtins 21
GCD, computation 734
generator in GF(2^n) 854
generator of a group 740
generator, modulo p 426
generator, program producing programs 502
GF(2^n) (binary finite field) 851
`GF2n (C++ class)` 855, 873
`GF2n::init()` 856, 873
`gf2n_fast_trace()` 853
`gf2n_half_trace()` 863
`gf2n_order()` 854
`gf2n_solve_quadratic()` 861
GNB (Gaussian normal basis) 877
Golay-Rudin-Shapiro sequence 40, 696
Goldschmidt algorithm 555
Gray code
– and radix −2 representations, 57
– binary, reversed, 41
– combinatorial (minimal-change order), 161
– constant, 707
– for bit-subsets, 63
– for combinations, 170
– for combinations of a binary word, 69
– for Fibonacci words, 73, 283
– for Lyndon words, 367
– for mixed radix numbers, 210
– for Pell words, 288, 724
– for sparse signed binary words, 290
– for subsets, with shifts-order, 199
– of a binary word, 36
– permutation, 97
– powers of, 43
– single track, 367

Gray permutation 97
`gray_cycle_leaders (C++ class)` 98
green code 45
grep 139
ground field 770, 851
GRS (Golay-Rudin-Shapiro) sequence 40, 696
`grs_negate()` 437
gslex order, for mixed radix numbers 213

H

`haar()` 466
Haar transform 465
`haar_inplace()` 467
`haar_rev_nn()` 473
Hadamard matrix 347, 790
Hadamard transform 429
half-trace, in GF(2^n) with n odd 863
Halley's formula 565, 567
Hamiltonian cycle 355
Hanoi, towers of, puzzle 701
Hartley shift 484
Hartley transform (HT) 483
hashing, via CRC 77
heap 148
Heap's algorithm for permutations 234
heapsort 134
hexanacci numbers 286, 288
hidden constant, with asymptotics 524
high bits of a word, operations on 16
Hilbert curve
- 3D, 333
- by string substitution, 333
- finite state machine for, 154
- function encoding it, 712
- moves, 51
- turns, 714

homogeneous moves, with combinations 173, 176
homogenous moves, with k-subsets 205
Householder's iteration 566
Householder's method 564
HT (Hartley transform) 483
hyperbolic sine and cosine, by CORDIC 633
hypercomplex numbers 788
hypergeometric function
- (definition), 663
- AGM algorithms, 578
- conversion to continued fraction, 688

I

identical permutation 104
`idsth()` 501
i^i, computation 603
indecomposable permutation 269
index of an element modulo m 740
index of the single set bit 14
index sort 118
infinite products, from series 659
inhomogeneous recurrence 639
initial approximations, for iterations 549
integer partitions 311
integer sequence
- Beatty seq. with Φ, 721
- Carmichael numbers, 753
- Catalan numbers, 306, 565
- Euler function $\varphi(n)$, 742
- Euler numbers, 269
- Feigenbaum symbolic seq., 700
- Fibbinary numbers, 60, 719
- Fibonacci numbers, 286, 288, 718
- fixed points in lex-rev seq., 67
- Golay-Rudin-Shapiro seq., 40, 696
- Gray codes, 707
- GRS (Golay-Rudin-Shapiro) seq., 40, 696
- hexanacci numbers, 286, 288
- hypercomplex multiplication, 790
- indecomposable permutations, 270
- integer partitions, 316
- involutions, 268
- irreducible polynomials, 824
- Kronecker symbols $\left(\frac{-1}{n}\right)$, 710
- Lyndon words, 824
- Mephisto Waltz seq., 695
- Moser – De Bruijn sequence, 58
- necklaces, 343
- non-generous primes, 745
- number of XYZ, *see* number of, XYZ
- numbers both triangular and square, 785
- optimal normal bases, type-1, 875
- optimal normal bases, type-2, 876
- paper-folding seq., 709
- paper-folding seq., signed, 710
- paren words, 75
- partitions into distinct parts, 318
- Pell equation not solvable, 780
- pentanacci numbers, 286, 288
- period-doubling seq., 11, 700
- primes with primitive root 2, 821, 842
- primitive roots of Mersenne primes, 339
- primitive trinomials, 819
- quadratic residues all non-prime, 751
- rabbit seq., 479, 718
- radix -2 representations, 58
- restricted growth strings, 308, 327, 329
- ruler function, 698
- sparse signed binary words, 291
- Stirling numbers of the second kind, 320
- subfactorial numbers, 270
- subset-lex words, 66

– sum of binary digits, 704
– sum of digits of binary Gray code, 709
– swaps with revbin permutation, 86
– tetranacci numbers, 286, 288
– Thue-Morse seq., 39, 432, 691, 790
– tribonacci numbers, 286, 288
– type-1 optimal normal bases, 875
– type-2 optimal normal bases, 876
– values of the Möbius function, 658
– Wieferich primes, 745

integer sequence, by OEIS number
– A000009, 318
– A000010, 742
– A000011, 130
– A000013, 130, 372
– A000029, 130
– A000031, 130, 343
– A000041, 316
– A000045, 286, 288, 290, 293, 295, 718
– A000048, 372
– A000073, 286
– A000078, 286
– A000085, 268
– A000108, 306, 308
– A000110, 131, 320, 329
– A000111, 269
– A000123, 693
– A000129, 722
– A000166, 270
– A000201, 721
– A000213, 288
– A000288, 288
– A000322, 288
– A000383, 288
– A000695, 58
– A001037, 344, 824
– A001045, 291, 293
– A001110, 785
– A001122, 821, 842
– A001220, 745
– A001262, 756
– A001333, 290, 722
– A001511, 698
– A001591, 286
– A001592, 286
– A001764, 308
– A002293, 308
– A002294, 308
– A002475, 819
– A002997, 753
– A003188, 707
– A003319, 270
– A003688, 290
– A003714, 60, 719, 720
– A004211, 329
– A004212, 329
– A004213, 329
– A005351, 58
– A005352, 58
– A005418, 130, 697
– A005578, 291
– A005614, 718
– A005727, 677
– A005811, 709
– A006130, 293
– A006131, 293
– A006206, 660
– A006498, 296
– A006945, 756
– A007895, 719
– A008275, 267
– A008277, 320
– A008683, 658
– A010060, 40, 692
– A011260, 824
– A014577, 709
– A014578, 696
– A015440, 293
– A015441, 293
– A015442, 293
– A015443, 293
– A015448, 290
– A015449, 290
– A019320, 768
– A020229, 756
– A020985, 40, 696
– A022342, 718
– A027362, 868
– A028859, 295
– A031399, 780
– A032908, 720
– A034947, 710
– A035263, 11, 700, 704
– A036991, 75
– A045687, 86
– A046116, 354
– A046699, 69
– A055578, 745
– A055881, 234
– A057460, 819
– A057461, 819
– A057463, 819
– A057474, 819
– A064990, 695
– A065428, 751
– A071642, 875
– A072226, 768
– A072276, 756

– A073639, 819
– A073726, 850
– A079471, 67
– A079559, 68
– A079972, 298
– A080337, 327
– A080764, 722
– A086347, 295
– A093467, 720
– A095076, 719
– A096393, 339
– A100661, 66
– A104521, 723
– A106400, 691, 694
– A107220, 822
– A107222, 868
– A108918, 66
– A118666, 47
– A118685, 790
– A125145, 295
interior bit blocks, determination 13
interleaving process
– for set partitions, 321
– for Trotter's permutations, 239
interpolation binary search 118
interpolation, linear 118
introsort 117
inverse
– 2-adic, 55
– additive, modulo m, 734
– by exponentiation, 853
– cube root, iteration for, 543
– in GF(Q), 853
– iteration for, 541
– modulo m, by exponentiation, 746
– multiplicative, modulo m, 734
– of a circulant matrix, 869
– permutation, 106
– permutation, in-place computation, 108
– power series over GF(2), 798
– root, iteration for, 546
– square root, iteration for, 542
– XYZ transform, *see* XYZ transform
`inverse_haar()` 467
`inverse_haar_inplace()` 467
inversion principle, Möbius 657
invertible modulo m 734
involutions 106, 268
irreducible
– polynomial, 808
– trinomial, 817
`is_cyclic()` 108
`is_quadratic_residue_2ex()` 751
`is_small_prime()` 739
isolation of single bits or zeros 12
iteration
– and multiple roots, 568
– divisionless, for polynomial roots, 560
– for exp, 603
– for inverse, 541
– for inverse cube root, 543
– for inverse root, 546
– for inverse square root, 542
– for logarithm, 599
– for roots, p-adic, 543
– for the zero of a function, 563
– Goldschmidt, 555
– Householder's, 566
– Schröder's, 564
– synthetic, 691
– to compute π, 582
Ives' algorithm for permutation generation 252

J

Jacobi matrix 520

K

K, elliptic function 575
k-subset 200
Karatsuba multiplication
– for integers, 524
– for polynomials, 799
Komornik-Loreti constant 694
`kronecker()` 750
Kronecker product
– (definition), 433
– of Hadamard matrices, 351
`ksubset_gray (C++ class)` 203
`ksubset_rec (C++ class)` 200
`ksubset_twoclose (C++ class)` 205
Kummer's transformation 670

L

Lambert series 658, 703, 739
LCM, least common multiple 735
least common multiple (LCM) 735
left-to-right powering 538
Legendre symbol 749
Legendre's relation 577, 680
Lehmer code, of a permutation 222
lex (lexicographic) order 161
lexicographic order
– (definition), 161
– for bit combinations, 62
– for combinations, 166
– for multiset permutations, 276
– for subsets, 191
– for subsets of a binary word, 64

– generalized, for mixed radix numbers, 213
`lfsr (C++ class)` 834
LFSR (linear feedback shift register) 833
LFSR, and Hadamard matrices 347
LHCA (linear hybrid cellular automaton) 842
`lhca2poly()` 844
`lhca_next()` 842
LIFO (last-in, first-out), stack 141
`lin2hilbert()` 154
linear convolution 412
linear correlation 414
linear feedback shift register (LFSR) 833
Linear hybrid cellular automaton (LHCA) 842
linear interpolation 118
linear, function in a finite field 852
Lipski's Gray codes for permutations 236
list recursions, for Gray codes 281
little endian machine 3
localized Hartley transform algorithm 497
localized Walsh transform algorithm 440
logarithm
– bit-wise computation, 625, 628
– computation by rectangular scheme, 618
– computation via AGM, 597
– computation via $\pi/\log(q)$, 597
– curious series for, 602
– iteration using exp, 599
– of power series, 606
loop in a graph 355
loopless algorithm 162
low bits, operations on 9
Lucas test, for primality 766
Lucas-Lehmer test, for Mersenne numbers 763
lucky path, in a graph 366
Lyndon words
– (definition), 336
– and Mersenne primes, 339
– binary, number of, 343
– number of, 343
– with fixed content, 346
– with fixed density, 344
`lyndon_gray (C++ class)` 370

M

m-sequence 347, 838
MAC (modular adjacent changes), Gray code 363
`make_complement_shift_digraph()` 361
`make_complete_digraph()` 356
`make_oddprime_bitarray()` 739
mass storage convolution 421
`matrix (C++ class)` 435
matrix Fourier algorithm (MFA) 406
matrix square root, applications 550
matrix transposition, and zip permutation 94
matrix transposition, in-place 89
maximal order modulo m 740
`maxorder_element_mod()` 745
mean, arithmetic-geometric 573
median of three elements 116
Mephisto Waltz sequence 695
Mersenne numbers, Lucas-Lehmer test 763
Mersenne primes
– 2^j-th roots, 774
– and Lyndon words, 339
– generalized, 735
Mersenne-Walsh transform 480
MFA (matrix Fourier algorithm) 406
minimal polynomial, in GF(2^n) 857
minimal-change order *see* Gray code
minimum, among bit-wise rotations 27
`minweight_lhca_rule()` 843
missing, CPU instructions 84
mixed radix numbers 207
`mixedradix_endo (C++ class)` 216
`mixedradix_endo_gray (C++ class)` 217
`mixedradix_gray (C++ class)` 210
`mixedradix_gslex (C++ class)` 213
`mixedradix_gslex_alt (C++ class)` 215
`mixedradix_lex (C++ class)` 207
`mixedradix_modular_gray (C++ class)` 213
Möbius function 657
Möbius inversion principle 657
`mod (C++ class)` 509, 775
mod m FFTs 507
modular adjacent changes (MAC), Gray code 363
modular arithmetic 731
modular multiplication 732
modular reduction, with structured primes 735
modular square root 751
modulo, as equivalence classes 128
modulus
– composite, 741
– prime, 741
– prime, with NTTs, 508
moment conditions, for wavelet filters 518
monotone sequence 131
Moser – De Bruijn sequence 58
moves, of the Hilbert curve 51
`mpartition (C++ class)` 315
`mset_lex (C++ class)` 278
`mset_lex_rec (C++ class)` 277
multi-dimensional Walsh transform 432
multigrades 781
multigraph 355
multinomial coefficient 276, 346
multiple roots, iterations for 568
multiplication
– by FFT, 532

– carry, 533
– integer vs. float, 6
– is convolution, 532
– Karatsuba, 524
– modulo m, 732
– of complex numbers via 3 real mult., 772
– of hypercomplex numbers, 788
– of octonions, 790
– of polynomials, 413
– of quaternions, 790
– sum-of-digits test, 536
multiplication matrix, for normal bases 865
multiplication table, of an algebra 787
multiplicative function 657
multiplicative inverse, modulo m 734
multiset 275

N

N-polynomial (normal polynomial) 865
NAF (nonadjacent form) 59
NAF, Gray code 290
`necklace (C++ class)` 197, 337
`necklace2bitpol (C++ class)` 824
necklaces
– as equivalence classes, 129
– binary, 27
– binary, number of, 343
– definition, 335
– with fixed content, 346
– with fixed density, 344
negacyclic convolution 418, 496
neighbors of a node in a graph 355
Newton's formula 859
Newton's iteration, for vector-valued functions 520
node (vertex) of a graph 355
non-generous primes 745
nonadjacent form (NAF) 59
nonadjacent form (NAF), Gray code 290
noncyclic ring 743
normal bases, for GF(2^n) 865
normal basis, optimal 875
normal polynomial 865
`normal_mult()` 866
`normal_solve_reduced_quadratic()` 867
NTT
– (number theoretic transforms), 507
– radix-2 DIF, 510
– radix-2 DIT, 509
– radix-4, 512
number of
– alternating permutations, 269
– aperiodic necklaces, 343
– binary necklaces, 343
– binary partitions of even numbers, 693
– binary reversible strings, 130
– binary words at most r successive ones, 285
– bracelets, 130
– carries, 210
– cycles in De Bruijn graph, 361
– De Bruijn sequences, 839
– derangements, 270
– equivalence relations, 131
– fixed density Lyndon words, 344
– fixed density necklaces, 344
– generators modulo n, 745
– indecomposable permutations, 270
– integer partitions, 316
– integers coprime to n, 741
– invertible circulant matrices, 869
– involutions, 268
– irreducible polynomials, 824
– Lyndon words, 343, 824
– m-sequences, 838
– necklaces, 130, 343
– normal polynomials, 868, 870
– ones in binary Gray code, 709
– parenthesis pairs, 306
– partitions into distinct parts, 318
– permutations of a multiset, 276
– permutations with m cycles, 267
– primitive normal polynomials, 868
– primitive polynomials, 824
– restricted growth strings, 308, 327, 329
– shift register sequences, 838
– sparse signed binary words, 291
– strings with fixed content, 346
– swaps with revbin permutation, 86
– units in GF(Q), 854
– units modulo m, 742
– unlabeled bracelets, 130
– unlabeled necklaces, 130
number theoretic transforms (NTT) 507

O

$O(1)$ algorithm 162
octonions 788
ONB (optimal normal basis) 875
one-point iteration 563
optimal normal basis (ONB) 875
optimization, with combinatorial generation 162
OR-convolution 462
order
– of a polynomial, 809
– of an element modulo m, 739
– of an iteration, 563
out of core algorithms 421

P

p-adic roots, iterations for 543
Padé approximants
– for arctan, 600
– for exp, 604
– for the logarithm, 599
paper-folding sequence 709
`paren (C++ class)` 299
`paren_gray (C++ class)` 306
parentheses, and binary words 74
parity
– number, 692, 704
– of a binary word, 37
– of a permutation, 108
`parity` (parity of a word), GCC builtin 21
Parseval's equation 376
partition
– of a set, 319
– of an integer, 311
`partition (C++ class)` 313
`partition_rec (C++ class)` 311
partitioning, for quicksort 116
Pascal's triangle 166
path in a graph 355
`pcrc64 (C++ class)` 80
Pell
– constant, 723
– equation, 778
– Gray code constant, 726
– palindromic constant, 723
– ruler function, 724
Pell words, Gray code 288, 724
pentanacci numbers 286, 288
pentanomial 820
Pepin's test, for Fermat numbers 762
period of a polynomial 809
period-doubling constant 700
period-doubling sequence 11, 700
`perm_colex (C++ class)` 221
`perm_derange (C++ class)` 260
`perm_gray_ffact (C++ class)` 245
`perm_gray_ffact2 (C++ class)` 244
`perm_gray_lipski (C++ class)` 236
`perm_gray_rfact (C++ class)` 246
`perm_gray_rot1 (C++ class)` 248
`perm_gray_wells (C++ class)` 238
`perm_heap (C++ class)` 234
`perm_heap2 (C++ class)` 235
`perm_heap2_swaps (C++ class)` 236
`perm_ives (C++ class)` 252
`perm_lex (C++ class)` 219
`perm_mv0 (C++ class)` 250
`perm_rec (C++ class)` 263
`perm_restrpref (C++ class)` 268
`perm_rev (C++ class)` 232
`perm_rev2 (C++ class)` 233
`perm_rot (C++ class)` 261
`perm_st (C++ class)` 254
`perm_st_gray (C++ class)` 258
`perm_star (C++ class)` 259
`perm_star_swaps (C++ class)` 259
`perm_trotter (C++ class)` 239
`perm_trotter_lg (C++ class)` 242
permutation
– alternating, 269
– as path in the complete graph, 359
– composition, 105
– connected, 269
– cycle type, 268
– cycles, 106
– cyclic, random, 113
– derangement, 270
– indecomposable, 269
– inverse of, 106
– involution, 106, 268
– of a multiset, 276
– random, 113
– with m cycles, number of, 267
Pfaff's reflection law 667
phi function, number thoeretic 741
π, computation 582
pitfall, shifts in C 5
pitfall, two's complement 5
Pocklington-Lehmer test, for primality 761
pointer sort 120
pointer, size of 3
polar decomposition, of a matrix 552
polynomial
– binary, weight, 809
– irreducible, 808
– multiplication, 413
– multiplication, splitting schemes, 799
– primitive, 809
– roots, divisionless iterations for, 560
`popcount` (bit-count), GCC builtin 21
power series
– computation of exponential function, 607
– computation of logarithm, 606
powering
– algorithms, 537
– modulo m, 734
– of permutations, 110
– of the binary Gray code, 43
Pratt's certificate of primality 759
prefix shifts, order for combinations 169
`prev_lexrev()` 445
prime length FFT, Rader's algorithm 427
primes

– and cyclotomic polynomials, 768
– as modulus, 741
– as modulus, with NTTs, 508
– non-generous, 745
– sieve of Eratosthenes, 738
– structured, 735
– Wieferich, 745
– with primitive root 2, 842
primitive
– n-th root, modulo m, 507
– r-th root of unity, 740
– elements of a group, 740
– pentanomial, 820
– polynomial, 809
– root, 426, 741
– root in GF(2^n), 854
– root of Mersenne primes, 339
– trinomial, 817, 850
`print_cycles()` 107
priority queue 149
`priority_queue (C++ class)` 150
product form
– for a-th root, 558
– for continued fractions, 685
– for elliptic K, 576
– for power series of exp, 607
– for square root, 653
products of k out of n factors 168
products, infinite, from series 659
Proth's theorem 762
Prouhet-Thue-Morse constant 692
pseudo graph 355
pseudo-inverse, of a matrix 553
pseudoprime 753
pseudoprime, strong (SPP) 753
Pythagorean triples 782

Q

Q-matrix 827
quadratic convergence 563
quadratic residue (square) modulo p 749
quadratic residues, and Hadamard matrices 349
quadruple reversion trick 92
quantization 126
`quantize()` 126
quartic convergence 563
quaternions 788
`queue (C++ class)` 144
queue (FIFO) 144
quicksort 116

R

R2CFT *see* real FFT
R2CFT (real to complex FT) 398
rabbit constant 718
rabbit sequence 479, 718
Rabin's test for irreducibility 811
Rabin-Miller test, for compositeness 754
Rader's algorithm, for prime length FFT 427
radix -2 representations 56
radix (base) conversion 616
radix permutation 89
radix sort 136
`radix_sort()` 136
random permutation 113
random selection 113
ranking, with combinatorial objects 162
rational, square root iterations 544
re-orthogonalization, of a matrix 550
real FFT
– by FHT, 491
– split-radix algorithm, 401
– with wrap routines, 399
reciprocal polynomial 813
rectangular scheme
– for arctan and log, 617
– for exp, sin, and cos, 619
recurrence
– (definition), 635
– inhomogeneous, 639
– relation, 635
– relation, for subsequences, 641
red code 45
reduction
– modular, with structured primes, 735
– modulo $x^2 + x + 1$ etc., 772
Reed-Muller transform
– (definition), 459
– and necklaces, 341
– convolution property, 463
relation, binary 127
relex order 161
representations, radix -2 56
restricted growth strings
– (definition), 301
– for k-ary trees, 307
– for parenthesis strings, 301
– for set partitions, 324
revbin
– (bit-wise reversal), 30
– constant, 706
– pairs, via shift registers, 837
– permutation, 85
– permutation, and convolution by FFT, 411
`revbin_permute()` 87
`revbin_permute0()` 88
reversal, of a permutation 105
`reverse_0()` 397

reversed Gray code 41
reversed Gray code permutation 101
reversed zip permutation 95
reversing the bits of a word 30, 31
RGS (restricted growth string) 301
RGS, for set partitions 324
`rgs_binomial (C++ class)` 307
`rgs_fincr (C++ class)` 329
`rgs_maxincr (C++ class)` 325
right-angle convolution 418
right-to-left powering 537
ring buffer 143
ring, cyclic 743
`ringbuffer (C++ class)` 143
rising factorial basis 222
root
– p-adic, iterations for, 543
– extraction, 546
– inverse, iteration for, 546
– of a polynomial, divisionless iterations, 560
– primitive, 741
– primitive, in GF(2^n), 854
– primitive, modulo m, 507
– primitive, of Mersenne primes, 339
`rotate_left()` 91
`rotate_sgn()` 461
rotation, bit-wise 26
rotation, by triple reversion 91
row-column algorithm 405
ruler constant 699
ruler function 196, 698
`ruler_func (C++ class)` 196, 271

S

Sande-Tukey FFT algorithm 381
scalar multiplication 851
Schröder's formula 564
search, binary 117
searching, with unsorted arrays 137
secant method 563
sedenions 788
selection sort 115
self correlation 414
self-dual (basis over GF2^n) 871
self-reciprocal polynomial 814
semi-continuous Fourier transform 377
sentinel element 163
sequence *see* integer sequence
sequency 448
sequency, of a binary word 42
set partition 319
`set_partition (C++ class)` 321
`set_partition_rgs (C++ class)` 324
`setup_q_matrix()` 827
shift operator, for Fourier transform 380
shift operator, for Hartley transform 484
shift register sequence (SRS) 833
shift-and-add algorithms 625
shifts in C, pitfall 5
shifts, and division 4
shifts-order, for subsets 199
sieve of Eratosthenes 738
sign decomposition, of a matrix 553
sign of a permutation 108
sign of the Fourier transform 376
signed binary representation 59
signed binary words, sparse, Gray code 290
simple continued fraction 680
simple path in a graph 355
sine transform (DST) 501
sine, CORDIC algorithm 630
sine, in a finite field 775
single bits or zeros, isolation of 12
single track
– binary Gray code, 367
– order for permutations, 254
– order for subsets, 197
singular value decomposition (SVD) 551
skew circular convolution 418
slant transform 454
slant transform, sequency ordered 455
`slow_convolution()` 409
`slow_ft()` 376
smart, your compiler 25
sorting, edges in a graph 366
sorting, using a heap 134
space-filling Hilbert curve 333
sparse counting, and bit subsets 63
sparse signed binary representation 59
sparse signed binary words, Gray code 290
sparse words, bit counting 20
split-radix FFT algorithm 392
splitting schemes for multiplication
– for integers, 524
– for polynomials over GF(2), 799
splitting, binary, for rational series 611
SPP (strong pseudoprime) 753
`sqrt_modf()` 753
`sqrt_modp()` 751
`sqrt_modpp()` 752
square modulo p 749
square of a permutation 109
square root
– 2-adic, 55
– in GF(2^n), 854
– iteration for, 542
– modulo p, 751
– of a matrix, applications, 550

squarefree factorization, with polynomials 832
SRS (shift register sequence) 833
stable sort 135
`stack (C++ class)` 141
stack (LIFO) 141
stack, for coroutine emulation 156
star-transposition order, for permutations 259
state engine, for coroutine emulation 156
state-engine 154
Stirling numbers
– of the first kind (cycle numbers), 267, 673
– of the second kind (set numbers), 320
strictly convex sequence 133
strictly monotone sequence 132
string substitution engine 331
`string_subst (C++ class)` 332
strings with fixed content 346
strong minimal-change order 161, 239, 306
strong minimal-change order for combinations 173
strong pseudoprime (SPP) 753
structured primes 735
subdegree of a polynomial 821
subfactorial numbers 270
subsequences, recurrence relations for 641
subset of bit-set, testing 6
`subset_debruijn (C++ class)` 197
`subset_deltalex (C++ class)` 192
`subset_gray (C++ class)` 195
`subset_gray_delta (C++ class)` 193
`subset_lex (C++ class)` 192
subsets
– of k bits (combinations), 61
– of a binary word, 63, 64
– of a multiset, 275
subtraction, modulo m 731
sum of Gray code digits constant 709
sum of two squares 777
sum-of-digits constant 705
sum-of-digits test, with multiplication 536
sumalt algorithm 621
`sumdiff()` 401
super-linear iteration 563
SVD (singular value decomposition) 551
Swan's theorem 819
swap without temporary 7
swap, conditional 23
swapping blocks via quadruple reversion 92
swapping two bits 9
symmetries, of revbin permutation 86
synthetic iterations 691

T

`tcrc64 (C++ class)` 79
tensor product 433
tetranacci numbers 286, 288
Thue constant 696
Thue-Morse sequence 39, 432, 691, 790
`thue_morse (C++ class)` 40
TMFA (transposed matrix Fourier algorithm) 406
toggle between two values 6
Toom-Cook algorithm 525
Toom-Cook algorithm for binary polynomials 803
totient function 741
towers of Hanoi 701
trace
– of a polynomial, 865
– of an element in GF(2^n), 853
– vector, fast computation, 859
– vector, in finite field, 853
transformations, for elliptic K and E 677
transformations, of hypergeometric functions 666
transforms, on binary words 45
transition count, for a Gray code 367
`transpose()` 89, 90
transposed matrix Fourier algorithm (TMFA) 406
transposed Reed-Muller transform 460
transposition of a matrix, and zip permutation 94
transposition of a matrix, in-place 89
transposition, with permutation 108
`tree_gray (C++ class)` 308
triangular square numbers 785
tribonacci numbers 286, 288
trinomial
– primitive, 817, 850
– trace vector, 860
triple reversion trick 91
Trotter's algorithm for permutations 239
two's complement, pitfall 5
two-close order for k-subsets 205
two-close order for combinations 176
type-1 optimal normal basis 875
type-2 optimal normal basis 876
type-t Gaussian normal basis 877

U

unique 124
units (invertible elements) 734
unlabeled bracelets 130
unranking, with combinatorial objects 162
unsorted arrays, searching 137
unzip permutation 93
`unzip_rev()` 95

V

vertex, of a graph 355

W

Walsh transform 429

Walsh transform, multi-dimensional 432
`walsh_pal_basefunc()` 448
`walsh_q1()` 453
`walsh_q2()` 453
`walsh_wak_basefunc()` 431
`walsh_wak_dif2()` 431, 459
`walsh_wak_dit2()` 430, 459
`walsh_wal_basefunc()` 448
`walsh_wal_rev()` 450, 451
`walsh_wal_rev_basefunc()` 452
wavelet conditions 516
wavelet filter 516
wavelet transform 515
`wavelet_filter (C++ class)` 517, 518
weight, of binary polynomial 809
weighted convolution 418
weighted sum of Gray code digits constant 711
weighted sum-of-digits constant 706
weighted transform 417
Wells' Gray code for permutations 238
Whipple's identity 668
Wieferich primes 745

X

XOR permutation 96
XOR, cyclic 29
XOR-convolution 445
`xor_permute()` 96
`xrevbin()` 50
x^x, series for 676

Y

yellow code 45, 341

Z

Z-order 53
z-transform 423
Zeckendorf representation 719
zero bytes, finding 54
zero divisors, of an algebra 788
zero padding, for linear convolution 412
zero-one transitions, determination 13
`zip()` 93
zip permutation 93
zip, bit-wise 35
`zip_rev()` 95
z^z, series for 676