Juraj Hromkovič

Design and Analysis of Randomized Algorithms

Introduction to Design Paradigms

Springer

This page intentionally left blank

J. Hromkovič

Design and Analysis of Randomized Algorithms

Introduction to Design Paradigms

With 23 Figures

Springer

Author

Prof. Dr. Juraj Hromkovič
ETH Zentrum, RZ F2
Department of Computer Science
Swiss Federal Institute of Technology
8092 Zürich, Switzerland
juraj.hromkovic@inf.ethz.ch

Series Editors

Prof. Dr. Wilfried Brauer
Institut für Informatik der TUM
Boltzmannstrasse 3
85748 Garching, Germany
Brauer@informatik.tu-muenchen.de

Prof. Dr. Grzegorz Rozenberg
Leiden Institute of Advanced Computer Science
University of Leiden
Niels Bohrweg 1
2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Prof. Dr. Arto Salomaa
Turku Centre for Computer Science
Lemminkäisenkatu 14 A
20520 Turku, Finland
asalomaa@utu.fi

Illustrations
Ingrid Zámečniková

Library of Congress Control Number: 2005926236

ACM Computing Classification (1998): F.2, F.1, G.2

ISBN-10 3-540-23949-9 Springer Berlin Heidelberg New York

ISBN-13 978-3-540-23949-9 Springer Berlin Heidelberg New York

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera ready by author
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Printed on acid-free paper   45/3142/YL - 5 4 3 2 1 0

To my teachers

O. Demáček

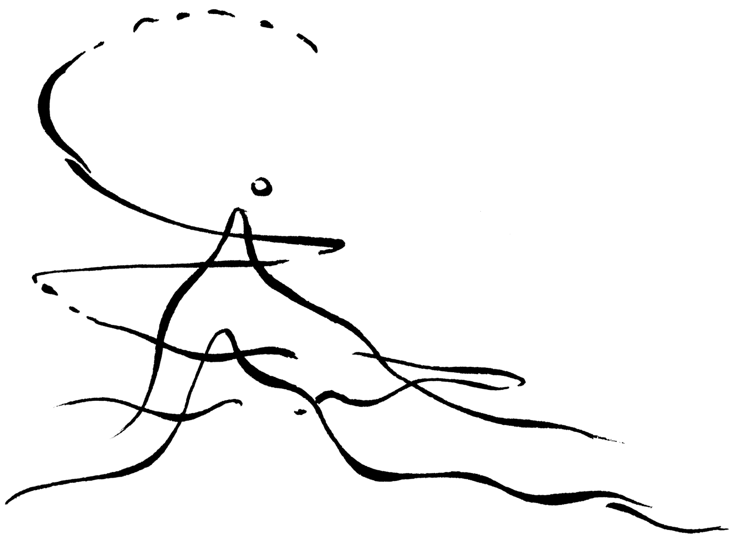
P. Ďuriš

R. Hammerová

B. Rovan

V. Šimčisková

E. Toman



This page intentionally left blank

As soon as man is,

man has to look for

to be.

And as one is looking for

to be,

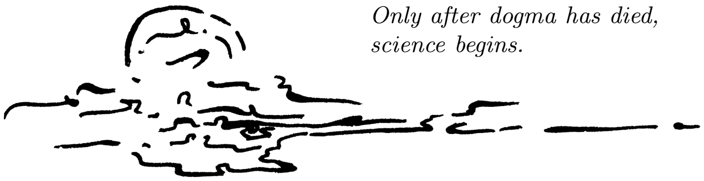
one has to be oneself

and not someone else,

as very often is the case.

Jan Werich

Preface



Only after dogma has died,
science begins.

Galileo Galilei

Randomization has become a standard approach in algorithm design. Efficiency and simplicity are the main features of randomized algorithms that often made randomization a miraculous springboard for solving complex problems in various applications. Especially in the areas of communication, cryptography, data management, and discrete optimization, randomization tends to be an indispensable part of the development of software systems. We know several situations and algorithmic tasks for which any deterministic approach leads to so much computer work that it is impossible to perform it in practice, but for which randomized methods can be successfully applied. This huge gain of going from an intractable amount of computer work[1] (computational complexity) to short computations of randomized algorithms is paid for by the risk of computing a wrong output. But the probability of executing an erroneous computation of a randomized algorithm can often be reduced to below the probability of winning the first prize in a lottery in the first attempt. This means that one pays for saving a huge amount of computing time with a very small loss in the degree of reliability. How to orchestrate such huge quantitative jumps in computational complexity by small relaxations of reliability (correctness) constraints and why such big gains for small "prices" are at all possible are the topics of this book.

Despite many spectacular and surprising results and successful applications of randomized algorithms, the basic concepts of randomization are not sufficiently broadcasted in academic education. One of the main reasons for this unpleasant circumstance is that simple explanations and transparent presentations of the discoveries in randomized computing are missing in textbooks for introductory courses that could be available even to non-scientists and beginners. This is usually the situation when principal contributions and concepts of a particular scientific discipline are not recognized as paradigms in the broad community, and are even considered to be too difficult to be presented in basic courses. The aim of this textbook is to close this gap. We focus

[1]Requiring, for instance, billions of years on the fastest computers.

on a transparent explanation of the paradigms of the design of randomized algorithms. This first book of our series on the design and analysis of randomized algorithms provides a readable introduction to this topic, giving an exhaustive, technical survey of the best and most important results achieved. Providing an accessible ticket to randomization we would like to encourage colleagues not working in the area of randomization to present some randomized concepts in their courses, or even give courses specialized on randomized algorithm design. In this way we would like to contribute to speeding up the inclusion of paradigmatic results on randomized computing in the educational mainstream of computer science.

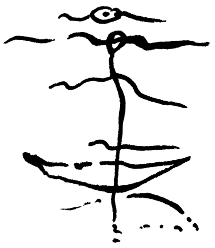
The didactic method of this book is similar to that in our textbook *Theoretical Computer Science.* The main strategies are called “simplicity,” “transparency,” and “less is sometimes more.” Especially in this first book in the series, clarity takes priority over the presentation of the current state of research and development. When a transparent argument of a weaker result can bring across the idea succinctly, then we will opt for it instead of presenting a strong but technically demanding and confusing argument of the best known result. Throughout this book, we work systematically, taking small steps to travel from the simple to the complicated, and so avoid any interruption in thoughts. We are far from trying to present too many results. Instead, we take the time and space to explain what we want to explain in detail, and also in the general context of our scientific discipline. We also dedicate time to the development of informal ideas, and ways of thinking in this area.

I would like to express my deepest thanks to Dirk Bongartz, Hans-Joachim Böckenhauer, and Manuel Wahle for carefully reading the whole manuscript and for their numerous comments and suggestions. Very special thanks go to Bagdat Aslan and Manuel Wahle for their technical help on this manuscript. The excellent cooperation with Alfred Hofmann and Ingeborg Mayer of Springer is gratefully acknowledged. Last but not least I would like to cordially thank Ingrid Zámečniková for her illustrations and Tanja for her collection of citations.

Aachen,
January 2005 *Juraj Hromkovič*

Contents

1 Introduction



To doubt everything or to believe everything –
these are two equally easy solutions,
because both of them relieve us of the necessity of thinking.

Jules Henri Poincaré

1.1 What Is Randomness and Does There Exist True Randomness?

The notion of "randomness" is one of the most fundamental and most discussed terms in science. Following the definition used in dictionaries, an event is considered to be *random* when it happens unpredictably. An object is called *random*, when it is created without any plan. The fundamental question is whether randomness really exists, or whether we use this term only to model objects and events with unknown lawfulness. Philosophers and scientists have disputed the answer to this question since ancient times. Democritos believed that

the randomness is the unknown,
and that the nature is determined
in its fundamentals.

Thus, Democritos asserted that order conquers the world and this order is governed by unambiguous laws. Following Democritos's opinion, one uses the notion of "randomness" only in the subjective sense in order to veil one's inability to truly understand the nature of events and things. Hence the existence of the notion of randomness is only a consequence of the incompleteness of our knowledge. To present his opinion transparently, Democritos liked to use the following example. Two men agreed on sending their slaves to bring water at the same time in order to cause the slaves to meet. The slaves really met at the source and said, "Oh, this is randomness that we have met."

In contrast to Democritos, Epikurus claimed that

the randomness is objective,
it is the proper nature of events.

Thus, Epikurus claimed that there exists a true randomness that is completely independent of our knowledge. Epikurus's opinion was that there exist processes whose development is ambiguous rather than unambiguous, and an unpredictable choice from the existing possibilities is what we call randomness.

One could simply say, "Epikurus was right because there are games of chance, such as rolling dice or roulette, that can have different outcomes, and the results are determined by chance. Unfortunately, the story is not so simple, and discussing gambling one gets the support for the opinion of Democritos rather than for Epikurus's view on the nature of events. Rolling dice is a very complex activity, but if one knows the direction, the speed and the surface on which a die is tossed, then it may be possible to compute and predict the result. Obviously, the movement of the hand controlled by the human brain is too complex to allow us to estimate the values of all important parameters. But we may not consider the process of rolling a die as an objectively random process only because it is too complex for us to predict the outcome. The same is true of roulette and other games of chance. Physics also often uses random models to describe and analyze physical processes that are not inherently or necessarily random (and are sometimes clearly deterministic), but which are too complex to have a realistic possibility of modeling them in a fully deterministic way. It is interesting to note that based on this observation even Albert Einstein accepted the notion of randomness only in relation to an incomplete knowledge, and strongly believed in the existence of clear, deterministic laws for all processes in nature.[1]

Before the 20th century, the world view of people was based on causality and determinism. The reasons for that were, first, religion, which did not accept the existence of randomness in a world created by God[2], and, later, the optimism created by the success of natural sciences and mechanical engineering in the 19th century, which gave people hope that everything could be discovered, and everything discovered could be explained by deterministic causalities of cause and resulting effect.[3]

This belief in determinism also had emotional roots, because people connected randomness (and even identified it) with chaos, uncertainty, and unpredictability, which were always related to fear; and so the possibility of random events was not accepted. To express the strongly negative connotation of randomness in the past, one can consider the following quotation of Marcus Aurelius:

There are only two possibilities,
either a big chaos conquers the world,
or order and law.

[1] "God does not roll dice" is a famous quotation of Albert Einstein. The equally famous reply of Niels Bohr is, "The true God does not allow anybody to prescribe what He has to do."

[2] Today we know that this strong interpretation is wrong and that the existence of true randomness does not contradict the existence of God.

[3] Take away the cause, and the effect must cease.

Because randomness was undesirable, it may not be surprising that philosophers and researchers performed their investigations without allowing the existence of random events in their concepts or even tried to prove the nonexistence of randomness by focusing on deterministic causalities. Randomness was in a similarly poor situation with Galileo Galilei, who claimed that the earth is not a fixed center of the whole universe. Though he was able to prove his claim by experimental observations, he did not have any chance to convince people about it because they were very afraid of such a reality. Life in the medieval world was very hard, and so people clung desperately to the very few assurances they had. And the central position of the earth in the universe supported the belief that the poor man is at the center of God's attention. The terrible fear of losing this assurance was the main reason for the situation, with nobody willing to verify the observations of Galileo Galilei. And the "poor" randomness had the same trouble gaining acceptance[4].

Finally, scientific discoveries in the 20th century (especially in physics and biology) returned the world to Epikurus's view on randomness. The mathematical models of evolutionary biology show that random mutations of DNA have to be considered a crucial instrument of evolution. The essential reason for accepting the existence of randomness was one of the deepest discoveries in the history of science: the theory of quantum mechanics. The mathematical model of the behavior of particles is related to ambiguity, which can be described in terms of random events. All important predictions of this theory were proved experimentally, and so some events in the world of particles are considered as truly random events. For accepting randomness (or random events) it was very important to overcome the restricted interpretation of randomness, identifying it with chaos and uncertainty. A very elegant, modern view on randomness is given by the Hungarian mathematician Alfréd Rényi:

Randomness and order do not contradict each other;
more or less both may be true at once.
The randomness controls the world
and due to this in the world there are order and law,
which can be expressed in measures of random events
that follow the laws of probability theory.

For us, as computer scientists, it is important to realize that there is also another reason to deal with randomness than "only" the modeling of natural processes. Surprisingly, this reason was already formulated 200 years ago by the great German poet Johann Wolfgang von Goethe as follows:

The tissue of the world
is built from necessities and randomness;
the intellect of men places itself between both

[4]One does not like to speak about emotions in the so-called exact (hard) sciences, but this is a denial of the fact that the emotions of researchers (the subjects in the research) are the aggregates of the development and the progress.

and can control them;
it considers the necessity
as the reason of its existence;
it knows how randomness can be
managed, controlled, and used...

In this context Johann Wolfgang von Goethe is the first "computer scientist", who recognized randomness as a useful source for performing some activities. The use of randomness as a resource of an unbelievable, phenomenal efficiency is the topic of this book. We aim to convince the reader that it can be very profitable to design and implement randomized algorithms and systems instead of completely deterministic ones. This realization is nothing other than the acceptance of nature as teacher. It seems to be the case that nature always uses the most efficient and simplest way to achieve its goal, and that randomization of a part of the control is an essential concept of nature's strategy. Computer science practice confirms this point of view. In many everyday applications, simple randomized systems and algorithms do their work efficiently with a high degree of reliability, and we do not know any deterministic algorithms that would do the same with a comparable efficiency. We even know of examples where the design and use of deterministic counterparts of some randomized algorithms is beyond physical limits. This is also the reason why currently one does not relate tractability (practical solvability) with the efficiency of deterministic algorithms, but with efficient randomized algorithms.

To convince the reader of the enormous usefulness of randomization, the next section presents a randomized protocol that solves a concrete communication task within communication complexity that is substantially smaller than the complexity of the best possible deterministic protocol.

We close this section by calling attention to the fact that we did not give a final answer to the question of whether or not true randomness exists, and it is very improbable that science will be able to answer this question in the near future. The reason for this pessimism is that the question about the existence of randomness lies in the very fundamentals of science, i.e., on the level of axioms, and not on the level of results. And, on the level of axioms (basic assumptions), even the exact sciences like mathematics and physics do not have any generally valid assertion, but only assumptions expressed in the form of axioms. The only reason to believe in axioms is that they fully correspond to our experience and knowledge. An example of an axiom of mathematics (viewed as a formal language of science) is that our way of thinking is correct, and so all our formal arguments are reliable. Starting with the axioms, one builds the building of science very carefully, in such a way that all results achieved are true provided the axioms are valid. If an axiom is shown to be not generally valid, one has to revise the entire theory built upon it[5].

Here, we allow ourselves to believe in the existence of randomness, and not only because the experience and knowledge of physics and evolutionary theory support this belief. For us as computer scientists, the main reason to believe in randomness is that randomness can be a source of efficiency. Randomness enables us to reach aims incomparably faster, and it would be very surprising for us if nature left this great possibility unnoticed.

1.2 Randomness as a Source of Efficiency – an Exemplary Application

The aim of this section is to show that randomized algorithms can be essentially more efficient than their deterministic counterparts.

Let us consider the following scenario. We have two computers R_{I} and R_{II} (Figure 1.1) that are very far apart[6]. At the beginning both have a database with the same content. In the meantime the contents of these databases dynamically developed in such a way that one now tries to perform all changes simultaneously in both databases with the aim of getting the same database, with complete information about the database subject (for instance, genome sequences), in both locations. After some time, we want to check whether this process is successful, i.e., whether R_{I} and R_{II} contain the same data.

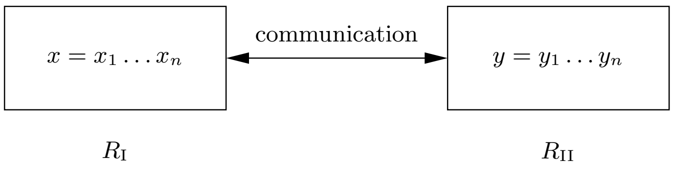


Fig. 1.1.

Let n be the size of the database in bits. For instance, n can be approximately $n = 10^{16}$, which is realistic for biological applications. Our goal is to design a communication protocol between R_{I} and R_{II} that is able to determine whether the data saved on both computers is the same or not. The complexity of the communication protocol is the number of bits that have to

[5]Disproving the general validity of an axiom should not be considered a "tragedy." Such events are part of the development of science and they are often responsible for the greatest discoveries. The results built upon the old, disproved axiom usually need not be rejected; it is sufficient to relativize their validity, because they are true in frameworks where the old axiom is valid.

[6]For instance, one in Europe and one in America.

be exchanged between R_{I} and R_{II} in order to solve this decision problem, and we obviously try to minimize this complexity.

One can prove that every deterministic communication protocol solving this task must exchange at least n bits[7] between R_{I} and R_{II}, i.e., there exists no deterministic protocol that solves this task by communicating $n-1$ or lower bits. Sending $n = 10^{16}$ bits and additionally assuring that all arrive safely[8] at the other side is a practically nontrivial task, so one would probably not do it in this way.

A reasonable solution can be given by the following randomized protocol. Let $x = x_1x_2 \ldots x_n \in \{0,1\}^*$, $x_i \in \{0,1\}$ for all $i = 1, \ldots, n$. We denote by

$$\mathrm{Number}(x) = \sum_{i=1}^{n} 2^{n-i} \cdot x_i$$

the natural number whose binary representation is the string x.

$R = (R_{\mathrm{I}}, R_{\mathrm{II}})$ (Randomized Protocol for Equality)

Initial situation: R_{I} has a sequence x of n bits, $x = x_1 \ldots x_n$, and R_{II} has a sequence y of n bits $y = y_1 \ldots y_n$.

Phase 1: R_{I} chooses uniformly[9] a prime p from the interval $[2, n^2]$ at random.

Phase 2: R_{I} computes the integer

$$s = \mathrm{Number}(x) \mod p$$

and sends the binary representations of s and p to R_{II}.
{Observe that $s \leq p < n^2$ and so each of these integers can be represented by $\lceil \log_2 n^2 \rceil$ bits.}

Phase 3: After reading s and p, R_{II} computes the number

$$q = \mathrm{Number}(y) \mod p.$$

If $q \neq s$, then R_{II} outputs "$x \neq y$".
If $q = s$, then R_{II} outputs "$x = y$".

Now we analyze the work of $R = (R_{\mathrm{I}}, R_{\mathrm{II}})$. First, we look at the complexity measured in the number of communication bits, and then we analyze the reliability (error probability) of the randomized protocol $R = (R_{\mathrm{I}}, R_{\mathrm{II}})$.

The only communication of the protocol involves submitting the binary representations of the positive integers s and p. As we have already observed, $s \leq p < n^2$; hence the length of the message is at most[10]

[7]This means that sending all data of R_{I} to R_{II} for the comparison is an optimal communication strategy.

[8]without flipping a bit

[9]This means that every prime from the interval $[2, n^2]$ has the same probability of being chosen.

[10]Every positive integer m can be represented by $\lceil \log_2(m+1) \rceil$ bits.

$$2 \cdot \lceil \log_2 n^2 \rceil \leq 4 \cdot \lceil \log_2 n \rceil .$$

For $n = 10^{16}$, the binary length of the message is at most $4 \cdot 16 \cdot \lceil \log_2 10 \rceil = 256$. This is a very short message that can be safely transferred.

Now we show not only that for most inputs (initial situations) the randomized strategy works, but also show that the probability of providing the right answer is high for every input. Let us first recognize that the randomized protocol may err[11]. For instance, if $x = 01111$ and $y = 10110$, i.e., $\text{Number}(x) = 15$ and $\text{Number}(y) = 22$, then the choice of the prime 7 from the set $\{2, 3, 5, 7, 11, 13, 17, 19, 23\}$ yields the wrong answer, because

$$15 \bmod 7 = 1 = 22 \bmod 7.$$

To analyze the error probability for any input (x, y), with $x = x_1 \ldots x_n$, and $y = y_1 \ldots y_n$, we partition the set

$$\text{PRIM}\left(n^2\right) = \{p \text{ is a prime} \mid p \leq n^2\}$$

into two subsets (Figure 1.2). The first subset contains the **bad primes**, where a prime p is **bad for (x, y)** if the random choice of p results in the wrong output of the protocol R. The second subset of $\text{PRIM}\left(n^2\right)$ is the complementary subset to the subset of bad primes and we call the primes in this subset **good for (x, y)** because the choice of any of them results in the right answer for the input (x, y).

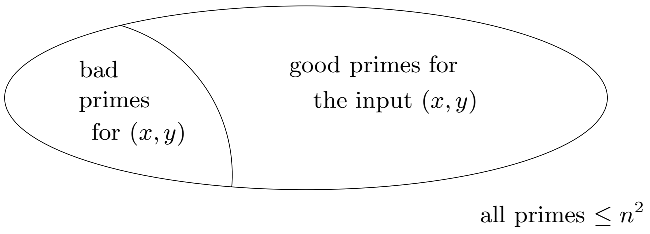


Fig. 1.2.

Since every prime in $\text{PRIM}\left(n^2\right)$ has the same probability of being chosen, the error probability[12] for the input (x, y) is

$$\frac{\text{the number of bad primes for } (x, y)}{\mathit{Prim}\left(n^2\right)},$$

[11]In the sense that the randomized protocol outputs "$x = y$" for different x and y.

[12]Here we work with an informal understanding of the notion of probability. The exact definition of probability and related notions will be presented in the next chapter, and we will then repeat this argument formally.

where $Prim\left(n^2\right)$ denotes the cardinality of $Prim\left(n^2\right)$. The famous Prime Number Theorem says that

$$\lim_{m\to\infty} \frac{Prim\,(m)}{m/\ln m} = 1,$$

and we know that

$$Prim\,(m) > \frac{m}{\ln m}$$

for all positive integers $m > 67$. Hence, we have

$$Prim\left(n^2\right) > \frac{n^2}{2\ln n}$$

for all $n \geq 9$. Our aim is now to show that

for any input (x, y), the number of bad primes for (x, y) is at most $n - 1$,

i.e., that the number of primes that are bad for (x, y) is essentially smaller than $n^2/2\ln n$.

Analyzing the error probability, we distinguish two possibilities with respect to the real relation between x and y.

(i) Let $x = y$.
Then one has

$$\text{Number}(x) \mod p = \text{Number}(y) \mod p$$

for all primes p, i.e., these are no bad primes for the input (x, y). Therefore R_{II} outputs "$x = y$" with certainty, i.e., the error probability is equal to 0 in this case.

(ii) Let $x \neq y$.
One gets the wrong answer "$x = y$" only if R_{I} has chosen a prime p such that

$$s = \text{Number}(x) \mod p = \text{Number}(y) \mod p.$$

In other words, p is a bad prime for (x, y) when

$$\text{Number}(x) = x' \cdot p + s \text{ and } \text{Number}(y) = y' \cdot p + s$$

for some nonnegative integers x' and y'.
A consequence is that

$$\text{Number}(x) - \text{Number}(y) = x' \cdot p - y' \cdot p = (x' - y') \cdot p,$$

i.e., that

$$p \text{ divides the number } |\text{Number}(x) - \text{Number}(y)|.$$

Thus, the protocol R outputs the wrong answer "$x = y$" only if the chosen prime p divides the number $|\text{Number}(x) - \text{Number}(y)|$. This way, we have the following new definition of bad primes:

$$\textit{a prime } p \textit{ is bad for } (x, y) \textit{ iff}$$

$$p \textit{ divides the number } w = |\text{Number}(x) - \text{Number}(y)|.$$

Thus, to estimate the error probability, it is sufficient to estimate how many primes from the $\mathit{Prim}\left(n^2\right) \sim n^2/\ln n^2$ primes divide the number w. Since the length of the binary representations of x and y is equal to n,

$$w = |\text{Number}(x) - \text{Number}(y)| < 2^n.$$

Obviously[13], we can factorize w to get

$$w = p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k},$$

where $p_1 < p_2 < \ldots < p_k$ are primes and $i_1, i_2, \ldots, i_k$ are positive integers. Our aim is to prove that

$$k \le n - 1.$$

We prove it by contradiction. Assume $k \ge n$. Then,

$$w = p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k} \ge p_1 \cdot p_2 \cdot \ldots \cdot p_n > 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n! > 2^n,$$

which contradicts the fact that $w < 2^n$. In this way we have proved that w has at most $n - 1$ different prime factors. Since every prime in $\{2, 3, \ldots, n^2\}$ has the same probability of being chosen, the probability of choosing a bad prime p dividing w is at most

$$\frac{n-1}{\mathit{Prim}\,(n^2)} \le \frac{n-1}{n^2/\ln n^2} \le \frac{\ln n^2}{n}$$

for all $n \ge 9$.
Thus the error probability of R for an input (x, y) with $x \ne y$ is at most

$$\frac{\ln n^2}{n},$$

which is at most

$$0.36892 \cdot 10^{-14}$$

for $n = 10^{16}$.

An error probability of this size is no real risk, but let us assume that a pessimist is not satisfied with this error probability and wants to have an error probability below all physical limits. In such a case one can execute the work of the protocol R ten times, always with an independent, new choice of a prime.

[13]We know from number theory that every positive integer has a unique factorization.

Protocol R_{10}

Initial situation: R_{I} has n bits $x = x_1 \ldots x_n$ and R_{II} has n bits $y = y_1 \ldots y_n$.
Phase 1: R_{I} chooses 10 uniformly random primes

$$p_1, p_2, \ldots, p_{10}$$

from $\{2, 3, \ldots, n^2\}$.
Phase 2: R_{I} computes

$$s_i = \text{Number}(x) \mod p_i$$

for $i = 1, 2, \ldots, 10$ and sends the binary representations of

$$p_1, p_2, \ldots, p_{10}, s_1, s_2, \ldots, s_{10}$$

to R_{II}.
Phase 3: Upon receiving $p_1, p_2, \ldots, p_{10}, s_1, s_2, \ldots, s_{10}$ R_{II} computes

$$q_i = \text{Number}(y) \mod p_i$$

for $i = 1, 2, \ldots, 10$.
If there exists an $i \in \{1, 2, \ldots, 10\}$ such that $q_i \neq s_i$, then R_{II} outputs "$x \neq y$".
Else (if $q_j = s_j$ for all $j \in \{1, 2, \ldots, 10\}$) R_{II} outputs "$x = y$".

We observe that the communication complexity of R_{10} is 10 times larger than that of R. But, for $n = 10^{16}$, the message consists of at most 2560 bits, which is no issue for discussion.

What is the gain with respect to error probability?

If $x = y$, then we again have the situation that the protocol R_{10} provides the right answer "$x = y$" with certainty, i.e., the error probability is equal to 0.

However, if $x \neq y$, R_{10} outputs the wrong answer "$x = y$" only if all 10 chosen primes belong to the maximal $n - 1$ bad primes that divide the difference $w = |\text{Number}(x) - \text{Number}(y)|$. Since the 10 bad primes are chosen in 10 independent experiments, the error probability is at most[14]

$$\left(\frac{n-1}{\mathit{Prim}\,(n^2)}\right)^{10} \leq \left(\frac{\ln n^2}{n}\right)^{10} = \frac{2^{10} \cdot (\ln n)^{10}}{n^{10}}.$$

For $n = 10^{16}$, the error probability is smaller than

$$0.4717 \cdot 10^{-141}.$$

[14]Why the probability of independently choosing two bad primes is equal to the multiplication of the probabilities of choosing a bad prime is carefully explained in Section 2.3.

If one takes into account the fact that the number of microseconds since the Big Bang is a number of 24 digits, and that the number of protons in the known universe is a number of 79 digits, an event with a probability below 10^{-141} is a real wonder. Note also that in the case where a deterministic protocol communication of 10^{16} bits would be executable, the costs speak clearly in favor of the implementation of the above randomized protocol.

We can learn a lot from the construction of the protocol R_{10} that consists of independent repetitions of R. We see that the error probability of a randomized algorithm A can be substantially pushed down by executing several independent runs of A. In cases such as the above communication protocol, even a few repetitions result in an enormous decrease in error probability.

We have observed that using randomization one can gain phenomenally in the efficiency by paying a very small price in reliability. Here we call attention to the fact that in practice randomized algorithms with very small error probability can be even more reliable than their best deterministic counterparts. What do we mean by this? Theoretically, all deterministic algorithms are absolutely correct, and randomized algorithms may err. But the nature of the story is that deterministic programs are not absolutely reliable because during their runs on a computer a hardware error may occur and then they may produce wrong results. Clearly, the probability of the occurrence of a hardware error grows proportionally with the running time of the program. Therefore a fast randomized algorithm can be more reliable than a slow deterministic algorithm. For instance, if a randomized algorithm computes a result in 10 seconds with an error probability 10^{-30}, then it is more reliable than a deterministic algorithm that computes the result in one week. Another good example is our randomized protocol R for Equality. For $n = 10^{16}$, the protocol R has to communicate 256 bits only and the error probability is at most $0.4 \cdot 10^{-14}$. On the other hand, every deterministic protocol has to safely communicate at least 10^{16} bits and the probability of flipping some of them because of a hardware error is essentially larger than the error probability of R.

1.3 Concept of the Book

The aim of this book is to provide an elementary course on the design and analysis of efficient randomized algorithms. We focus not on giving an overview of the deepest contributions to this area, but on a transparent presentation of the most successful design methods and concepts, and we try to contribute to understanding why randomized approaches can be essentially more efficient than their best deterministic counterparts. In this way, we aim to contribute to capturing formal methods as instruments for problem solving and to developing a feeling for the computer scientist's way of thinking.

The only presumed background for reading this textbook is a basic knowledge of introductory courses, such as "programming," "algorithms and data structures" and introduction to the "theory of computation." Thus, we assume that the reader is familiar with terms such as computing task (or problem), decision problem, optimization problem, algorithm, and complexity of algorithms. We use the formal definitions of these basic terms, and the same notation as that presented in our textbook *Theoretical Computer Science* [Hro03]. From mathematics, we assume some elementary knowledge of combinatorics and linear algebra. All other concepts and assertions of probability theory, algebra, and number theory are either presented whenever they are needed or surveyed in the Appendix.

The book is divided into eight chapters, including this introduction. In order to support the iterative way of teaching, these chapters are organized as follows. Every chapter opens with a section "Objectives," in which the motivations, teaching objectives, and relations to topics of the previous chapters are presented. The core of the chapter is dedicated to the formalization, development, and application of the ideas presented in the "Objectives." For every essential development and achievement, we will pinpoint its relevance to our objectives. We end each chapter with a short summary and outlook. Here the major highlights of the chapter are informally summarized, and the chapter's relevance to other parts of the book is once again reviewed.

Chapter 2 provides the fundamentals. One learns here what randomized algorithms are, and how to design and analyze them. The core of Chapter 2 begins with Section 2.2, with elementary fundamentals of probability theory, reduced to a simple kernel that is sufficient for our purposes. In Section 2.4 we explain what randomized algorithms are and how to model and analyze them by means of probability theory. Section 2.5 presents the fundamental classification of randomized algorithms with respect to their error probabilities.[15] Section 2.5 shows how to model and classify randomized algorithms in the areas of discrete optimization, where we usually do not speak about error probability but about a probability of getting a good approximation of an optimal solution. From a contextual point of view Section 2.5 is central to this textbook. Here we introduce the most successful and recognized paradigms of the design of randomized algorithms such as "Fooling an Adversary," "Fingerprinting," "Amplification," "Random Sampling," "Abundance of Witnesses," and "Random Rounding." In this way, we start not only to build the methodology and the machinery for the design of efficient and simple randomized algorithms, but also to capture the nature of the fascinating computational power of randomization in many applications. The paradigms introduced here determine the structure of this book because each of the following chapters (apart from the Appendix) is devoted to the study of one of these paradigms.

Chapter 3 provides a deeper insight into the application of the method of fooling an adversary, which is also called the method of eliminating worst-case

[15]More precisely, with respect to the speedup of reducing the error probability with the number of independently executed runs of the randomized algorithm on the same input.

problem instances. Here, one views a randomized algorithm as a probability distribution over a set of deterministic algorithms (strategies). The crucial point is creating a set of deterministic strategies such that, for any problem instance, most of these strategies efficiently compute the correct result[16]. This can be possible even when there does not exist any efficient deterministic algorithm for solving the problem[17] considered. First, we make this approach transparent by presenting hashing, where universal hashing is nothing other than an application of the method of fooling an adversary. A deeper insight into the power of this method is given by applying it in the area of online algorithms.

The fingerprinting method is successfully applied several times in Chapter 4. The idea of this method is to solve equivalence problems in such a way that instead of trying to compare full complex representations of given objects one compares rather their randomly chosen partial representations called fingerprints. The design of our randomized protocol presented in Section 2.2 can also be viewed as an application of this design paradigm. In Section 4.2 we apply fingerprinting in order to solve other communication problems that can be viewed as generalizations of the equality problem. Section 4.3 uses our motivation example once again in order to design an efficient randomized algorithm for searching for a string (pattern) in a longer string (text). Section 4.4 shows how one can apply fingerprinting in order to verify the correctness of the multiplication of two matrices in a more efficient way than the matrix multiplication.[18] In Section 4.5 we generalize the idea of Section 4.4 in order to develop a polynomial randomized algorithm for deciding the equivalence of two polynomials. This application of fingerprinting is of special importance, because a deterministic polynomial algorithm for this decision problem is not known.

Because amplification and random sampling are often combined, or even indistinguishably mixed, we present them together in Chapter 5. The paradigm of success probability amplification is common to all randomized algorithms and it says that one can increase the success probability of any randomized algorithm by several independent runs of the algorithm on the same input. Section 5.2 shows a more clever application of this paradigm by repeating only some critical parts of a computation instead of repeating all the random runs. Random sampling enables us to create objects with some required properties by a simple random choice from a set of objects, despite the fact that one does not know how to efficiently construct such objects in the deterministic way. In Section 5.3 we combine amplification with random sampling in order to successfully attack the NP-complete satisfiability problem. Section

[16]This means that some of these strategies are allowed to compute wrong results on some inputs.

[17]I.e., a deterministic algorithm that is correct and efficient on any input.

[18]I.e., one can verify whether $A \cdot B = C$ for three matrices A, B, and C without computing $A \cdot B$.

5.4 shows an application of random sampling for efficiently generating non-quadratic residua, which one does not know how to generate deterministically in polynomial time.

Chapter 6 is devoted to the method of abundance of witnesses, which can be viewed as the deepest paradigm of randomization. A witness is additional information to an input, whose knowledge makes a hard problem efficiently solvable. The idea of this method is to generate such witnesses at random. The art of successfully applying this method lies in searching for a suitable kind of witness for the given problem. Here we present a part of such a search for a convenient definition of witnesses for primality testing, which results in the design of efficient randomized primality testing algorithms.

Chapter 7 is devoted to the design of randomized approximation algorithms for the NP-hard maximum satisfiability problem (MAX-SAT). We show how one can round a real solution of the relaxed version of MAX-SAT at random in such a way that a good approximation of an optimal solution to the original discrete optimization problem can be expected.

Appendix A provides some fundamentals of mathematics sufficient for the purposes of the previous chapters. The mathematics is viewed here as a formal language and as a machinery (sets of instruments and methods) for designing, modeling and analyzing randomized algorithms, and it is also presented in this way. Section A.2 provides a short, concise introduction to fundamentals of group theory and number theory. Section A.3 presents some basic facts and methods of combinatorics.

1.4 To the Student

This textbook has been written primarily for you. The aim of this book is not only to introduce and explain some basic methods for the design of efficient randomized algorithms, but also to inspire you for the study of theoretical computer science. In the previous sections of this chapter we have attempted to convince you that randomization is a fascinating area of computer science, because due to randomization one can efficiently perform things that were not considered possible before, and so one can enjoy work on a topic that offers a lot of exciting surprises.

But to teach an exciting topic is not sufficient to fill the lecture room with many interested students. A good didactic presentation of the topic for the success of a course is at least as important as the attractiveness of the subject. Therefore, our presentation of this topic is based on the following three concepts:

(i) *Simplicity and transparency*
We explain simple notions, concepts, and methods in simple terms. We avoid the use of unnecessary mathematical abstractions by attempting to be as concrete as possible. Through this we develop an introduction to the

design of randomized algorithms on elementary mathematical knowledge. Presenting complicated arguments or proofs, we first explain the ideas in a simple and transparent way, and then provide the formal, detailed proofs. Sections and theorems marked with a "$*$" are more involved and technical. Undergraduates are advised to skip these parts when reading the material for the first time.

Clarity takes priority over the presentation of the best known results. When a transparent argument of a weaker result can bring across the idea succinctly, we opt for it instead of presenting a strong but technically demanding and confusing argument of the best known result.

Throughout this book, we work systematically, taking small steps to journey from the simple to the complicated. We avoid any interruption in thoughts.

(ii) *Less is sometimes more, or a context-sensitive presentation*
Many study guides and textbooks falsely assume that the first and foremost aim is the delivery of a quantum of information to the reader. Hence, they often go down the wrong track: maximum knowledge in minimum time, presented in minimal space. This haste usually results in the presentation of a great amount of individual results, thus neglecting the context of the entire course. The philosophy behind this book is different. We would like to build and influence the student's way of thinking. Hence, we are not overly concerned about the amount of information, and are prepared to sacrifice 10% to 20% of the teaching material. In return we dedicate more time to the informal ideas, motivations, connections between practice and theory, and, especially, to internal contexts of the presented research area. We place special emphasis on the creation of new terms. The notions and definitions do not appear out of the blue, as seemingly so in some lectures using the formal language of mathematics. The formally defined terms are always an approximation or an abstraction of intuitive ideas. The formalization of these ideas enables us to make accurate statements and conclusions about certain objects and events. They also allow for formal and direct argumentation. We strive to explain our choice of the formalization of terms and models used, and to point out the limitations of their usage. To learn to work on the level of terms creation (basic definitions) is very important, because most of the essential progress happens exactly at this level.

(iii) *Support of iterative teaching*
The strategy of this book is also tailored to cultivate repetitive reconsideration of presented concepts. As already mentioned, every chapter opens with a section "Objectives" in which the objectives are presented in an informal way and in the context of knowledge from the previous chapters. Every essential development in the main body of a chapter is accomplished with a discussion about its importance in the context of already presented knowledge. The conclusion of each chapter informally summarizes its major highlights and weighs its contribution on a contex-

tual level. As usual, the learning process is supported by exercises. The exercises are not allocated to special subsections, but are distributed in the text, with our recommendation to work through them immediately after you have encountered them while reading the book. They help learn how to successfully apply presented concepts and methods and to deepen your understanding of the material.
But the most important point is that this textbook is self-contained, with all formal and informal details presented in the lectures, and so one can use it for a complete review of all explanations given in the teacher's lecture. In fact, one can master the subject of this book by only reading it (without attending the lecture).

1.5 To the Teacher

The aim of this textbook is to support you in creating an introductory course on randomized algorithms. The advantage of this book is that it provides a lot of space for informal development of concepts and ideas, which unfortunately are often presented only orally in lectures, and are not included in the written supporting materials. Therefore, if the teacher followed this book in her or his lecture, the student would have the possibility to review the complete lecture, or a part of it as many times as she or he wanted. Additionally, the students are not required to write all technical details presented during the lecture, and can concentrate on the explanations given by the teacher.

Finally, we allow ourselves to formulate four rules which can be very helpful in inducting a successful course on any topic. All have been very well known for many years (there is no original idea of ours behind them), but teachers often forget about their consistent application, which is the main problem with education quality.

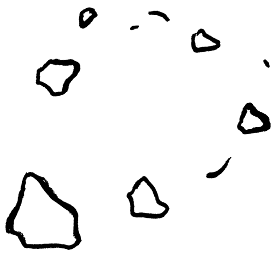
(i) Make sure that your students can review the topic of your lectures any time and as often as they need to. For instance, you can save the entire presentation on the Internet or write (use) detailed supporting materials.
(ii) Provide with your lectures, especially if you have many students in the course, one more public discussion hour per week. In this additional hour students may ask anything related to topics already presented. Typically they ask for more careful repetitions of some complex parts of the lecture or for alternative explanations. Anonymous, written questions should be allowed also.
(iii) Do not save time when one needs to develop concepts and ideas on an informal level or to create new terms. This often underestimated part of the lecture is at least as important as the correct, detailed presentation of results and their proofs. Exactly telling the development of ideas in a scientific discipline in a broad context essentially contributes to a deeper understanding of the subject and motivates the student to deal with the topic.

(iv) Organize small groups for exercises. Take care in choosing exercises for homework in order to fix and deepen the understanding of the actual topic of your lecture. The solutions of the homework have to be made public before meeting the students for exercises in order to prevent the exercises from becoming a presentation of correct solutions. Alternative solutions and the most frequent mistakes have to be discussed. All homework has to be individually corrected and given back to the students.

This page intentionally left blank

2

Fundamentals



At the end of the work
we learn only
with what we should have begun.

Blaise Pascal

2.1 Objectives

The aim of this chapter is to provide the fundamental basis for the design of randomized algorithms.

Section 2.2 begins with an introduction to the fundamental concepts of probability theory, and we define there key terms such as elementary events, events, probability space, random variables, and the expectation of a random variable. Our presentation is restricted to finite probability spaces, because on the one hand they are sufficient for our purposes (for modeling randomized computations considered in this textbook), and on the other hand we avoid the presentation of unnecessary, opaque abstractions of measure theory.

In Section 2.3 we learn how to use probability spaces for modeling randomized algorithms. Special attention is called to the choice of random variables for analyzing the efficiency and the error probability of designed randomized algorithms.

The fundamental classification of randomized algorithms with respect to their error probability is introduced in Section 2.4. Basically, we distinguish between Las Vegas algorithms, that never err (their error probability is equal to 0), and Monte Carlo algorithms that possess a positive error probability. Monte Carlo algorithms are further partitioned into three classes with respect to the speed of reducing their error probability by independently repeating randomized computations of the algorithms on the same input.

Section 2.5 is devoted to the modeling of randomized algorithms for solving discrete optimization problems. Here, one considers a special classification of randomized algorithms that follows the goals of discrete optimization and is based on the concept of approximation algorithms.

The topic of Section 2.6 is central to this textbook. Here, the most important and recognized paradigms and methods for the design of randomized algorithms are presented. In this way we start here our first attempt to provide a deeper insight into the reasons why randomized algorithms can be essentially faster than their best deterministic counterparts.

Finally, Section 2.7 summarizes the most important concepts and ideas presented in this chapter.

2.2 Elementary Probability Theory

If an event is an inevitable consequence of another event, then one speaks of causality or determinism. As already mentioned in the introduction, there may exist events that are not completely determinable. Probability theory was developed to model and analyze situations and experiments with ambiguous outcomes. Simple examples of such experiments are tossing a coin or rolling a 6-sided die. If there is no (apparent) possibility of predicting the outcome of such experiments, one speaks of **random events**. When modeling a probabilistic experiment, one considers all possible outcomes of the experiment, called **elementary events**. From a philosophical point of view it is important that these elementary events are atomic. Atomic means that an elementary event cannot be viewed as a collection of other even more elementary events of the experiments, and so one elementary event excludes any other elementary event. The set of all elementary events of an experiment is called the **sample space** of the experiment.

For the tossing of a coin, the elementary events are “head” and “tail”. For the rolling of a 6-sided die the elementary events are “1”, “2”, “3”, “4”, “5”, and “6”.

An **event** is a set of elementary events (i.e., a subset of the set[1] of elementary events). For instance, $\{2, 4, 6\}$ is an event of die rolling that corresponds to rolling an even number. Since elementary events can be also considered events, we represent elementary events as one element sets.

In the following we consider only experiments with finitely many elementary events to increase the transparency of the next definition. Our aim now is to develop a reasonable theory that assigns a probability to every event. This aim was not easy to achieve. Probability theory took almost 300 years to advance from the works of Pascal, Fermat, and Huygens in the middle of the 17th century to the currently accepted axiomatic definition of probability by Kolmogorov. Limiting the set S of elementary events to a finite set is helpful for overcoming the technicalities associated with a possible uncountability of S in the general definition of Kolmogorov[2]. The basic idea is to define the probability of an event E as

the ratio between the sum of probabilities of (favorable) elementary events involved in E to the sum of the probabilities of all possible elementary events. (2.1)

[1]of the sample space

[2]These technicalities arise for a correct handling of sets of elementary events that are not countable.

Fixing the probability of events in this way, one standardizes the probability values, in the sense that probability 1 corresponds to a certain event[3] and probability 0 corresponds to an impossible (empty[4]) event.

Another important point is that the probabilities of elementary events unambiguously determine the probabilities of all events.

For symmetric experiments such as tossing a coin, one wants to assign the same probability to all elementary events.

Let $\mathrm{Prob}(E)$ be the probability of an event E. In our model, the result of the experiment must be one of the elementary events; hence we set

$$\mathrm{Prob}(S) = 1$$

for the sample set S of all elementary events. Then, for the rolling of a die, we have

$$\begin{aligned}\mathrm{Prob}(\{2,4,6\}) &= \frac{\mathrm{Prob}(\{2\}) + \mathrm{Prob}(\{4\}) + \mathrm{Prob}(\{6\})}{\mathrm{Prob}(S)} \\ &= \mathrm{Prob}(\{2\}) + \mathrm{Prob}(\{4\}) + \mathrm{Prob}(\{6\}) \\ &= \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2},\end{aligned}$$

i.e., the probability of getting an even number is exactly $1/2$. Following the concept (2.1) of measuring probability, we obtain

$$\begin{aligned}\mathrm{Prob}(A \cup B) &= \frac{\mathrm{Prob}(A) + \mathrm{Prob}(B)}{\mathrm{Prob}(S)} \\ &= \mathrm{Prob}(A) + \mathrm{Prob}(B)\end{aligned}$$

for all disjoint events A and B. These considerations result in the following axiomatic definition of probability. Let, for any set A,

$$\mathcal{P}(A) = \{B \mid B \subseteq A\}$$

be the power set of A.

Definition 2.2.1. *Let S be the sample space of a probability experiment. A* **probability distribution on S** *is every function*

$$\mathrm{Prob} : \mathcal{P}(S) \to [0,1]$$

that satisfies the following conditions (probability axioms):

(i) $\mathrm{Prob}(\{x\}) \geq 0$ *for every elementary event x,*
(ii) $\mathrm{Prob}(S) = 1$*, and*
(iii) $\mathrm{Prob}(A \cup B) = \mathrm{Prob}(A) + \mathrm{Prob}(B)$ *for all events $A, B \subseteq S$ with $A \cap B = \emptyset$.*

[3] to the event S consisting of all elementary events
[4] called also null event

The value **Prob(A)** *is called the* **probability of the event A**. *The pair* (S, Prob) *is called a* **probability space**. *If*

$$\text{Prob}(\{x\}) = \text{Prob}(\{y\}) = \frac{1}{|S|}$$

for all $x, y \in S$, Prob *is called the* **uniform probability distribution on S**.

Observe, that exactly the condition (iii) of Definition 2.2.1 is the formalization of our informal concept (2.1) of probability.

Exercise 2.2.2. Prove that the following properties hold for every probability space (S, Prob):

(i) $\text{Prob}(\emptyset) = 0$.
(ii) $\text{Prob}(S - A) = 1 - \text{Prob}(A)$ for every $A \subseteq S$.
(iii) $\text{Prob}(A) \leq \text{Prob}(B)$, for all $A, B \subseteq S$ with $A \subseteq B$.
(iv) $\text{Prob}(A \cup B) = \text{Prob}(A) + \text{Prob}(B) - \text{Prob}(A \cap B)$
$\leq \text{Prob}(A) + \text{Prob}(B)$ for all $A, B \subseteq S$.
(v) $\text{Prob}(A) = \sum_{x \in A} \text{Prob}(x)$ for all $A \subseteq S$.

We observe that all properties from 2.2.2 correspond to our intuition, and hence to the informal concept (2.1) of probability.

Thus the addition of probabilities corresponds to the idea that the probability of several pairwise exclusive (disjoint) events is the sum of the probabilities of these events.

Exercise 2.2.3. In fact, the condition (v) of Exercise 2.2.2 is an exact formulation of the notion of probability with respect to (2.1). Hence, one can take the conditions (i) and (ii) of Definition 2.2.1 and (v) of Exercise 2.2.2 in order to get an alternative definition of probability distribution. Prove that such a definition of probability distribution on S is equivalent to Definition 2.2.1.

Example 2.2.4. Let us consider the sample space

$$S_3 = \{(x, y, z) \mid x, y, z \in \{\text{head}, \text{tail}\}\}$$

of the experiment of tossing three coins. We account for the order of tossing the coins, i.e., an elementary event (x_1, x_2, x_3) describes the outcome of the experiment where x_i is the result of the i-th coin tossing. Assuming a "fair" tossing of "fair" coins, one has

$$\text{Prob}(\{a\}) = \frac{1}{|S_3|} = \frac{1}{8}$$

for each elementary event $a \in S_3$.

How high is the probability of getting at least one head? The corresponding event is

$$\begin{aligned}\text{HEAD} = \{&(\text{head},\text{head},\text{head}),(\text{head},\text{head},\text{tail}),(\text{head},\text{tail},\text{head}),\\&(\text{tail},\text{head},\text{head}),(\text{head},\text{tail},\text{tail}),(\text{tail},\text{head},\text{tail}),\\&(\text{tail},\text{tail},\text{head})\}.\end{aligned}$$

Hence,

$$\text{Prob}(\text{HEAD}) = \sum_{a\in\text{HEAD}} \text{Prob}(\{a\}) = 7\cdot\frac{1}{|S_3|} = \frac{7}{8}.$$

To estimate Prob(HEAD) in an easier way, one can consider the complementary event

$$S_3 - \text{HEAD} = \{(\text{tail},\text{tail},\text{tail})\}$$

to S_3. Based on the fact

$$\text{Prob}(S_3 - \text{HEAD}) = \frac{1}{|S_3|} = \frac{1}{8}$$

one immediately gets (see also (ii) of Exercise 2.2.2)

$$\begin{aligned}\text{Prob}(\text{HEAD}) &= \text{Prob}(S_3 - (S_3 - \text{HEAD})) =\\&= 1 - \text{Prob}(S_3 - \text{HEAD}) = 1 - \frac{1}{8} = \frac{7}{8},\end{aligned}$$

because HEAD $= (S_3 - (S_3 - \text{HEAD}))$. □

Exercise 2.2.5. Let n and k be integers, $n \geq k \geq 0$. Let us consider the experiment of tossing n coins and let (S_n, Prob) be the corresponding probability space, where

(i) $S_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in \{\text{head},\text{tail}\}\}$, and
(ii) Prob is a uniform probability distribution on S_n.

(i) How large is the probability to get "head" exactly k times?
(ii) How large is the probability to get "head" at least (at most) k times?

Exercise 2.2.6. As already observed, condition (iii) of Definition 2.2.1 (or condition (v) of Exercise 2.2.2) is the kernel of the definition of a probability distribution on a sample space S. Condition (i) requiring $\text{Prob}(\{x\}) \geq 0$ for every $x \in S$ is natural, because we no not have any interpretation for negative probabilities. But condition (ii) is not necessary. Why did we decide to introduce (ii)? What drawbacks could appear if (ii) were not part of the definition of probability distribution ?

We agreed above that the addition of probabilities corresponds to the fundamental idea that the probability of several pairwise exclusive (disjoint) events is the sum of the probabilities of these events. Now, we prove the following fundamental question:

To what does the multiplication of probabilities correspond?

Consider two probabilistic experiments that are independent in the sense that the result of an experiment has no influence on the result of the other experiment. An example of such a situation is the rolling of a die twice. It does not matter, whether one rolls two dice at once or whether one uses the same die twice, because the results do not influence each other. For instance, the elementary event “3” of the first roll does not have any influence on the result of the second roll. We know that $\text{Prob}(\{i\}) = \frac{1}{6}$ for both experiments, and for all $i \in \{1, 2, \ldots, 6\}$. Consider now joining both probabilistic experiments into one probabilistic experiment. The set of elementary events of this joint experiment is

$$Q_2 = \{(i, j) \mid i, j \in \{1, 2, \ldots, 6\}\}$$

where, for an elementary event (i, j) of Q_2, i is the result of the first roll and j is the result of the second roll. What is the fair probability distribution Prob_2 over Q_2, that can be determined from the basic experiment$(\{1, 2, \ldots, 6\}, \text{Prob})$? We consider our hypothesis that

> the probability of an event consisting of two fully independent events is equal to the product of the probabilities of these events,

so that

$$\text{Prob}_2(\{(i, j)\}) = \text{Prob}(\{i\}) \cdot \text{Prob}(\{j\}) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}$$

for all $i, j \in \{1, 2, \ldots, 6\}$. We verify the correctness of this hypothesis for our example. The sample space Q_2 contains exactly 36 elementary events, and each of these elementary events is equally probable. Hence

$$\text{Prob}_2(\{(i, j)\}) = \frac{1}{36}$$

for all $(i, j) \in Q_2$.

Exercise 2.2.7. Let k be a positive integer. Let (S, Prob) be a probability space where Prob is a uniform probability distribution over the sample space $S = \{0, 1, 2, \ldots, 2^k - 1\}$. Create (S, Prob) from k coin tossing experiments.

The above example confirms our intuition (hypothesis), but it does not suffice for fully realizing exactly why the term of independency of two events is so strongly related to the product of their probabilities. To be really convinced about the truthfulness (validity) of our hypothesis, we introduce the notion of conditional probabilities.

The notions defined above are suitable and useful when looking at an experiment only once, namely at the very end. But sometimes one can obtain partial information about the outcome of an experiment by some intermediate observation. For instance, we flip three coins one after the other and look at the result of the first coin flipping. Knowing this result one can ask for estimating the probability of getting at least two heads in the whole experiment. Or, somebody tells us that the result (x, y, z) contains at least one head and knowing this fact, we have to estimate the probability that (x, y, z) contains at least two heads. The tasks of this kind result in the following definition of conditional probability, which is an important instrument in the analysis of probabilistic experiments.

Definition 2.2.8. *Let* (S, Prob) *be a probability space. Let* A *and* B *be events* $(A, B \subseteq S)$*, and let* $\text{Prob}(B) \neq 0$.

The **conditional probability** *of the event* A *given that the event* B *occurs (with certainty) is*

$$\mathbf{Prob}\,(\boldsymbol{A}|\boldsymbol{B}) = \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)}.$$

We also say $\text{Prob}(A|B)$ *is the* **probability of $\boldsymbol{A}$ given $\boldsymbol{B}$**.

Observe that the definition of conditional probability is natural,[5] because

(i) $A \cap B$ consists of all elementary events that are in both A and B, and
(ii) B happens with certainty, i.e., no elementary event from $A - B$ can occur.[6]

Thus, when dividing $\text{Prob}(A \cap B)$ by $\text{Prob}(B)$, one normalizes the probabilities of all elementary events in B, because

$$\sum_{e \in B} \frac{\text{Prob}(\{e\})}{\text{Prob}(B)} = \frac{1}{\text{Prob}(B)} \cdot \sum_{e \in B} \text{Prob}(\{e\})$$
$$= \frac{1}{\text{Prob}(B)} \cdot \text{Prob}(B) = 1.$$

The intuitive meaning of conditional probabilities we try to formalize is that

> *the sample space* S *is exchanged for the sample space* $B \subseteq S$*, because* B *now contains all elementary events that may appear (no elementary event from* $S - B$ *can occur). Thus, the conditional probability of* A *given* B *is the ratio of the probability of the event* $A \cap B$ *and the event* B.

One can consider the exchange of S for B as follows. Every elementary event $s \in S$ has probability $\text{Prob}(\{s\})$ in the probability space (S, Prob). When S is exchanged for a $B \subset S$, and B occurs with certainty, one has to take a new probability space (B, Prob_B) as the model of the experiment. This means one has to fix the probability $\text{Prob}_B(\{s\})$ for each $s \in B$. How do we do it in the sense of our informal concept of conditional probability? The only requirements are

[5]i.e., it corresponds to our intuition about the meaning of independence
[6]Observe that $A = (A \cap B) \cup (A - B)$.

(i) $\text{Prob}_B(B) = 1$, and
(ii) the ratios between the probabilities of arbitrary two elementary events from $B \subseteq S$ may not change,[7] i.e.,

$$\frac{\text{Prob}(\{a\})}{\text{Prob}(\{b\})} = \frac{\text{Prob}_B(\{a\})}{\text{Prob}_B(\{b\})}$$

for all $a, b \in B$ with $\text{Prob}(\{b\}) \neq 0$.

Definition 2.2.8 forces us to take

$$\text{Prob}_B(\{s\}) = \text{Prob}(\{s\} \mid B) = \frac{\text{Prob}(\{s\})}{\text{Prob}(B)}$$

for every $s \in B$, and so

$$\frac{\text{Prob}_B(\{a\})}{\text{Prob}_B(\{b\})} = \frac{\frac{\text{Prob}(\{a\})}{\text{Prob}(B)}}{\frac{\text{Prob}(\{b\})}{\text{Prob}(B)}} = \frac{\text{Prob}(\{a\})}{\text{Prob}(\{b\})}$$

for all $a, b \in B$ with $\text{Prob}(\{b\}) \neq 0$.

Exercise 2.2.9. Let (S, Prob) be a probability space. Let $B \subseteq S$, $\text{Prob}(B) \neq 0$, and let

$$\text{Prob}_B(A) = \text{Prob}(A|B)$$

for all $A \subseteq B$ be a function from $\mathcal{P}(B)$ to $[0, 1]$. Prove that (B, Prob_B) is a probability space.

Example 2.2.10. Consider the experiment of flipping three coins. To model it we used the probability space (S_3, Prob), where

$$S_3 = \{(x, y, z) \mid x, y, z \in \{\text{head}, \text{tail}\}\}$$

and Prob is the uniform probability distribution on S_3. Let A be the event that the outcome involves at least two heads and let B the event that the result of the experiment contains at least one tail. Obviously,

$$A = \{(\text{head}, \text{head}, \text{head}), (\text{head}, \text{head}, \text{tail}), (\text{head}, \text{tail}, \text{head}), (\text{tail}, \text{head}, \text{head})\}$$

and so

$$\text{Prob}(A) = \frac{4}{8} = \frac{1}{2}.$$

Since $S - B = \{(\text{head}, \text{head}, \text{head})\}$, we have $\text{Prob}(S - B) = \frac{1}{8}$, and so

$$\text{Prob}(B) = \text{Prob}(S - (S - B)) = 1 - \frac{1}{8} = \frac{7}{8}.$$

Since $A \cap B = \{(\text{head}, \text{head}, \text{tail}), (\text{head}, \text{tail}, \text{head}), (\text{tail}, \text{head}, \text{head})\}$, we have

[7] with respect to the ratios in the probability space (S, Prob)

$$\text{Prob}(A \cap B) = \frac{3}{8}.$$

From Definition 2.2.8 of conditional probability we obtain

$$\text{Prob}(A|B) \underset{def.}{=} \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)} = \frac{\frac{3}{8}}{\frac{7}{8}} = \frac{3}{7}.$$

Let us check whether this corresponds to our intuition of conditional probability. If one knows that B occurs with certainty, then the sample space S_3 is reduced to B, because the elementary events from $S_3 - B$ cannot occur. Since all elementary events in S_3 have the same probability, the elementary events of B have to have the same probability too. In this way, we get a new probability space (B, Prob_B) with

$$\text{Prob}_B(\{x\}) = \frac{1}{|B|} = \frac{1}{7}$$

for every elementary event $x \in B$. We extend the definition of Prob_B for all $C \subseteq B$ by

$$\text{Prob}_B(C) = \sum_{s \in C} \text{Prob}_B(\{s\}).$$

We observe that Prob_B satisfies the conditions (i), (ii), and (iii) of Definition 2.2.1, and so (B, Prob_B) is a probability space. If one wants to study the occurrence of A in (B, Prob_B), then one has to reduce A to $A_B = A \cap B$, because no $y \in A - B$ is an elementary event in B. Clearly, $\text{Prob}_B(A_B) = \text{Prob}_B(A \cap B)$ is the probability of the occurrence of an elementary event from A in the probability space (B, Prob_B), and so the conditional probability of A given B. Let us estimate $\text{Prob}_B(A \cap B)$ in (B, Prob_B). Since $|A \cap B| = 3$, we have

$$\text{Prob}_B(A \cap B) = \sum_{x \in A \cap B} \text{Prob}_B(\{x\}) = \frac{1}{7} + \frac{1}{7} + \frac{1}{7} = \frac{3}{7}.$$

We see that $\text{Prob}(A|B) = \text{Prob}_B(A \cap B)$. □

Our hypothesis is that the probability of the occurrence of two independent[8] events is the product of the probabilities of these events. Let us set this hypothesis in a formal definition and check whether it really corresponds to our interpretation of independence.

Definition 2.2.11. *Let* (S, Prob) *be a probability space. Two events* $A, B \subseteq S$ *are* **independent**, *if*

$$\text{Prob}(A \cap B) = \text{Prob}(A) \cdot \text{Prob}(B).$$

[8]The occurrence of one of these events does not have any influence on the probability of the other event.

Example 2.2.12. Consider once again the experiment of flipping three coins. Let

$$A = \{(\text{head}, \text{head}, \text{head}), (\text{head}, \text{head}, \text{tail}), (\text{head}, \text{tail}, \text{head}), (\text{head}, \text{tail}, \text{tail})\}$$

be the event that the result of the first coin flipping is "head". Clearly,

$$\text{Prob}(A) = \frac{4}{|S_3|} = \frac{4}{8} = \frac{1}{2}.$$

Let

$$B = \{(\text{head}, \text{tail}, \text{head}), (\text{head}, \text{tail}, \text{tail}), (\text{tail}, \text{tail}, \text{head}), (\text{tail}, \text{tail}, \text{tail})\}$$

be the event that the outcome of the second coin flipping is "tail". Obviously,

$$\text{Prob}(B) = \frac{4}{|S_3|} = \frac{4}{8} = \frac{1}{2}.$$

Since $A \cap B = \{(\text{head}, \text{tail}, \text{head}), (\text{head}, \text{tail}, \text{tail})\}$, we have

$$\text{Prob}(A \cap B) = \frac{2}{8} = \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2} = \text{Prob}(A) \cdot \text{Prob}(B)\,.$$

Now, with respect to Definition 2.2.11 we can say that A and B are independent. Observe, that this exactly corresponds to our interpretation of independence because the result of the first coin flipping does not have any influence on the result of the second coin flipping, and vice versa. □

The following assertion relates independence and conditional probability to each other. This way, one has got a new definition of the independence of two events.

Lemma 2.2.13. *Let* (S, Prob) *be a probability space. Let* $A, B \subseteq S$ *and let* $\text{Prob}(B) \neq 0$. *Then, A and B are independent if and only if*

$$\text{Prob}(A|B) = \text{Prob}(A)\,.$$

Proof. We prove the equivalence by proving the two corresponding implications.

(i) Assume A and B are independent and $\text{Prob}(B) \neq 0$. From Definition 2.2.11, we have

$$\text{Prob}(A \cap B) = \text{Prob}(A) \cdot \text{Prob}(B)$$

and consequently

$$\begin{aligned} \text{Prob}(A|B) &\underset{def.}{=} \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)} = \frac{\text{Prob}(A) \cdot \text{Prob}(B)}{\text{Prob}(B)} \\ &= \text{Prob}(A)\,. \end{aligned}$$

(ii) Assume $\text{Prob}(A|B) = \text{Prob}(A)$ and $\text{Prob}(B) \neq 0$. Then,

$$\text{Prob}(A) = \text{Prob}(A|B) \underset{def.}{=} \frac{\text{Prob}(A \cap B)}{\text{Prob}(B)}.$$

Multiplying this equality by $\text{Prob}(B)$ one obtains

$$\text{Prob}(A \cap B) = \text{Prob}(A) \cdot \text{Prob}(B),$$

and so A and B are independent.

$\square$

Imagine that the assertion of Lemma 2.2.13 captures exactly our intuitive interpretation of the independence of two events.

If A and B are independent, then the occurrence of B with certainty does not change the probability of A.

We were able to transparently show, why for two independent events A and B, the equality

$$\text{Prob}(A \cap B) = \text{Prob}(A) \cdot \text{Prob}(B)$$

must hold. It is a direct consequence of the equality

$$\text{Prob}(A|B) = \text{Prob}(A),$$

which truly reflects our imagination of the term of independence of two events A and B.

Exercise 2.2.14. Let (S_3, Prob) be the probability space of the experiment of flipping three coins.

(i) Let A be the event that the number of heads is even. Does there exist an event $B \subset S_3$ such that A and B are independent?
(ii) Let D be the event that either the result of the second coin flipping is tail or the result of the third coin flipping is tail. Find all events C such that C and D are independent.

Exercise 2.2.15. Let (S, Prob) be a probability space, and let A and B be two events having positive probabilities. Prove the following equalities:[9]

(i) $\text{Prob}(A \cap B) = \text{Prob}(B) \cdot \text{Prob}(A|B)$,
(ii) $\text{Prob}(A|B) = \dfrac{\text{Prob}(A) \cdot \text{Prob}(B|A)}{\text{Prob}(B)}$, and
(iii) $\text{Prob}(A|B) = \dfrac{\text{Prob}(A) \cdot \text{Prob}(B|A)}{\text{Prob}(A) \cdot \text{Prob}(B|A) + \text{Prob}(S - A) \cdot \text{Prob}(B|S - A)}$.

[9]The equality (ii) is known as Bayes' Theorem.

Exercise 2.2.16. Let (S, Prob) be a probability space. Prove, for all events $A_1, A_2, \ldots, A_n \subseteq S$, such that

$$\begin{aligned}&\text{Prob}(A_1) \neq 0, \text{Prob}(A_1 \cap A_2) \neq 0, \ldots,\\ &\text{Prob}(A_1 \cap A_2 \cap \ldots \cap A_{n-1}) \neq 0,\end{aligned}$$

the following equality

$$\begin{aligned}\text{Prob}(A_1 \cap A_2 \cap \ldots \cap A_n) = {} & \text{Prob}(A_1) \cdot \text{Prob}(A_2|A_1) \cdot \\ & \cdot \text{Prob}(A_3|A_1 \cap A_2) \cdot \ldots \\ & \cdot \text{Prob}(A_n|A_1 \cap A_2 \cap \ldots \cap A_{n-1}) .\end{aligned}$$

Next, we define a term that is crucial for the analysis of random experiments. In this book it will become a powerful instrument for analyzing randomized algorithms.

Let $\mathbb{R}$ denote the set of real numbers.

Definition 2.2.17. *Let S be a finite*[10] *sample space of a probabilistic experiment. Every function X from S to $\mathbb{R}$ is called a* **(discrete) random variable** *on S.*

This means that by determining (choosing) an X, one associates a real number with every elementary event of S (outcome of the experiment). The main point is that we have a choice of the random variable, and we use it in order to express what is of our interest. For instance, we will later consider a probability space as a probability distribution over all computations (runs) of a randomized algorithm on a fixed given input. When one wants to investigate the efficiency of this randomized algorithm on this input, then it is convenient to choose a random variable that assigns its length to any computation. Using this random variable, one can calculate the "expected" complexity of the randomized algorithm on the given input as the weighted average value of this random variable, where the weights are given by the probabilities of particular computations. Another convenient choice for a random variable is to assign 1 to every computation calculating the correct output, and 0 to all computations with a wrong result. In this way one can investigate the probability[11] of computing the correct result for the given input.

When S is finite (countable), then the set

$$\mathbb{R}_X = \{x \in \mathbb{R} \mid \exists s \in S, \text{ such that } X(s) = x\}$$

is finite (countable) for any random variable from S to $\mathbb{R}$.

[10]To exchange "finite" for "countable" in this definition does not change anything for the terms defined.

[11]Again, this probability is the weighted average value of the random variable, where the weights are given by the probabilities of the elementary events (computations).

Definition 2.2.18. *Let* (S, Prob) *be a probability space and let* X *be a random variable on* S. *For every* $z \in \mathbb{R}$, *we define the event* $X = z$ *as*

$$\textbf{Event}\,(\boldsymbol{X = z}) = \{s \in S \mid X(s) = z\}.$$

The function $f_X : \mathbb{R} \to [0,1]$, *defined by*

$$\boldsymbol{f_X(z)} = \text{Prob}(\text{Event}(X = z))$$

for all $z \in \mathbb{R}$, *is called the* **probability density function** *of the random variable* X.

The **distribution function** *of* X *is a function* $\text{Dis}_X : \mathbb{R} \to [0,1]$, *defined by*

$$\textbf{Dis}_{\boldsymbol{X}}\boldsymbol{(z)} = \text{Prob}(X \leq z) = \sum_{\substack{y \leq z \\ y \in \mathbb{R}_X}} \text{Prob}(\text{Event}(X = y))\,.$$

In order to shorten our notation in what follows, we use the notation $X = z$ instead of $\text{Event}\,(X = z)$, and consequently the notation $\text{Prob}(X = z)$ is used instead of $\text{Prob}(\text{Event}\,(X = z))$.

Lemma 2.2.19. *Let* (S, Prob) *be a probability space and let* X *be a random variable on* S. *Then, for every* $z \in \mathbb{R}$,

(i) $\text{Prob}(X = z) \geq 0$,
(ii) $\sum_{y \in \mathbb{R}_X} \text{Prob}(X = y) = 1$, *and*
(iii) $f_X(z) = \text{Prob}(X = z) = \sum_{\substack{s \in S \\ X(s) = z}} \text{Prob}(\{s\})$.

Proof. The property (i) is obvious, because $X = z$ is an event in S. The property (iii) directly follows from the definition of f_X and from the fact, that Prob is a probability distribution over S (see (v) in Exercise 2.2.2). The property (ii) is a direct consequence of (iii), because

$$\sum_{y \in \mathbb{R}_X} \text{Prob}(X = y) \underset{(iii)}{=} \sum_{y \in \mathbb{R}_X} \sum_{\substack{s \in S \\ X(s) = y}} \text{Prob}(\{s\}) = \sum_{s \in S} \text{Prob}(\{s\}) = 1.$$

□

An interesting consequence of Lemma 2.2.19 is that by the choice of a random variable X on a sample space S of a probability space (S, Prob), one creates a new probability space $(\mathbb{R}_X, F_X)$, where

$$\mathbb{R}_X = \{x \in \mathbb{R} \mid \exists s \in S \text{ with } X(s) = x\}$$

and

$$F_X(B) = \sum_{y \in B} f_X(y)$$

for all $B \subseteq \mathbb{R}_X$.

Exercise 2.2.20. Let (S, Prob) be a probability space and let X be a random variable on S. Prove that $(\mathrm{I\!R}_X, F_X)$ is a probability space.

Example 2.2.21. Consider the experiment of rolling three dice. The outcome of rolling one die is one of the numbers $1, 2, 3, 4, 5$ and 6. We consider "fair" dice, so every outcome from $\{1, 2, 3, 4, 5, 6\}$ has the same probability. Thus, the experiment is described by the probability space (S, Prob) with

$$S = \{(a, b, c) \mid a, b, c \in \{1, 2, 3, 4, 5, 6\}\}$$

and

$$\mathrm{Prob}(\{s\}) = \frac{1}{63} = \frac{1}{216}$$

for all $s \in S$.

Consider the random variable X, defined by

$$X((a, b, c)) = a + b + c,$$

i.e., as the sum of the values of all three dice. For instance, $X((2, 5, 1)) = 2 + 5 + 1 = 8$. The probability of the event $X = 5$ is

$$\begin{aligned}
\mathrm{Prob}(X = 5) &= \sum_{\substack{s\in S\\ X(s)=5}} \mathrm{Prob}(\{s\}) \\
&= \sum_{\substack{a+b+c=5\\ a,b,c\in\{1,\ldots,6\}}} \mathrm{Prob}(\{(a, b, c)\}) \\
&= \mathrm{Prob}(\{(1, 1, 3)\}) + \mathrm{Prob}(\{(1, 3, 1)\}) \\
&\quad + \mathrm{Prob}(\{(3, 1, 1)\}) + \mathrm{Prob}(\{(1, 2, 2)\}) \\
&\quad + \mathrm{Prob}(\{(2, 1, 2)\}) + \mathrm{Prob}(\{(2, 2, 1)\}) \\
&= 6 \cdot \frac{1}{216} = \frac{1}{36}.
\end{aligned}$$

Let Y be the random variable defined by

$$Y((a, b, c)) = \max\{a, b, c\} \text{ for every } (a, b, c) \in S.$$

Let us calculate the probability of the event $Y = 3$, i.e., the highest outcome in the experiment is 3.

$$\begin{aligned}
\mathrm{Prob}(Y = 3) &= \sum_{\substack{s\in S\\ Y(s)=3}} \mathrm{Prob}(\{s\}) = \sum_{\substack{\max\{a,b,c\}=3\\ a,b,c\in\{1,\ldots,6\}}} \mathrm{Prob}(\{(a, b, c)\}) \\
&= \sum_{b,c\in\{1,2\}} \mathrm{Prob}(\{(3, b, c)\}) + \sum_{a,c\in\{1,2\}} \mathrm{Prob}(\{(a, 3, c)\}) \\
&\quad + \sum_{a,b\in\{1,2\}} \mathrm{Prob}(\{(a, b, 3)\}) + \sum_{c\in\{1,2\}} \mathrm{Prob}(\{(3, 3, c)\})
\end{aligned}$$

$$
\begin{aligned}
&+\sum_{b\in\{1,2\}} \text{Prob}(\{(3,b,3)\}) + \sum_{a\in\{1,2\}} \text{Prob}(\{(a,3,3)\})\\
&+\text{Prob}(\{(3,3,3)\})\\
&= \frac{4}{216}+\frac{4}{216}+\frac{4}{216}+\frac{2}{216}+\frac{2}{216}+\frac{2}{216}+\frac{1}{216}\\
&= \frac{19}{216}.
\end{aligned}
$$

□

Exercise 2.2.22. Estimate the probability density function Dis_X and the distribution function f_Y for the random variables X and Y from Example 2.2.21.

Next, we define the independency of two random variables as a natural generalization of the interdependence of two events.

Definition 2.2.23. *Let* (S, Prob) *be a probability space, and let* X *and* Y *be two random variables on* S*. We introduce the notation*

$$\text{Event}(X = x \textit{ and } Y = y) = \text{Event}(X = x) \cap \text{Event}(Y = y)\,.$$

We say that the random variables X *and* Y *are* **independent** *if, for all* $x, y \in \mathbb{R}$*,*

$$\text{Prob}(X = x \textit{ and } Y = y) = \text{Prob}(X = x) \cdot \text{Prob}(Y = y)\,.$$

To illustrate the notation of the independence of random variables, we consider the experiment of rolling three dice with the random variables X and Y defined by

$$X((a,b,c)) = \begin{cases} 1 & a \text{ is even} \\ 0 & a \text{ is odd} \end{cases}$$

and

$$Y((a,b,c)) = b$$

for all $(a,b,c) \in S$.

Clearly,

$$\text{Prob}(X = 1) = \frac{1}{2} = \text{Prob}(X = 0)$$

and

$$\text{Prob}(Y = i) = \frac{1}{6}$$

for all $i \in \{1, \ldots, 6\}$. Then, for all $i \in \{1, \ldots, 6\}$

$$
\begin{aligned}
\text{Prob}(X = 1 \text{ and } Y = i) &= \text{Prob}\left(\left\{ (a,i,c) \;\middle|\; \begin{array}{c} a \text{ is even, and} \\ c \in \{1,\ldots,6\} \end{array} \right\}\right)\\
&= \frac{18}{216} = \frac{1}{12} = \frac{1}{2}\cdot\frac{1}{6}\\
&= \text{Prob}(X = 1)\cdot\text{Prob}(Y = i)\,.
\end{aligned}
$$

Since the same equality holds for the case $X = 0$, we can conclude that X and Y are independent.[12]

The simplest and most useful characterization of the distribution of a random variable is the weighted average of the values it takes on. This average value is called the expected value in what follows. Our motivation for the study of the expected value of random variables is in applying it as an instrument for analyzing the efficiency and reliability of randomized algorithms.

Definition 2.2.24. *Let* (S, Prob) *be a probability space and let* X *be a random variable on* S*. The* **expectation** *of* X *(or the* **expected value** *of* X*) is*

$$\mathbf{E}[\boldsymbol{X}] = \sum_{x \in \mathbb{R}_X} x \cdot \text{Prob}(X = x)\,,$$

if this sum is finite[13] *or converges absolutely.*

The following lemma provides another possibility of calculating the expectation $\mathrm{E}[X]$ of a random variable X.

Lemma 2.2.25. *Let* (S, Prob) *be a finite probability space and let* X *be a random variable on* S*. Then,*

$$\mathrm{E}[X] = \sum_{s \in S} X(s) \cdot \text{Prob}(\{s\})\,.$$

Proof. For any random variable X on S

$$\begin{aligned}
\mathrm{E}[X] &\underset{def.}{=} \sum_{x \in \mathbb{R}_X} x \cdot \text{Prob}(X = x) \\
&= \sum_{x \in \mathbb{R}_X} x \cdot \sum_{\substack{s \in S \\ X(s) = x}} \text{Prob}(\{s\}) \\
&\qquad \{\text{since Event}(X = x) = \{s \in S \mid X(s) = x\}\} \\
&= \sum_{x \in \mathbb{R}_X} \sum_{\substack{s \in S \\ X(s) = x}} x \cdot \text{Prob}(\{s\}) \\
&= \sum_{x \in \mathbb{R}_X} \sum_{\substack{s \in S \\ X(s) = x}} X(s) \cdot \text{Prob}(\{s\}) \\
&= \sum_{s \in S} X(s) \cdot \text{Prob}(\{s\})
\end{aligned}$$

{since X is a function on S, i.e., for every $s \in S$ there exists exactly one $x \in \mathbb{R}$ with $X(s) = x$}

$\square$

[12]This may not be surprising, because the value of $X((a, b, c))$ depends on the outcome of the first die rolling only and the value of Y depends on the outcome of the second die rolling only.

[13]If S is finite, then this sum must be finite, too.

Exercise 2.2.26. Let (S, Prob) be a probability space and let X and Y be two random variables on S such that $X(s) \leq Y(s)$ for all $s \in S$. Prove that

$$\mathrm{E}[X] \leq \mathrm{E}[Y]\,.$$

In the analysis of randomized algorithms we frequently use a special type of random variable, called indicator variable. A random variable X is called an **indicator variable**, if it takes only values 0 and 1, i.e., if X is a function from S to $\{0, 1\}$. An indicator variable X on S partitions S into two subclasses. One class contains all elementary events s with $X(s) = 1$ and the other one contains all elementary events u with $X(u) = 0$. For instance, if S is the set of all runs (computations) of a randomized algorithm on a given input, then one can mark all runs s with the correct output by $X(s) = 1$ and all runs u with a wrong result by $X(u) = 0$. In general, for any event $A \subseteq S$, one can take an indicator variable $\boldsymbol{X_A}$, such that

$$\begin{aligned} A &= \{s \in S \mid X_A(s) = 1\} = \text{Event}(X_A = 1)\,, \text{ and} \\ S - A &= \{s \in S \mid X_A(s) = 0\} = \text{Event}(X_A = 0)\,. \end{aligned}$$

Then,

$$\begin{aligned} \mathrm{E}[X_A] &\underset{Lem.2.2.25}{=} \sum_{s \in S} X_A(s) \cdot \text{Prob}(\{s\}) \\ &= \sum_{s \in A} X_A(s) \cdot \text{Prob}(\{s\}) + \sum_{s \in S-A} X_A(s) \cdot \text{Prob}(\{s\}) \\ &= \sum_{s \in A} 1 \cdot \text{Prob}(\{s\}) + \sum_{s \in S-A} 0 \cdot \text{Prob}(\{s\}) \\ &= \sum_{s \in A} \text{Prob}(\{s\}) \\ &= \text{Prob}(A)\,. \end{aligned}$$

We have proved the following assertion.

Lemma 2.2.27. *Let (S, Prob) be a probability space. For every indicator variable X_A on S with $A = \{s \in S \mid X_A(s) = 1\}$,*

$$\mathrm{E}[X_A] = \text{Prob}(A)\,,$$

i.e., the expectation of X_A is equal to the probability of the event A.

For any random variable X on S and any function $g : \mathbb{R} \to \mathbb{R}$, the function $Z = g(X)$ defined by

$$Z(s) = g(X(s)) \text{ for all } s \in S,$$

is a random variable on S, too. For instance, if for given $a, b \in \mathbb{R}$,

$$g(y) = a \cdot y + b$$

for every $y \in \mathrm{I\!R}$, then

$$Z(s) = g(X(s)) = a \cdot X(s) + b,$$

and

$$\begin{aligned}
\mathrm{E}[Z] = \mathrm{E}[a \cdot X + b] &= \sum_{x \in \mathrm{I\!R}_X} (a \cdot x + b) \cdot \mathrm{Prob}(X = x) \\
&= a \cdot \sum_{x \in \mathrm{I\!R}_X} x \cdot \mathrm{Prob}(X = x) + b \cdot \sum_{x \in \mathrm{I\!R}_X} \mathrm{Prob}(X = x) \\
&= a \cdot \mathrm{E}[X] + b.
\end{aligned}$$

The property

$$\mathrm{E}[a \cdot X + b] = a \cdot \mathrm{E}[X] + b$$

of random variables is called **weak linearity of expectation**.

Often one needs random variables that are a combination of several random variables.

Definition 2.2.28. *Let* (S, Prob) *be a probability space. Let* $X_1, X_2, \ldots, X_n$ *be random variables on* S*. We denote by*

$$\boldsymbol{X_1 + X_2 + \ldots + X_n}$$

or by $\sum_{i=1}^{n} X_i$ *the random variable* $Z = \sum_{i=1}^{n} X_i$ *defined by*

$$Z(s) = X_1(s) + X_2(s) + \ldots + X_n(s) = \sum_{i=1}^{n} X_i(s)$$

for all $s \in S$*. We denote by*

$$\boldsymbol{X_1 \cdot X_2 \cdot \ldots \cdot X_n}$$

or by $\prod_{i=1}^{n} X_i$ *the random variable* Y *defined by*

$$Y(s) = Y_1(s) \cdot Y_2(s) \cdot \ldots \cdot Y_n(s) = \prod_{i=1}^{n} Y_i(s)$$

for all $s \in S$.

The following property of the expectation of $X_1 + X_2 + \ldots + X_n$ is often applied in the analysis of randomized systems.

Lemma 2.2.29. *Let* (S, Prob) *be a probability space and let* X *and* Y *be two random variables on* S*. Then*

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y].$$

Proof. Let Z denote the random variable $X + Y$. Then

$$\begin{aligned}
\mathrm{E}[X+Y] = \mathrm{E}[Z] &= \sum_{s\in S} Z(s)\cdot \mathrm{Prob}(\{s\}) \\
&\quad \{\text{by Lemma 2.2.25}\} \\
&= \sum_{s\in S} (X(s)+Y(s))\cdot \mathrm{Prob}(\{s\}) \\
&\quad \{\text{by definition of } Z = X+Y\} \\
&= \sum_{s\in S} X(s)\cdot \mathrm{Prob}(\{s\}) + \sum_{s\in S} Y(s)\cdot \mathrm{Prob}(\{s\}) \\
&= \mathrm{E}[X] + \mathrm{E}[Y] \\
&\quad \{\text{by Lemma 2.2.25}\}
\end{aligned}$$

□

The property

$$\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$$

of random variables is called **linearity of expectation**.

Exercise 2.2.30. Prove that

$$\mathrm{E}[X_1 + X_2 + \ldots + X_n] = \mathrm{E}[X_1] + \mathrm{E}[X_2] + \ldots + \mathrm{E}[X_n]$$

for all random variables $X_1, X_2, \ldots, X_n$ on a sample space S.

Exercise 2.2.31. Let (S, Prob) be a probability space and let X and Y be two different random variables on S. Let $Z = \min\{X, Y\}$ be the random variable defined by $Z(s) = \min\{X(s), Y(s)\}$ for every $s \in S$. Prove or disprove the following claim:

$$\mathrm{E}[Z] = \min\{\mathrm{E}[X], \mathrm{E}[Y]\}.$$

2.3 Models of Randomized Algorithms

The goal of this section is to show how randomized algorithms can be modeled by means of probability spaces, and how the concept of random variables can be used for analyzing randomized computations.

Randomized algorithms are a special case of stochastic algorithms. A stochastic algorithm can be viewed as an algorithm that is partially controlled by a random process. A stochastic algorithm is allowed to flip a "fair" coin whenever needed, and the outcome of the coin flipping is used to decide in which way the algorithm will continue in its work. The quality of a stochastic algorithm is usually measured in the running time and in the degree of reliability (correctness). The meaning of the term "degree of reliability" can be interpreted in different ways. For instance, one can consider a probability distribution over all feasible inputs, and then measure the degree of reliability as the probability of getting the correct result on a randomly chosen input. This means that one can accept a stochastic algorithm that behaves "poorly" (it runs for too long, or incorrectly) on some input instances when the algorithm is "good" (efficient, and correct with high probability) for most inputs. The randomized algorithms are a special case of stochastic algorithms in the sense that randomized algorithms are not allowed to behave poorly on any input instance.

> *The design of randomized algorithms is subject to the very strong requirement, that the randomized algorithms work efficiently and correctly with high probability on every input, i.e., that they are reliable for each input data.*

Because of the requirement of efficiently computing the correct result with a reasonable probability for each input, one has to investigate the behavior of a randomized algorithm on any feasible input. Therefore, it does not make any sense to consider here probability distributions on input sets. The only source of randomness under consideration is the randomized control of the algorithm itself. In what follows, we consider two models of randomized algorithms.

THE FIRST MODEL

This model is the simpler of the two models presented here, and it considers a randomized algorithm A as a probability distribution over a finite collection $A_1, A_2, \ldots, A_n$ of deterministic strategies algorithms[14]. This means that for any input w, A chooses an A_i at random and lets A_i work on w. This way there are exactly n computations of A on n, each one given by the computation of an algorithm from $\{A_1, A_2, \ldots, A_n\}$ on w. In what follows, we also use the term "**a run of $\boldsymbol{A}$ on $\boldsymbol{w}$**" for each of the n computations of A_i on w for $i = 1, \ldots, n$. Thus, we model the experiment of the work of A on an input w as the probability space

$$(S_{A,w}, \text{Prob}),$$

where $S_{A,w} = \{A_1, A_2, \ldots, A_n\}$ and Prob is a probability distribution over $S_{A,w}$.

[14]We prefare to use the term strategy because usually one understands an algorithm as a program that works correctly on any input. But here, we consider a strategy as a program that may fail for some particular inputs. To fail can mean computing a wrong output or working too long. Later we will call randomized algorithms, based on strategies that may fail in the second sense only (i.e., to be not efficient on any input), Las Vegas algorithms. Randomized algorithms that are allowed to err (i.e., to produce wrong outputs) will be called Monte Carlo algorithms.

Typically, Prob is a uniform probability distribution. We usually prefer to consider

$$S_{A,w} = \{C_1, C_2, \ldots, C_n\}$$

as the set of all runs (computations) of A on w, where C_i is the computation of the i-th deterministic algorithm on w. Given A_i and w, the computation of C_i is unambiguously determined and therefore formally it does not matter whether one consider $S_{A,w}$ as $\{A_1, A_2, \ldots, A_n\}$ or as $\{C_1, C_2, \ldots, C_n\}$. We prefer the latter because we in fact investigate the concrete computations $C_1, \ldots, C_n$, and not the general descriptions of the algorithms $A_1, \ldots, A_n$, in this random experiment.[15]

This modeling is transparently presented in Figure 2.1. For a given input w the randomized algorithm A chooses the i-th deterministic algorithm with probability $\text{Prob}(\{A_i\})$ for $i = 1, \ldots, n$ at the beginning, and the rest of the computation is completely deterministic.

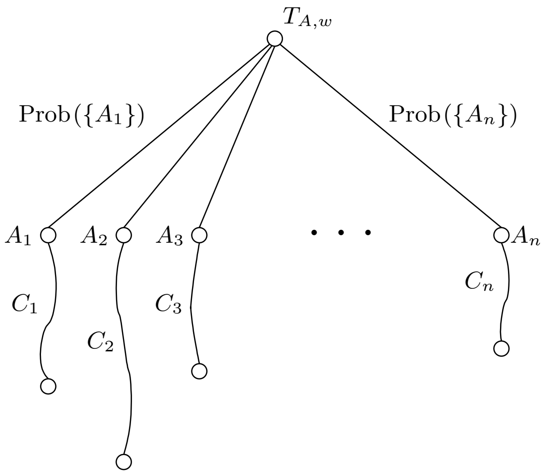


Fig. 2.1.

As indicated in Figure 2.1, the particular runs $\{C_1, C_2, \ldots, C_n\}$ of A on w can be of different lengths. Let **Time**$\boldsymbol{(C_i)}$ denote the length (the time complexity) of the computation C_i. If one wants to study the efficiency of A on w, one can consider the random variable[16] $Z : S_{A,w} \to \mathbb{N}$ defined by

$$Z(C_i) = \text{Time}(C_i) \text{ for } i = 1, \ldots, n.$$

Then one can measure the efficiency of the work of A on w by the **expected time complexity of** $\boldsymbol{A}$ **on** $\boldsymbol{w}$, defined by

[15]of the work of A on w

[16]In fact, one can directly use Time as the name of this random variable.

$$\textbf{Exp-Time}_{\boldsymbol{A}}\boldsymbol{(w)} = \mathrm{E}[Z] = \sum_{i=1}^{n} \mathrm{Prob}(\{C_i\}) \cdot Z(C_i)$$
$$= \sum_{i=1}^{n} \mathrm{Prob}(\{C_i\}) \cdot \mathrm{Time}(C_i)\,.$$

Since we are striving for an assured upper bound on the expected time complexity on any input of a length n, we define the expected time complexity of A in the worst-case manner.

The **expected time complexity of a randomized algorithm *A*** is the function $\text{Exp-Time}_A : \mathbb{N} \to \mathbb{N}$, defined by

$$\textbf{Exp-Time}_{\boldsymbol{A}}\boldsymbol{(n)} = \max\{\text{Exp-Time}_A(w) \mid \text{the length}^{17}\text{of } w \text{ is } n\}$$

for all $n \in \mathbb{N}$.

If one strives for a strong upper bound on the time complexity of every run of a randomized algorithm, one considers the **time complexity of a randomized algorithm *A*** as the function $\mathrm{Time}_A(n) : \mathbb{N} \to \mathbb{N}$, defined by

$$\textbf{Time}_{\boldsymbol{A}}\boldsymbol{(n)} = \max\{\mathrm{Time}(C) \mid C \text{ is a run of } A \text{ on an input of length } n\}.$$

If one wants to measure the "reliability" of a randomized algorithm A on an input w, the one can consider the indicator variable $X : S_{A,w} \to \{0,1\}$, defined by

$$X(C_i) = \begin{cases} 1 \text{ if } C_i \text{ computes the correct result on } w_i \\ 0 \text{ if } C_i \text{ computes a wrong result on } w_i \end{cases}$$

for $i = 1, \ldots, n$. Then,

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{n} X(C_i) \cdot \mathrm{Prob}(\{C_i\})\,. \\ &= \sum_{X(C_i)=1} 1 \cdot \mathrm{Prob}(\{C_i\}) + \sum_{X(C_i)=0} 0 \cdot \mathrm{Prob}(\{C_i\}) \\ &= \mathrm{Prob}(\mathrm{Event}(X=1)) \\ &= \text{the probability that } A \text{ computes the right result.} \end{aligned}$$

The value $\mathrm{E}[X]$ is called the **success probability of *A* on *w***. The value $1 - \mathrm{E}[X]$ is called the **error probability of *A* on *w***, denoted $\textbf{Error}_{\boldsymbol{A}}\boldsymbol{(w)}$.

[17]How one measures the input length is determined by concrete applications. For instance, the input length can be considered to be the number of symbols used for the representation of the input or, more roughly, the number of the elements (numbers, vertices of a graph, ...) included in the input object.

The **error probability of $\boldsymbol{A}$** is defined in the worst-case manner as the function

$$\mathbf{Error}_{\boldsymbol{A}}\boldsymbol{(n)} = \max\{\mathrm{Error}_A(w) \mid \text{the length of } w \text{ is } n\}$$

from $\mathbb{N}$ to $\mathbb{N}$.

In this way Error_A provides the guarantee that the error probability is at most $\mathrm{Error}_A(n)$ on every input of length n.

Observe that our definitions of time complexity measures assume that all runs of A are finite,[18] i.e., that A is not allowed to take an infinite computation. In general, randomized algorithms are allowed to have infinite computations. In that case one measures the expected time[19] complexity over the finite computations of A only, and calculates the probability of running an infinite computation, which is then added to the error probability (i.e., the occurrence of an infinite computation is considered to be an error).

In what follows we present two examples of randomized algorithms that can be successfully modeled and analyzed by the formalism described above.

Example 2.3.32. First, we consider the randomized protocol designed in Section 1.2 for the comparison of two strings x and y of length n. At the beginning, this protocol R uniformly chooses a prime p from the set[20] $\mathrm{PRIM}\left(n^2\right)$ at random. If C_p denotes the run of the protocol R given by the prime p on an input (x, y), then one can model the work of R on (x, y) by the probability space $(S_{R,(x,y)}, \mathrm{Prob})$, where

(i) $S_{R,(x,y)} = \{C_p \mid p \in \mathrm{PRIM}\left(n^2\right)\}$, and
(ii) Prob is a uniform probability distribution on $S_{R,(x,y)}$.

Considering the notation C_p one has a bijection between $\mathrm{PRIM}\left(n^2\right)$ and $S_{R,(x,y)}$, and so we may directly use the probability space

$$\left(\mathrm{PRIM}\left(n^2\right), \mathrm{Prob}\right)$$

for modeling the work of R on (x, y).

It does not make any sense here to investigate the expected communication complexity of R because all runs of R on (x, y) have the same[21] complexity, $4 \cdot \lceil \log_2 n \rceil$.

We have already fixed in Section 1.3 that, for any input (x, y) with $x = y$, the error probability is 0, and so we do not need to investigate it.

[18]In the first model this is naturally satisfied because one always assumes, that deterministic algorithms terminate on any input.

[19]the time complexity is not considered in such a case

[20]Remember that for any positive integer m, $\mathrm{PRIM}\,(m)$ denotes the set of all primes smaller than or equal to m.

[21]All natural numbers smaller than n^2 can be represented by $\lceil \log_2 n^2 \rceil$ bits. The protocol fixes the length of the binary representations of numbers submitted to $2 \cdot \lceil \log_2 n \rceil$, and so the length of the message is independent of p and $x \bmod p$.

To investigate the error probability of R on (x, y) for $x \neq y$ in the given formal framework, we choose the indication variable X defined by

$$X(C_p) = \begin{cases} 1 \text{ if } p \text{ is ``good''}^{22} \text{ for } (x,y) \\ 0 \text{ if } p \text{ is ``bad'' for } (x,y). \end{cases}$$

Since

$$\text{Prob}(C_p) = \frac{1}{Prim\,(n^2)} \text{ for all } p \in \text{PRIM}\left(n^2\right),$$

and we have proved in Section 1.3 that

the number of bad primes for any input (x, y) with $x \neq y$ is at most $n - 1$,

one obtains

$$\begin{aligned} \text{E}[X] &= \sum_{p \,\in\, \text{PRIM}(n^2)} X(C_p) \cdot \text{Prob}(\{C_p\}) \\ &= \sum_{p \,\in\, \text{PRIM}(n^2)} X(C_p) \cdot \frac{1}{Prim\,(n^2)} \\ &= \frac{1}{Prim\,(n^2)} \cdot \sum_{p \text{ is good}} X(C_p) \\ &\geq \frac{1}{Prim\,(n^2)} \cdot \left(Prim\left(n^2\right) - (n-1)\right) \\ &= 1 - \frac{n-1}{Prim\,(n^2)}. \end{aligned}$$

In this way we bound the error probability of R on (x, y) by

$$\begin{aligned} \text{Error}_R((x,y)) &= 1 - \text{E}[X] \\ &\leq \frac{n-1}{Prim\,(n^2)} \leq \frac{2 \cdot \ln n}{n} \\ &\quad \left\{\text{since } Prim\left(n^2\right) \geq n^2/\ln n^2 \text{ for all } n \geq 9\right\} \end{aligned}$$

for all $n \geq 9$.

Let us now consider the modified protocol R_2 that chooses two primes, p and q, at random, and accepts an input (x, y) if and only if

$$(x \bmod p = y \bmod p) \text{ and } (x \bmod q = y \bmod q). \tag{2.2}$$

A computation $C_{p,q}$ of R_2 consists of submitting

$$p, q, (x \bmod p), \text{ and } (x \bmod q)$$

[22]Remember that p is "good" for (x, y) iff the computation C_p produces the correct output for (x, y)

from R_{I} to R_{II}, and of the comparing of $x \bmod p$ with $y \bmod p$ and $x \bmod q$ with $y \bmod q$ by R_{II}. Analyzing the error probability of R_2, we choose the indicator variable Y defined as follows:

$$Y(C_{p,q}) = \begin{cases} 1 \text{ if } p \text{ or } q \text{ is good for } (x,y) \\ 0 \text{ if both } p \text{ and } q \text{ are bad for } (x,y) \end{cases}$$

for all $C_{p,q} \in S_{R_2,(x,y)} = \{C_{r,s} \mid r,s \in \text{PRIM}\left(n^2\right)\}$.

Analyzing in the probability space $(S_{R_2,(x,y)}, \text{Prob}_2)$, we obtain

$$\begin{aligned}
\text{E}[Y] &= \sum_{C_{p,q} \in S_{R_2,(x,y)}} Y(C_{p,q}) \cdot \text{Prob}_2(\{C_{p,q}\}) \\
&= \sum_{p,q \in \text{PRIM}(n^2)} Y(C_{p,q}) \cdot \frac{1}{(\mathit{Prim}\,(n^2))^2} \\
&= \text{Prob}_2(\text{Event}(Y=1)) \\
&= 1 - \text{Prob}_2(\text{Event}(Y=0)) \\
&= 1 - \text{Prob}_2(\text{Event}(p \text{ and } q \text{ are bad})) \\
&= 1 - \text{Prob}(p \text{ is bad}) \cdot \text{Prob}(q \text{ is bad}) \\
&\quad \{\text{Because } p \text{ and } q \text{ were chosen independently each of} \\
&\quad \text{each other.}\} \\
&\geq 1 - \left(\frac{n-1}{\mathit{Prim}\,(n^2)}\right)^2 \\
&\geq 1 - \frac{4 \cdot (\ln n)^2}{n^2}
\end{aligned}$$

for sufficiently large n. Since $\text{E}[Y]$ is the success probability of the computation of R_2 on (x,y) with $x \neq y$ (i.e., of the event $\text{Event}(Y=1)$ that R_2 outputs the right answer), the error probability of R_2 on (x,y) is

$$\text{Error}_{R_2}((x,y)) = 1 - \text{E}[Y] \leq \frac{4 \cdot (\ln n)^2}{n^2}.$$

□

Exercise 2.3.33. Consider the probabilistic experiment given by the protocol R_2. Define two indicator variables X_1 and X_2, where $X_1 = 1$ if and only if the first prime p is good for (x,y) and $X_2 = 1$ if and only if the second prime q is good for (x,y). Prove that X_1 and X_2 are independent. Can X_1 and X_2 be used to define the random variable Y used above?

Exercise 2.3.34. Let k be a positive integer, $k \geq 2$. Model the work of the protocol R_k choosing k primes from $\text{PRIM}\left(n^2\right)$ at random, and estimate the error probability of R_k.

Example 2.3.35. Consider the optimization problem MAX-SAT.[23] Given a formula Φ in CNF, one has to find an assignment to the variables of Φ such that the maximum possible number of clauses of Φ is satisfied. This problem is NP-hard, and so we weaken our requirement to find an optimal solution to the requirement to find a solution that satisfies a reasonably large proportion of clauses with high probability. This enables us to design the following simple and efficient algorithm for MAX-SAT.

Algorithm RSAM (Random Sampling)

Input: A formula $\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$ in CNF over a set $\{x_1, x_2, \ldots, x_n\}$ of Boolean variables.
Step 1: Choose an assignment $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\}^n$ to $x_1, x_2, \ldots, x_n$ at random with

$$\text{Prob}(\alpha_i = 1) = \text{Prob}(\alpha_i = 0) = \frac{1}{2}$$

for all $i = 1, \ldots, n$.
Output: $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

Thus, the algorithm RSAM simply generates a random assignment to the variables of Φ and takes it as the output. For any input the time complexity of RSAM is the length of the output. Since every algorithm for MAX-SAT must produce a feasible solution of this length, an asymptotically faster algorithm than RSAM does not exist.

Each output of the algorithm RSAM is an assignment to the variables of the given formula Φ, and so is a feasible solution for the problem instance Φ. In this sense there are no incorrect computations, and so we need not deal with error probability here.

We will measure the goodness of the algorithm RSAM as the ratio of the number of satisfied clauses to the number of all clauses. To do that, we define, for any input $\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$, m indicator variables $Z_1, Z_2, \ldots, Z_m$ by

$$Z_i(\alpha) = \begin{cases} 1 \text{ if the clause } F_i \text{ is satisfied by } \alpha \\ 0 \text{ if the clause } F_i \text{ is not satisfied by } \alpha \end{cases}$$

for every $\alpha \in \{0,1\}^n$. Clearly, the considered probability space is

$$(\{0,1\}^n, \text{Prob}),$$

where Prob is a uniform probability distribution over $\{0,1\}^n$. Further, we consider the random variable

$$Z = \sum_{i=1}^{m} Z_i,$$

[23]The formal definition of MAX-SAT is given in Example 2.5.64 (Section 2.5).

which counts the number of satisfied clauses. So, we are interested in learning $\mathrm{E}[Z]$. Because of the linearity of expectation, one has

$$\mathrm{E}[Z] = \mathrm{E}\left[\sum_{i=1}^{m} Z_i\right] = \sum_{i=1}^{m} \mathrm{E}[Z_i]\,.$$

To complete our calculation, we need to estimate $\mathrm{E}[Z_i]$. Since Z_i is an indicator variable, the expectation $\mathrm{E}[Z_i]$ is the probability that F_i is satisfied. Let, for $i = 0, \ldots, m$,

$$F_i = l_{i1} \vee l_{i2} \vee \ldots \vee l_{ik}$$

where l_{ij}s are k literals over k different[24] variables. The clause is not satisfied if and only if all k literals are not satisfied. The probability that a literal is not satisfied by a random assignment is exactly $1/2$. Since the random choice of a Boolean value of a variable is independent of the random choices of assignments to all other variables, and no pair of literals of F_i is over the same variable, the probability that none of the literals of F_i is satisfied is exactly

$$\left(\frac{1}{2}\right)^k = \frac{1}{2^k}.$$

Hence, the probability of satisfying F_i is

$$\mathrm{E}[Z_i] = 1 - \frac{1}{2^k}.$$

Since every clause consists of at least one literal (i.e., $k \geq 1$), we obtain

$$\mathrm{E}[Z_i] \geq \frac{1}{2}$$

for all $i \in \{1, \ldots, m\}$. Thus,

$$\mathrm{E}[Z] = \sum_{i=1}^{m} \mathrm{E}[Z_i] \geq \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2},$$

and so one expects that a random assignment α to the variables of a given formula satisfies at least half the clauses. Observe that our derivation of the lower bound $\mathrm{E}[Z_i] \geq \frac{m}{2}$ was very rough because one can expect that most of the clauses contain more than one literal. For instance, if all clauses contain at least 3 literals, then we would obtain $\mathrm{E}[Z_i] \geq \frac{7}{8}$ for all $i = 1, \ldots, m$, and so the expectation would be that at least $\frac{7}{8}$ of the clauses are satisfied. □

Exercise 2.3.36. Let us modify the algorithm RSAM as follows.

Input: A formula $\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$ over $\{x_1, x_2, \ldots, x_n\}$ in CNF.

[24] If this is not the case, one can simplify this clause.

Step 1: Choose uniformly an assignment $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ to $x_1, x_2, \ldots, x_n$ at random.

Step 2: Compute the number $r(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of clauses that are satisfied by $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

Step 3: If $r(\alpha_1, \alpha_2, \ldots, \alpha_n) \geq \frac{m}{2}$, then output $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, else repeat Step 1.

If this algorithm halts, then we have the assurance that it outputs an assignment satisfying at least half the clauses. Estimate the expected value of the number of executions of Step 1 of the algorithm (i.e., the expected running time of the algorithm).

Exercise 2.3.37. Let us choose t assignments of the variables of a given formula Φ at random. How large is the probability that the best one of these t assignments satisfies at least half the clauses of Φ?

THE SECOND MODEL

Sometimes it is more natural to represent a randomized algorithm as a nondeterministic algorithm with a probability distribution for every nondeterministic choice. To simplify the matter, one usually considers random choices from only two possibilities, each with probability 1/2. In general, one can describe all computations (runs) of a randomized algorithm A on an input w by the so-called computation tree[25] $T_{A,w}$ of A on w (Figure 2.2).

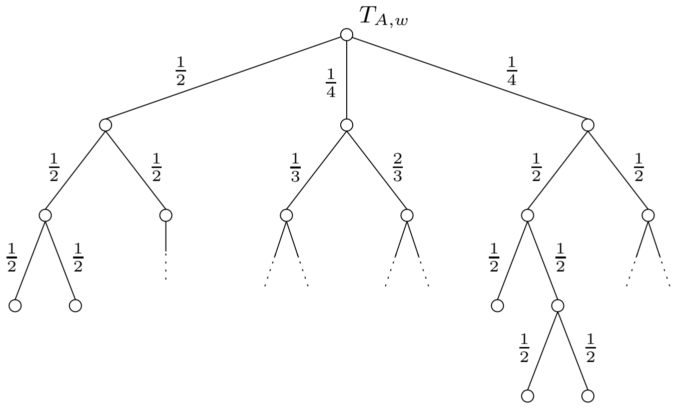


Fig. 2.2.

The vertices of the tree are labeled by the configurations of A. Every path from the root to a leaf in this tree corresponds to a computation of A on w.

[25]This tree is in fact the same as a computation tree of a nondeterministic algorithm, except for the labeling of the edges [Hro04].

Each edge of the tree is labeled by a value from $[0, 1]$ that determines the probability of its choice from the given configuration.

Taking the sample space $S_{A,w}$ as the set of all runs (computations) of A on w, one creates the probability space $(S_{A,w}, \text{Prob})$ by calculating the probability $\text{Prob}(C)$ of any computation $C \in S_{A,w}$ as the product of all labels (probabilities) of the edges of the corresponding path. Since every splitting of a computation (every random choice from at least two possibilities) is controlled by a probability distribution, the above defined function Prob from $S_{A,w}$ to $[0, 1]$ is a probability distribution[26] on $S_{A,w}$.

Clearly, the second model is a generalization of the first model. Because of the simplicity and transparency of the first model of randomized algorithms one prefers, if possible, to use the first way of modeling a randomized algorithm. The second model is used for describing algorithms in which one repeatedly makes a random choice after some deterministic parts of computations. The well known randomized Quicksort is a representative of the description of randomized algorithms in terms of the second model.

Example 2.3.38. We consider the problem of sorting the elements of a given set A by comparisons of pairs of elements only, assuming there is a linear order on all elements which are allowed to appear in the input. The well known randomized Quicksort can be described as the following recursive procedure.

Algorithm RQS(A)

```
Input: A set $A$ of elements with a given linear order.
Step 1: If $A = \{b\}$ (i.e., $|A| = 1$), then output "b".
    If $|A| \geq 2$, then choose an element $a \in A$ at random.
Step 2: Set
    $A_< := \{b \in A \mid b < a\}$
    $A_> := \{c \in A \mid c > a\}$
Step 3: output "RQS$(A_<)$, $a$, RQS$(A_>)$"
```

Obviously, the algorithm RQS(A) finishes its work with a sorted sequence of elements, and so there is no computation with a wrong output, i.e., the error probability is 0 for every input.

On the other hand, we observe that the complexities of different computations measured by the number of the executed comparisons of pairs of elements can essentially differ. Step 2 of RQS(A) forces exactly $|A| - 1$ comparisons, because every element in $A - \{a\}$ has to be compared with the pivot element a. Observe that RQS(A) has exponentially many[27] different computations on A. If RQS always chooses the smallest or the largest element of the set to be sorted, then any of the resulting computations has exactly $n - 1$ recursive calls, and the number of executed comparisons is

[26]This claim can be easily verified by induction on the depth of $S_{A,w}$.

[27]in $|A|$

$$\sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2} \in O\big(n^2\big) .$$

However when RQS always chooses the median as a pivot, then the classical analysis of this “divide and conquer” strategy provides the recurrence

$$\mathrm{Time}_{\mathrm{RQS}}(n) \leq 2 \cdot \mathrm{Time}_{\mathrm{RQS}}\left(\frac{n}{2}\right) + n - 1,$$

whose solution is $\mathrm{Time}_{\mathrm{RQS}}(n) \in O(n \cdot \log n)$. The corresponding computation has only $\log_2 n$ recursive calls. Since one can still show that, for the recurrence

$$T(n) \leq T(\frac{n}{8}) + T(\frac{7}{8} \cdot n) + n - 1,$$

$T(n) \in O(n \cdot \log n)$, RQS will behave well also if the size of $|A_<|$ very roughly approximates $|A_>|$. But this happens with probability at least 3/4, because at least $6/8 = 3/4$ of the elements are good choices. This is the reason for our hope that the algorithm RQS behaves very well on average. In what follows, we carefully analyze the expected complexity of RQS.

We have observed that the computation tree of RQS on an input is not only very large (especially its breadth), but that it is also very irregular. The depths (lengths) of the paths range from $\log_2 n$ to $n-1$, and consequently the corresponding computations have very different execution probabilities. If one considers the probability space $(S_{\mathrm{RQS}(A)}, \mathrm{Prob})$[28] and chooses the random variable X defined by

$$X(C) = \text{the number of comparisons in } C$$

for every computation $C \in S_{\mathrm{RQS}(A)}$, then one can use X to calculate the expected complexity of RQS, which is $\mathrm{E}[X]$. But because of the size and irregularity of the computation tree $T_{\mathrm{RQS}(A)}$, the analysis of $\mathrm{E}[X]$ is not only nontrivial, but also requires a lot of effort. Our next goal is to show that a suitable choice of random variables is determinig not only the success of the algorithm analysis, but also the hardness of the analysis.

Let $s_1, s_2, \ldots, s_n$ be the output of RQS(A), i.e., let s_i be the ith smallest element in A. We define the indicator variables X_{ij} by

$$X_{ij}(C) = \begin{cases} 1 \text{ if } s_i \text{ and } s_j \text{ were compared in the run } C \\ 0 \text{ if } s_i \text{ and } s_j \text{ were not compared in } C \end{cases}$$

for all $i, j \in \{1, \ldots, n\}$, $i < j$. Obviously, the random variable T defined by

$$T(C) = \sum_{i=1}^{n} \sum_{j>i} X_{ij}(C)$$

[28]where Prob is not a uniform probability distribution over S_{RQS}

counts the total number of comparisons, and so $\mathrm{E}[T] = \text{Exp-Time}_{\mathrm{RQS}}(A)$. Due to linearity of expectation we have

$$\mathrm{E}[T] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i} \mathrm{E}[X_{ij}]\,. \tag{2.3}$$

In order to estimate $\mathrm{E}[T]$, it remains to estimate the expectations $\mathrm{E}[X_{ij}]$ for all $i, j \in \{1, \ldots, n\}$, $i < j$. Let p_{ij} denote the probability that s_i and s_j are compared in an execution of $\mathrm{RQS}(A)$. Since X_{ij} is an indicator variable, we have[29]

$$\mathrm{E}[X_{ij}] = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}, \tag{2.4}$$

and so it remains to estimate p_{ij}. In which computations are s_i and s_j compared? Only in those in which one of the elements s_i and s_j is chosen as a pivot at random by $\mathrm{RQS}(A)$ before any of the elements $s_{i+1}, s_{i+2}, \ldots, s_{j-1}$ between s_i and s_j have been chosen as a pivot (Figure 2.3). Namely, if the first pivot of the set $\{s_i, s_{i+1}, \ldots, s_j\}$ is an element from the set $\text{Middle} = \{s_{i+1}, \ldots, s_{j-1}\}$, then s_i is put into $A_<$ and s_j is put into $A_>$, and so s_i and s_j cannot be compared in the rest of the computation. On the other hand, if s_i [or s_j] is chosen as the first pivot element of the set $\{s_i, \ldots, s_j\}$, then s_i [s_j] is compared with all elements from $\text{Middle} \cup \{s_j\}$ [or $\text{Middle} \cup \{s_i\}$], and so with s_j [s_i].

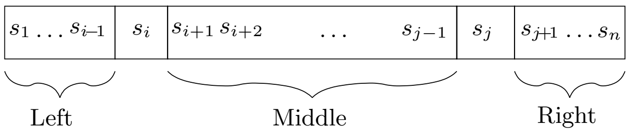


Fig. 2.3.

Each of the elements of A has the same probability of being chosen as the first pivot. If an element from Left or Right (Figure 2.3) is chosen, then s_i and s_j are put into the same set $A_>$ or $A_<$, and so this choice does not have any influence on whether s_i and s_j will be compared later or not. Thus, we may model this situation as a probabilistic experiment of uniformly choosing an element from $\text{Middle} \cup \{s_i, s_j\}$ at random. Hence,

$$p_{ij} = \frac{|\{s_i, s_j\}|}{|\text{Middle} \cup \{s_i, s_j\}|} = \frac{2}{j - i + 1}. \tag{2.5}$$

Let $\mathrm{Har}(n) = \sum_{k=1}^{n} \frac{1}{k}$ be the nth Harmonic number.[30] Inserting (2.5) into (2.3) we obtain

[29]see Lemma 2.2.27.

[30]For some estimations of $\mathrm{Har}(n)$, see Section A.3.

$$\begin{aligned}
\mathrm{E}[T] &\underset{\substack{(2.3)\\(2.4)}}{=} \sum_{i=1}^{n}\sum_{j>i} p_{ij} \\
&\underset{(2.5)}{=} \sum_{i=1}^{n}\sum_{j>i} \frac{2}{j-i+1} \\
&\leq \sum_{i=1}^{n}\sum_{k=1}^{n-i+1} \frac{2}{k} \\
&\quad \{\text{substituting } k = j-i+1\} \\
&\leq 2\cdot\sum_{i=1}^{n}\sum_{k=1}^{n} \frac{1}{k} \\
&= 2\cdot\sum_{i=1}^{n} \mathrm{Har}\,(n) \\
&= 2\cdot n\cdot \mathrm{Har}\,(n) \\
&\quad \{\text{Exercise A.3.67}\} \\
&= 2\cdot n\cdot \ln n + \Theta(n).
\end{aligned}$$

In this way we have showed that $\text{Exp-Time}_{\text{RQS}}(n) \in O(n \cdot \log n)$, and so randomized Quicksort is an efficient algorithm for sorting. □

Exercise 2.3.39. Construct the computation tree $T_{\text{RQS},A}$ for an A with $|A| = 5$.

Exercise 2.3.40. Consider the following problem. Given a set A of elements with a linear order and a positive integer k, $k \leq |A|$, find the k-th smallest element of A. Obviously, one can solve this problem by sorting A and then taking the k-th smallest element. But one knows that the complexity of sorting by comparisons is in $\Theta(n \cdot \log n)$ and so this naive approach is too costly for searching for the k-th position in the sorted sequence only.

Therefore we propose the following algorithm RSEL (*random select*) for this problem.

Algorithm RSEL(A, k)

```
Input: $A = \{a_1, a_2, \ldots, a_n\}$, $n \in \mathbb{N} - \{0\}$, and an integer $k$ satisfying $1 \leq k \leq n$.
Step 1:
    if $n = 1$ then
        output "$a_1$";
    else
        choose uniformly an $i \in \{1, 2, \ldots, n\}$ at random;
```

```
Step 2: Compute
    $A_< := \{b \in A \mid b < a_i\}$;
    $A_> := \{c \in A \mid c > a_i\}$;
Step 3:
    if $|A_<| > k$ then
        RSEL$(A_<, k)$;
    else if $|A_<| = k - 1$ then
        output "$a_i$";
    else
        RSEL$(A_>, k - |A_<| - 1)$;
```

Clearly, the algorithm RSEL(A, k) computes the correct output in each run. Show that $\text{Exp-Time}_{\text{RSEL}}(A, k) \in O(n)$.

In this section we have introduced two ways of modeling randomized algorithms. It is important to say that the choice of the model is first of all the question of transparency. We have already observed that the first model is a special case of the second one. But analyzing a randomized algorithm on a fixed input, one can also represent the second model by the first one. One simply generates a sufficiently large sequence of random bits at the very beginning of the computation. Then each such random sequence determines a run of the algorithm on the given input. Obviously, it does not matter when the random bits are used (viewed), whether in the beginning or later one after each other during the computation. But this representation of the second model by the first model is not uniform[31] and so we do not see any possibility of expressing any randomized algorithm described by the second model in terms of the first model as a probability distribution over a finite set of deterministic algorithms.

2.4 Classification of Randomized Algorithms

The aim of this section is to introduce the fundamental and generally accepted classification of randomized algorithms. This classification is based on the kind and size of the error probability of randomized algorithms, and has nothing to do with their design methodology. Clearly, the error probability can be viewed as a measure of the practicality of designed algorithms, and we will see that the main point of the classification is not only based on measuring the absolute size of the error probability, but is related mainly to the number of repetitions of independent runs of the algorithms on the same input necessary and sufficient in order to essentially reduce the probability of a wrong output. In other words, this classification focuses on the speed of the error probability reduction with the number of repeated runs.

[31] Usually the length of these random sequences grows with the input length, and so the number of all possible (over all inputs) random sequences in unbounded.

The classification of randomized algorithms is not the same for all different computing tasks. The meaning of a "wrong output" may vary under different circumstances. For solving a decision problem or for computing a function, the error probability is really the probability of computing a wrong result. On the other hand, when designing algorithms for optimization problems, it is not always reasonable to consider a feasible non-optimal solution as an unwanted result or even as an error. A feasible solution whose quality is not too far from that of an optimal solution can be very much appreciated.

Here we start with the standard classification of randomized algorithms for decision problems and for computing functions. After that we present the classification of randomized algorithms for optimization problems separately in Section 2.5.

LAS VEGAS ALGORITHMS

Las Vegas algorithms are randomized algorithms that guarantee that every computed output is correct. This means that wrong results (outputs) are forbidden in this model. In the literature one can find two different models of Las Vegas algorithms. These models differ in whether or not the answer "?" with the meaning "I do not know" ("In this run I was unable to compute the right solution") is allowed.

First, let us consider the case that the answer "?" is not allowed. In what follows, $\boldsymbol{A(x)}$ always denotes the output of the algorithm A on a given input x.

Definition 2.4.41. *A randomized algorithm A is called a* **Las Vegas algorithm** *computing a function F if, for any input x (any argument x of F),*

$$\text{Prob}(A(x) = F(x)) = 1.$$

For Las Vegas algorithms according to Definition 2.4.41, we always investigate the expected complexity (for instance, the expected time complexity $\text{Exp-Time}_A(n)$). Here, one has to expect that the runs of the algorithm on an input are of different lengths, because if all runs of the algorithm on any given input would have approximately the same length, then one could construct an equally efficient deterministic algorithm for this task, that simply simulates one fixed run[32] of the randomized algorithm.

Exemplary illustrations of this concept of Las Vegas algorithms are the randomized Quicksort and the randomized algorithm RSEL that were presented in the previous section. In both these algorithms we have essential differences between the worst-case complexity and the expected complexity. But the crucial point is that the expected complexity is very close to the complexity of the most efficient runs.

Now, let us consider the second way of defining Las Vegas algorithms.

[32]It does matter which one, because all runs are efficient.

Definition 2.4.42. *Let A be a randomized algorithm that allows the answer "?". We say that A is a* **Las Vegas algorithm** *for a function F if, for every input x,*

(i) $\mathrm{Prob}(A(x) = F(x)) \geq 1/2$, *and*
(ii) $\mathrm{Prob}(A(x) = \text{“?”}) = 1 - \mathrm{Prob}(A(x) = F(x)) \leq 1/2$.

We observe that the condition (ii) excludes the occurrence of a wrong output, i.e., one gets either the right output or the output "?". In condition (i) we require that a Las Vegas algorithm computes the correct value $F(x)$ with a probability greater than or equal to 1/2. The constant 1/2 is not crucial for this definition, and can be exchanged for an arbitrary ε satisfying $0 < \varepsilon < 1$. The reason for this is that for every $\varepsilon \leq 1/2$ one can increase the success probability (the probability of computing $F(x)$) to greater than 1/2 by executing a constant many independent runs of the algorithm on x.

Exercise 2.4.43. Let ε, δ be real numbers, $0 < \varepsilon < \delta < 1$. Let A be a randomized algorithm that computes a function F with

$$\mathrm{Prob}(A(x) = F(x)) \geq \varepsilon \text{ and } \mathrm{Prob}(A(x) = \text{“?”}) = 1 - \mathrm{Prob}(A(x) = F(x)) .$$

Let, for every $k \in \mathbb{N}$, $k \geq 2$, A_k be the randomized algorithm that for any input x executes k independent runs of A on x. The output of A_k is "?" if and only if all k runs of A on x finished with the output "?". In all other cases the algorithm A_k computes the right result $F(x)$. Estimate the smallest k such that $\mathrm{Prob}(A_k(x) = F(x)) \geq \delta$.

In what follows, we present a Las Vegas algorithm allowing the output "?", and then we compare these two models of Las Vegas algorithms.

Example 2.4.44. We consider again the model of communication protocols. We have two computers R_{I} and R_{II}. The input of R_{I} consists of ten strings, $x_1, x_2, \ldots, x_{10} \in \{0,1\}^n$, and R_{II} also has ten strings, $y_1, y_2, \ldots, y_{10} \in \{0,1\}^n$. The task is to estimate whether there is a $j \in \{1, \ldots, 10\}$ such that

$$x_j = y_j .$$

If such a j exists, then the protocol has to accept the input $((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10}))$, and if not, the input has to be rejected. As usual, the complexity is measured in the number of exchanged bits. One can prove that every deterministic protocol solving this task must allow a communication of $10n$ bits. Hence, no protocol can do better than to send the whole input of R_{I} of $10n$ bits to R_{II}, and then let R_{II} to perform the comparison.

Now, we design a Las Vegas protocol that solves the task with communication complexity of $n + O(\log n)$. We observe that the (communication) protocol from Section 1.3 is not a Las Vegas protocol. It is true that we may use the idea of generating a prime p at random, but we are required[33] to verify the correctness of the hypothesis proposed by the equality of the remainders modulo p.

[33]because the answer of a Las Vegas protocol must be correct with certainty

Protocol LV_{10}

Initial situation: R_{I} has ten strings, $x_1, x_2, \ldots, x_{10}$, $x_i \in \{0,1\}^n$ for all $i = 1, \ldots, 10$, and R_{II} has ten strings, $y_1, y_2, \ldots, y_{10}$, $y_i \in \{0,1\}^n$ for all $i = 1, \ldots, 10$.

Phase 1: R_{I} uniformly chooses 10 primes, $p_1, p_2, \ldots, p_{10}$, from $\mathrm{PRIM}\left(n^2\right)$ at random.

Phase 2: R_{I} computes

$$s_i = \mathrm{Number}(x_i) \bmod p_i$$

for $i = 1, 2, \ldots, 10$ and sends

$$p_1, p_2, \ldots, p_{10}, s_1, s_2, \ldots, s_{10}$$

to R_{II}.

Phase 3: R_{II} computes

$$q_i = \mathrm{Number}(y_i) \bmod p_i$$

for $i = 1, 2, \ldots, 10$, and computes s_i and q_i for all $i \in \{1, 2, \ldots, 10\}$.
If $s_i \neq q_i$ for all $i \in \{1, 2, \ldots, 10\}$, then R_{II} knows with certainty that $x_i \neq y_i$ for all $i \in \{1, 2, \ldots, 10\}$ and outputs "0" ("reject").
Else, let j be the smallest integer from $\{1, \ldots, 10\}$, such that

$$s_j = q_j .$$

Then R_{II} sends the whole string y_j and j to R_{I}.

Phase 4 R_{I} compares x_j with y_j bit by bit.
If $x_j = y_j$, then R_{I} outputs "1" ("accept").
If $x_j \neq y_j$, then R_{I} outputs "?".
{Observe that R_{I} really does not known the right answer in this case, because there can exist a $k > j$ such that $s_k = q_k$. But the protocol does not try to check whether $x_k = y_k$ in such a case.}

First, we analyze the worst-case complexity of LV_{10}. The maximal possible communication consists of sending

$$p_1, p_2, \ldots, p_{10}, s_1, s_2, \ldots, s_{10}$$

from R_{I} to R_{II}, and of sending

$$y_j \text{ and } j$$

from R_{II} to R_{I}.

All the numbers $p_1, p_2, \ldots, p_{10}, s_1, s_2, \ldots, s_{10}$ are smaller than n^2, and so every of them can be represented by $2\lceil \log_2 n \rceil$ bits. The integer j can be represented by 4 bits, and sending the whole y_j costs n bits. Altogether, the communication complexity is

$$n + 40\lceil \log_2 n \rceil + 4 = n + O(\log n) .$$

Now we will show that LV_{10} is a Las Vegas protocol. We distinguish two cases with respect whether the input has to be accepted or rejected.

(i) Let $((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10}))$ be an input with $x_i \neq y_i$ for all $i \in \{1, \ldots, 10\}$.
We have already learned[34] that, for every $i \in \{1, \ldots, 10\}$, the probability that a randomly chosen $p_i \in \text{PRIM}\left(n^2\right)$ satisfies

$$\text{Number}(x_i) \mod p_i \neq \text{Number}(y_i) \mod p_i$$

(i.e., the probability that the claim $x_i \neq y_i$ was justified by p_i) is at least

$$1 - \frac{2 \ln n}{n}.$$

Since $p_1, p_2, \ldots, p_{10}$ were chosen independently, the probability that

$$\text{Number}(x_i) \mod p_i \neq \text{Number}(y_i) \mod p_i$$

for all $i \in \{1, \ldots, 10\}$ is at least

$$\left(1 - \frac{2 \ln n}{n}\right)^{10}.$$

Hence,

$$\begin{aligned}
&\text{Prob}\Big(\text{LV}_{10}\left((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10})\right) = 0\Big) \\
&\quad \geq \left(1 - \frac{2 \ln n}{n}\right)^{10} \\
&\quad = \sum_{i=0}^{10} (-1)^i \cdot \binom{10}{i} \cdot \left(\frac{2 \ln n}{n}\right)^i \\
&\quad = 1 + \sum_{l=1}^{5} \binom{10}{2l} \cdot \left(\frac{2 \ln n}{n}\right)^{2l} - \sum_{m=1}^{5} \binom{10}{2m-1} \cdot \left(\frac{2 \ln n}{n}\right)^{2m-1} \\
&\qquad \{\text{the sum was partitioned into positive and negative elements}\} \\
&\quad = 1 + \sum_{i=1}^{5} \binom{10}{2i} \cdot \left(\frac{2 \ln n}{n}\right)^{2i} - \sum_{i=0}^{4} \binom{10}{2i+1} \cdot \left(\frac{2 \ln n}{n}\right)^{2i+1}. \qquad (2.6)
\end{aligned}$$

For all $i = 1, 2, 3, 4$ and all $n \geq 15$,

$$\binom{10}{2i} \cdot \left(\frac{2 \ln n}{n}\right)^{2i} \geq \binom{10}{2i+1} \cdot \left(\frac{2 \ln n}{n}\right)^{2i+1}. \qquad (2.7)$$

Omitting the positive numbers

[34] in Section 1.3

$$\binom{10}{2i} \cdot \left(\frac{2\ln n}{n}\right)^{2i} - \binom{10}{2i+1} \cdot \left(\frac{2\ln n}{n}\right)^{2i+1}$$

from (2.6) for $i = 1, 2, 3, 4$, one obtains

$$\begin{aligned}
&\text{Prob}\Big(\text{LV}_{10}\ ((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10})) = 0\Big)\\
&\quad \geq 1 - \binom{10}{1} \cdot \frac{2\ln n}{n} + \binom{10}{10} \cdot \left(\frac{2\ln n}{n}\right)^{10}\\
&\quad \geq 1 - \frac{20\ln n}{n} \geq \frac{1}{2}
\end{aligned}$$

for sufficiently large[35] n.

In the complementary case, when there exists a j such that

$$\text{Number}(x_j) \mod p_j = \text{Number}(y_j) \mod p_j,$$

the computation ends with the answer "?", because $x_j \neq y_j$ (i.e., the protocol cannot confirm the hypothesis "$x_j = y_j$" in Phase 4).

Thus, for all inputs that have to be rejected, the conditions (i) and (ii) of Definition 2.4.42 are satisfied, and so the protocol LV_{10} behaves as a Las Vegas protocol on inputs that have to be rejected.

(ii) Let $((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10}))$ be an input that has to be accepted, and let j be the smallest number from $\{1, \ldots, 10\}$ such that

$$x_j = y_j.$$

Obviously,

$$\text{Number}(x_j) \mod p_j = \text{Number}(y_j) \mod p_j$$

for all primes p, and so for p_j, too. Therefore, LV_{10} compares the strings x_j and y_j in Phase 4 and accepts the input if and only if[36]

$$\text{Number}(x_i) \mod p_i \neq \text{Number}(y_i) \mod p_i$$

for all $i \in \{1, 2 \ldots, j-1\}$. We denote this positive event by E_j. If $j = 1$, it is obvious that the input is accepted by LV_{10} with certainty. Now, consider the case $j > 1$. In part (i) of our analysis, we have already proved that the probability of the event E_j is at least

$$\left(1 - \frac{2\ln n}{n}\right)^{j-1} \geq 1 - \frac{2(j-1) \cdot \ln n}{n}$$

[35] In fact, the probability of getting the correct answer tends to 1 with growing n, and this is more than what one requires in the definition of Las Vegas algorithms.

[36] Remember that LV_{10} compares the substrings x_j and y_j if and only if $s_j = q_j$ and j is the smallest number with this property.

for sufficiently large n. This expression has its minimum for $j = 10$, and so

$$\text{Prob}\Big(\text{LV}_{10}\left((x_1, \ldots, x_{10}), (y_1, \ldots, y_{10})\right) = 1\Big) \geq 1 - \frac{18 \ln n}{n},$$

which is larger than $1/2$ for all $n \geq 189$.
In the complementary case, when there exists an $l < j$ such that

$$\text{Number}(x_l) \mod p_l = \text{Number}(y_l) \mod p_l,$$

the protocol LV_{10} ends with the output "?", because $x_l \neq y_l$ is fixed[37] in Phase 4.

In this way we have proved that LV_{10} is a Las Vegas protocol. □

Exercise 2.4.45. Modify the Las Vegas protocol LV_{10} in such a way that instead of choosing ten primes $p_1, \ldots, p_{10}$ at random one chooses only one prime $p \in \text{PRIM}\left(n^2\right)$ at random. Then, one computes the remainders $s_1, \ldots, s_{10}$, $q_1, \ldots, q_{10}$ as

$$s_i = \text{Number}(x_i) \mod p \text{ and } q_i = \text{Number}(y_i) \mod p.$$

In this way, the protocol saves $18 \cdot \lceil \log_2 n \rceil$ communication bits. Explain why our analysis of the success probability of LV_{10} works without any change also for the modified protocol, despite the exchange of $p_1, \ldots, p_{10}$ for an only one prime p.

If one achieves the appreciated situation (as in Example 2.4.44) that a Las Vegas algorithm A computes the right result $F(x)$ with a probability tending to 1 with growing $|x|$, then one speaks of **Las Vegas* algorithms**.

We defined above two models of Las Vegas algorithms. Do they essentially differ? Is it possible to transform a Las Vegas algorithm with respect to Definition 2.4.41 into a Las Vegas algorithm with respect to Definition 2.4.42, and vice versa? These are the main questions we are dealing with in what follows.

First, we show how to convert the model where the answer "?" is allowed into the model where all outputs must be the correct results. Let A be a Las Vegas algorithm that is allowed to output "?" with a bounded probability. We modify A to an algorithm A' by starting a new run on the same input whenever A would output "?". This strategy is depicted in Figure 2.4. If $T_{A,w}$ is the computation tree of A on w, then one obtains the computation tree $T_{A',w}$ of all computations of A' on w by hanging the tree $T_{A,w}$ at all leaves of $T_{A,w}$ with the output "?", etc. The drawback of this transformation is that the new algorithm A' contains infinite computations.[38] On the other hand, we have the assurance that the algorithm A' ends with the correct result if it halts.

[37]We assume that l is the smallest integer from $\{1, 2, \ldots, 10\}$, such that $x_l \equiv y_l \pmod{p_l}$.

[38]One does not know of any conversion of Las Vegas algorithms with output "?" into Las Vegas algorithms without the output "?" that would avoid infinite runs.

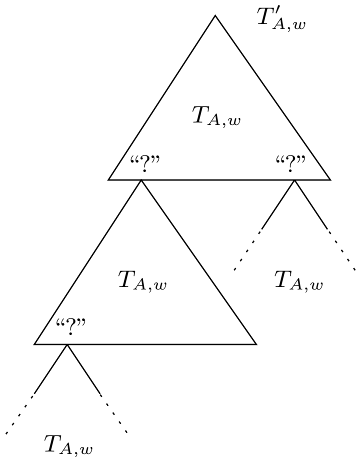


Fig. 2.4.

How high is the probability that A' halts? This probability is high because it tends to 1 with growing running time. Why is it so? Let $\text{Time}_A(w)$ be the worst-case complexity of A, i.e., the depth of $T_{A,w}$. This means that the probability that A' stops with outputting the right result in time $\text{Time}_A(w)$ is at least 1/2. The probability of successfully finishing the work in $2 \cdot \text{Time}_A(w)$ is already at least 3/4, because A' starts new runs of A on w for every leaf of $T_{A,w}$ with the output "?". This means that after time $k \cdot \text{Time}_A(w)$ the algorithm A' computes the correct result with probability at least

$$1 - \frac{1}{2^k},$$

because 2^{-k} is an upper bound on the probability of getting the output "?" in k independent runs of A on w.

How large is the expected value of the time complexity $\text{Exp-Time}_{A'}(n)$ of A'? We claim that

$$\text{Exp-Time}_{A'}(n) \in O(\text{Time}_A(n)) .$$

In what follows, we prove this claim. Without loss of generality one may assume that all computations of A on w with the output "?" have the maximal length $\text{Time}_A(w)$, and so all these computations have the same length. Let, for all $i \in \mathbb{N} - \{0\}$,

$$\text{Set}_i = \{C \in S_{A',w} \mid (i-1) \cdot \text{Time}_A(w) < \text{Time}(C) \leq i \cdot \text{Time}_A(w)\}$$

be the set of all computations that end (halt) exactly[39] in the ith run of A. Clearly, $S_{A',w} = \bigcup_{i=1}^{\infty} \text{Set}_i$ and $\text{Set}_r \cap \text{Set}_s = \emptyset$ for $r \neq s$. Above, we have

[39]those computations of A' that contain $(i-1)$ starts of A on w

already observed that

$$\sum_{C \in \bigcup_{i=1}^{k} \mathrm{Set}_i} \mathrm{Prob}(\{C\}) \geq 1 - \frac{1}{2^k}$$

for all $k \in \mathbb{N} - \{0\}$. A direct consequence of this fact is that

$$\sum_{C \in \mathrm{Set}_i} \mathrm{Prob}(\{C\}) \leq \frac{1}{2^{i-1}} \tag{2.8}$$

for all integers $i \geq 1$.

Thus, we obtain

$$\begin{aligned}
\text{Exp-Time}_{A'}(n) &\underset{def.}{=} \sum_{C \in S_{A',w}} \mathrm{Time}_{A'}(C) \cdot \mathrm{Prob}(\{C\}) \\
&= \sum_{i=1}^{\infty} \sum_{C \in \mathrm{Set}_i} \mathrm{Time}_{A'}(C) \cdot \mathrm{Prob}(\{C\}) \\
&\leq \sum_{i=1}^{\infty} \sum_{C \in \mathrm{Set}_i} i \cdot \mathrm{Time}_A(w) \cdot \mathrm{Prob}(\{C\}) \\
&\quad \{\text{since } \mathrm{Time}_{A'}(C) \leq i \cdot \mathrm{Time}_A(w) \\
&\quad\ \text{for every } C \in \mathrm{Set}_i\} \\
&= \sum_{i=1}^{\infty} i \cdot \mathrm{Time}_A(w) \cdot \sum_{C \in \mathrm{Set}_i} \mathrm{Prob}(\{C\}) \\
&\underset{(2.8)}{\leq} \sum_{i=1}^{\infty} i \cdot \mathrm{Time}_A(w) \cdot \frac{1}{2^{i-1}} \\
&= \mathrm{Time}_A(w) \cdot \sum_{i=1}^{\infty} \frac{i}{2^{i-1}} \\
&= \mathrm{Time}_A(w) \cdot 2 \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} \\
&= 2 \cdot \mathrm{Time}_A(w) \cdot \left(\frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{4}{16} + \sum_{i=5}^{\infty} \frac{i}{2^i} \right) \\
&< 6 \cdot \mathrm{Time}_A(w) \\
&\quad \{\text{since } \textstyle\sum_{i=5}^{\infty} \frac{i}{2^i} < 1\} \quad .
\end{aligned}$$

Exercise 2.4.46. Use the method described above to modify the algorithm LV_{10} from Example 2.4.44 in such a way[40] that the output "?" does not occur. Then, analyze the expected communication complexity of your Las Vegas protocol with forbidden "?".

[40] In the way described above, for instance.

Exercise 2.4.47. Modify the algorithm LV_{10} from Example 2.4.44 as follows. The first three phases remain unchanged. If in Phase 4 R_{I} fixes[41] that $x_j \neq y_j$, then (instead of finishing with "?") it sends this knowledge[42] to R_{II}. Now R_{II} looks for an $l > j$, such that $s_l = q_l$. If such an l does not exist, then R_{II} rejects the input (outputs "0") with certainty. If such an l exists, then R_{II} sends the whole string y_l to R_{I} which compares y_l and x_l. If $x_l = y_l$, then R_{I} accepts the input. If $x_l \neq y_l$, then R_{I} sends this information to R_{II}, and R_{II} looks again for another candidate for the equality, and so on. In the worst-case the protocol compares all pairs (x_m, y_m), with $s_m = q_m$. All computations of this algorithm are finite, and it is obvious that this algorithm computes the correct answer in every computation. Analyze the expected communication complexity of this protocol.

We observe that Las Vegas algorithms with respect to the first model (Definition 2.4.41) are a special case of Las Vegas algorithms with respect to the second model (Definition 2.4.42). Despite this, there are reasons to try to convert a Las Vegas algorithm always providing the correct result into a Las Vegas algorithm that may output "?". A situation of this kind is depicted in Figure 2.5. One considers a computation tree $T_{A,w}$ that contains many short (efficient) computations, but also a few relatively long (possibly infinite) ones. For instance, the short computations run in linear time, and the long ones in cubic time. In the case when a computation runs longer than the length of the efficient computations (i.e., when one may consider that the running computation is one of the very long computations), one can decide to stop this computation and to output "?". What bound on time complexity for stopping a computation can one take without losing the guarantee that the resulting algorithm is a Las Vegas algorithm with respect to Definition 2.4.42?

Our goal is to show that

$$2 \cdot \text{Exp-Time}_A(w)$$

is a sufficient upper bound for stopping the work of the algorithm. We prove this by contradiction. Assume

$$\mathrm{Prob}(B(w) = \text{“?”}) > \frac{1}{2}. \tag{2.9}$$

Setting

$$S_{A,w}(\text{“?”}) = \{C \in S_{A,w} \mid \mathrm{Time}(C) > 2 \cdot \text{Exp-Time}_A(w)\} \subset S_{A,w},$$

one can reformulate (2.9) as

$$\sum_{C \in S_{A,w}(\text{“?”})} \mathrm{Prob}(\{C\}) > \frac{1}{2}. \tag{2.10}$$

[41]recognizes

[42]One bit is enough for this.

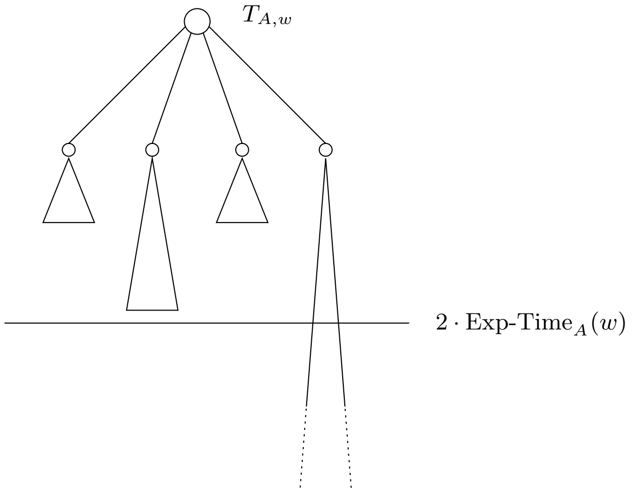


Fig. 2.5.

Set

$$S_{A,w}(F(w)) = S_{A,w} - S_{A,w}(\text{“?”}).$$

Following the definition of $S_{A,w}(\text{“?”})$, we have

$$\text{Time}(C) \geq 2 \cdot \text{Exp-Time}_A(w) + 1 \text{ for all } C \in S_{A,w}((\text{“?”}). \qquad (2.11)$$

Hence,

$$\begin{aligned}
\text{Exp-Time}_A(w) &= \sum_{C \in S_{A,w}} \text{Time}_A(C) \cdot \text{Prob}(\{C\}) \\
&= \sum_{C \in S_{A,w}(\text{“?”})} \text{Time}_A(C) \cdot \text{Prob}(\{C\}) \\
&\quad + \sum_{C \in S_{A,w}(F(w))} \text{Time}_A(C) \cdot \text{Prob}(\{C\}) \\
&\underset{(2.11)}{>} \sum_{C \in S_{A,w}(\text{“?”})} (2 \cdot \text{Exp-Time}_A(w) + 1) \cdot \text{Prob}(\{C\}) + 0 \\
&= (2 \cdot \text{Exp-Time}_A(w) + 1) \cdot \sum_{C \in S_{A,w}(\text{“?”})} \text{Prob}(\{C\}) \\
&\underset{(2.10)}{>} (2 \cdot \text{Exp-Time}_A(w) + 1) \cdot \frac{1}{2} = \text{Exp-Time}_A(w) + \frac{1}{2}.
\end{aligned}$$

In this way we have obtained $\text{Exp-Time}_A(w) > \text{Exp-Time}_A(w) + \frac{1}{2}$, which is obviously a contradiction. Hence, the negation of (2.9) (and also of (2.10)) is true.

The idea behind the proof above is the simple combinatorial fact saying that it is impossible that more than half the elements of a subset of $\mathbb{N}$ are larger than the doubled average.[43]

Exercise 2.4.48. Modify the randomized Quicksort by stopping all computations longer than $16 \cdot n \cdot \log_2 n$ with the output "?". Give an upper bound on the probability of the output "?".

Exercise 2.4.49. Consider the randomized algorithm RSEL(A, k) from Exercise 2.3.40. What upper bound on the computation length is sufficient for assuring the probability of getting the output "?" below 2^{-k} for a positive integer k? Consider the case when one stops every computation running longer than $6 \cdot n \cdot \log_2 n$ steps, and outputs "?". Is the resulting algorithm a Las Vegas* algorithm with respect to Definition 2.4.42?

Exercise 2.4.50. Consider the following communication task for the computers R_{I} and R_{II}. R_{I} has $2n$ bits, $x_1, x_2, \ldots, x_{2n}$, and R_{II} has an integer $j \in \{1, \ldots, 2n\}$. R_{I} does not know any bit of the input of R_{II}, and R_{II} does not have any information about the input of R_{I}. We allow the submission of one message from R_{I} to R_{II} only,[44] and after that we require that R_{II} provides the bit x_j as its output. The complexity of such a protocol is the length of the only binary message communicated.

(i) Prove that every deterministic protocol for this task must allow messages of length $2n$.
(ii) Does there exist a Las Vegas protocol that is allowed to output "?" and that solves this task within complexity $n + 1$?

ONE-SIDED-ERROR MONTE CARLO ALGORITHMS

This type of randomized algorithms is considered for decision problems only. Usually a decision problem is given by a pair (Σ, L) where, for every given string $x \in \Sigma^*$, one has to decide whether or not x is in the language $L \subseteq \Sigma^*$. The idea here is to allow errors only for inputs from L. This means that, for every $x \in \Sigma^* - L$, one requires that a one-sided-error randomized algorithm outputs the correct answer "0" ("$x \notin L$") with certainty, and that only for inputs from L may the algorithm err with a bounded probability. This concept of one-sided-error algorithms can be formally expressed as follows.

Definition 2.4.51. *Let A be a randomized algorithm and let (Σ, L) be a decision problem. We say that A is an* **one-sided-error Monte Carlo algorithm** *for L,* **1MC algorithm** *for short, if*[45]

[43] If more than half the numbers are larger than $2 \cdot \mathit{average} + \varepsilon$ for an arbitrary small $\varepsilon > 0$, then one cannot compensate for this contribution to the average, not even by setting all the remainding numbers to 0.

[44] Protocols with this property are called one-way protocols.

[45] Remember, that the output 1 means acceptance (yes) and the output 0 represents rejection (no).

(i) for every $x \in L$, $\text{Prob}(A(x) = 1) \geq \frac{1}{2}$, *and*
(ii) for every $x \notin L$, $\text{Prob}(A(x) = 0) = 1$.

The exemplary randomized protocol R from Section 1.2 is an appropriate illustration of Definition 2.4.51. If $\Sigma = \{0, 1\}$ and

$$L = L_{\text{unequal}} = \{(x, y) \mid x, y \in \{0, 1\}^n, x \neq y, n \in \mathbb{N}\},$$

the designed protocol R accepts the language L_{unequal}. If $x = y$ (i.e., when $(x, y) \notin L_{\text{unequal}}$), then the protocol R outputs[46] "equal" (i.e., "$(x, y) \notin L_{\text{unequal}}$") with certainty. If $x \neq y$, then R accepts (x, y) by outputting "$x \neq y$" with a probability at least $1 - \frac{2 \cdot \ln n}{n}$.

Similarly, as in the case of Las Vegas algorithms, we introduce the *-notation. A 1MC algorithm is a 1MC* algorithm, when

(i') for every $x \in L$, $\text{Prob}(A(x) = 1)$ tends to 1 with growing $|x|$.

Obviously, the protocol R for L_{unequal} is a 1MC* algorithm.

The reason to consider the 1MC algorithms as the most practical ones after Las Vegas algorithms is that their error probability is exponentially decreasing with the number of computation repetitions (independent runs on the same input).

Let us explain this in detail. Following condition (i) of Definition 2.4.51, for every $x \in L$, the error probability of any 1MC algorithm for L is at most 1/2. Following condition (ii) of Definition 2.4.51, for every $y \notin L$, the error probability is equal to 0. Let

$$\alpha_1, \alpha_2, \ldots, \alpha_k,\ \alpha_i \in \{0, 1\} \text{ for } i = 1, \ldots, k,$$

be the k outputs of k independent runs of a 1MC algorithm A on an input z. If there exists a $j \in \{1, \ldots, k\}$ such that

$$\alpha_j = 1,$$

then we know with certainty[47] that $x \in L$. Only if

$$\alpha_1 = \alpha_2 = \ldots = \alpha_k = 0,$$

(i.e., in the complementary case) one takes the answer "0". This output can be wrong when $x \in L$. Since, for every $z \in L$,

$$\text{Prob}(A(z) = 0) \leq \frac{1}{2},$$

we obtain that the probability of having $A(z) = 0$ in all k independent runs of A on z is

[46] rejects

[47] From condition (ii) of Definition 2.4.51, a 1MC algorithm can never output "1" ("accept") for an input $z \notin L$.

$$(\text{Prob}(A(z)=0))^k \le \left(\frac{1}{2}\right)^k = 2^{-k}.$$

Hence, we see why 1MC algorithms are very much appreciated in many applications. Executing k repetitions of the work of a 1MC algorithm A on the same input

(i) the complexity grows only k times (i.e., in a linear way), and
(ii) the error probability tends to 0 with exponential speed in k.

In what follows, we denote by $\boldsymbol{A_k}$ the 1MC algorithm that consists of k independent runs of a 1MC algorithm A on any input, and that accepts an input if and only if at least one of the runs accepts this input. We keep in our mind that the error probability of $\boldsymbol{A_k}$ is the error probability of A to k, and so A_k is a 1MC algorithm if A is a 1MC algorithm.

BOUNDED-ERROR MONTE CARLO ALGORITHMS

This large class of randomized algorithms is defined in such a way that

(i) a constant[48] number of independent computation repetitions on the same input is always sufficient for reducing the error probability below a given constant δ, and
(ii) any relaxation of the requirement of the following definition of bounded-error Monte Carlo algorithms can cause a situation in which the execution of polynomially[49] many computations on the same input does not suffice to reduce the error probability below a given constant δ.

Definition 2.4.52. *Let F be a function. We say that a randomized algorithm A is a* **bounded-error Monte Carlo algorithm for $\boldsymbol{F}$**, **2MC algorithm**[50] *for short, if*

there exists a real number ε, $0 < \varepsilon \le 1/2$ such that, for every input x of F,

$$\text{Prob}(A(x) = F(x)) \ge \frac{1}{2} + \varepsilon. \tag{2.12}$$

We call attention to the fact that the upper bound $\frac{1}{2} - \varepsilon$ on the error probability determined by a fixed $\varepsilon > 0$ independent of inputs is the core of Definition 2.4.52. Only this guaranted distance ε to the probability 1/2 for any input assures the existence of an efficient way for reducing the error probability to an arbitrarily small given δ. Note that the requirement (2.12) does not hinder us to viewing Las Vegas algorithms and one-sided-error Monte

[48]with respect to the input length, but not with respect to δ

[49]in the input size

[50]the notation 2MC comes from two-sided-error Monte Carlo, which is an alternative name of bounded-error Monte Carlo in the literature.

Carlo algorithms as special cases of bounded-error Monte Carlo algorithms. If A is a 1MC algorithm (or a Las Vegas algorithm), then the algorithm A_2 computes the correct result with a probability at least 3/4, and so A_2 satisfies the requirement (2.12).

In this sense one can view all up till now presented randomized algorithms as bounded-error Monte Carlo algorithms. We note that the randomized protocol from our motivation example in Section 1.2 is a 1MC* algorithm for L_{unequal}, but not a 1MC algorithm for the complementary language

$$L = L_{\text{equal}} = \{(x, y) \mid x, y \in \{0, 1\}^*, x = y\}.$$

On the other hand, it is obvious that the randomized protocol R is a 2MC algorithm for L_{equal}.

Our next goal is to analyze the speed of reduction of the error probability of 2MC algorithms with respect to the number of computation repetitions on the same input. For any 2MC algorithm A and any positive integer t, let A_t denote the following randomized algorithm:

2MC Algorithm A_t

```
Input: $x$
Step 1: Perform $t$ independent runs of $A$ on $x$ and save the $t$ computed results

            $\alpha_1, \alpha_2, \ldots, \alpha_t$.

Step 2: if there is an $\alpha$ that appears at least $\lceil \frac{t}{2} \rceil$ times in the sequence
    $\alpha_1, \alpha_2, \ldots, \alpha_t$,
    then
        output "$\alpha$"
    else[51]
        output "?"
```

Let $\varepsilon > 0$ be a constant, such that, for all feasible inputs x,

$$\text{Prob}(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon.$$

For every input x, let

$$p = p(x) = \text{Prob}(A(x) = F(x)) = \frac{1}{2} + \varepsilon_x \text{ for an } \varepsilon_x \geq \varepsilon$$

be a short notation for the success probability of A on x.

Clearly, A_t computes a wrong result or "?" only if the correct result was not computed at least $\lceil t/2 \rceil$ times in the t independent runs of A on x.

[51] A reasonable alternative way of determining the output of A_t would be to take the most frequently occurring result in $\alpha_1, \alpha_2, \ldots, \alpha_t$ as the final output. The following analysis works also for this possibility.

In what follows, we analyze, for all $i < \lceil t/2 \rceil$, the probability $\mathrm{pr}_i(x)$ that A_t computes the correct result in exactly i runs. In order to bound the error probability of A_t we are interested in deriving good upper bounds on $\mathrm{pr}_i(x)$ for all $i < \lceil t/2 \rceil$. The following holds:

$$\mathrm{pr}_i(x) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i}$$

{There are exactly $\binom{t}{i}$ ways of estimating the i positions with the correct result $F(x)$ in the sequence of t results. For each position, $p = p(x)$ is the probability of computing the correct result $F(x)$ and $1-p$ is the probability of computing a wrong result. Then p^i is the probability of computing the correct result exactly i times on i fixed positions, and $(1-p)^{t-i}$ is the probability of computing a wrong result on all $t-i$ remaining positions.}

$$= \binom{t}{i} \cdot (p \cdot (1-p))^i \cdot (1-p)^{t-2i}$$

{Since $t \geq 2i$, we have, $(1-p)^{t-i} = (1-p)^i \cdot (1-p)^{t-2i}$}

$$= \binom{t}{i} \cdot \left(\left(\frac{1}{2} + \varepsilon_x\right) \cdot \left(\frac{1}{2} - \varepsilon_x\right) \right)^i \cdot \left(\frac{1}{2} - \varepsilon_x\right)^{2 \cdot (\frac{t}{2} - i)}$$

{Since $p = \frac{1}{2} + \varepsilon_x$ and $1 - p = \frac{1}{2} - \varepsilon_x$}

$$= \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^i \cdot \left(\left(\frac{1}{2} - \varepsilon_x\right)^2 \right)^{\frac{t}{2} - i}$$

$$< \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^i \cdot \left(\left(\frac{1}{2} - \varepsilon_x\right) \cdot \left(\frac{1}{2} + \varepsilon_x\right) \right)^{\frac{t}{2} - i}$$

{Since $\frac{1}{2} - \varepsilon_x < \frac{1}{2} + \varepsilon_x$}

$$= \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^i \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^{\frac{t}{2} - i}$$

$$= \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^{\frac{t}{2}}$$

$$\leq \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon^2\right)^{\frac{t}{2}}$$

{Since $\varepsilon_x \geq \varepsilon$ for every input x}.

Since A_t computes the correct result $F(x)$ on x if and only if at least $\lceil t/2 \rceil$ runs of A have computed the correct result $F(x)$, one obtains for every input x the following lower bound on the success probability of A_t:

$$\mathrm{Prob}(A_t(x) = F(x)) = 1 - \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \mathrm{pr}_i(x)$$

$$\begin{aligned}
&> 1 - \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \binom{t}{i} \cdot \left(\frac{1}{4} - \varepsilon_x^2\right)^{\frac{t}{2}} \\
&= 1 - \left(\frac{1}{4} - \varepsilon_x^2\right)^{\frac{t}{2}} \cdot \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} \binom{t}{i} \\
&> 1 - \left(\frac{1}{4} - \varepsilon_x^2\right)^{\frac{t}{2}} \cdot 2^t \\
&\geq 1 - \left(1 - 4 \cdot \varepsilon_x^2\right)^{\frac{t}{2}} \\
&\geq 1 - \left(1 - 4 \cdot \varepsilon^2\right)^{\frac{t}{2}} .
\end{aligned}$$

We observe that the upper bound

$$(1 - 4 \cdot \varepsilon^2)^{t/2}$$

on the error probability of A_t tends to 0 with growing t. Thus, if one looks for a k such that

$$\mathrm{Prob}(A_k(x) = F(x)) \geq 1 - \delta,$$

for a chosen constant δ, it is sufficient to take

$$k \geq \frac{2 \cdot \ln \delta}{\ln(1 - 4 \cdot \varepsilon^2)}. \tag{2.13}$$

Thus, if δ and ε are considered[52] fixed constants, then the number k of computation repetitions is also a constant (with respect to the inputs). Hence, we can conclude

$$\mathrm{Time}_{A_k}(n) \in O(\mathrm{Time}_A(n)) .$$

A consequence of this observation is that if A is asymptotically faster than any deterministic algorithm computing F, then A_k with an error probability below a chosen δ is also more efficient than any deterministic algorithm.

Exercise 2.4.53. Modify the 2MC algorithm A_t to a randomized algorithm A'_t in such a way that A'_t takes the most frequent result as the output. What is the error probability of this modified algorithm A'_t?

UNBOUNDED-ERROR MONTE CARLO ALGORITHMS

Clearly, a randomized algorithm cannot be used for computing a function F when the error probability is not less than 1/2. If the error probability is at least 1/2, then a wrong result may be produced as frequently as the correct result, and even then the execution of several independent runs of the

[52]From Definition 2.4.52 of $2MC$ algorithms, ε is always a fixed constant with respect to inputs.

algorithm on the same input cannot help in recognizing which of the outputs is the correct result. Consequently, general randomized algorithms must require that error probability be less than 1/2.

Definition 2.4.54. *Let F be a function. We say that a randomized algorithm A is a* **(unbounded-error)**[53] **Monte Carlo algorithm** *computing F, an* **MC algorithm***, for short, if, for every input x of F,*

$$\text{Prob}(A(x) = F(x)) > 1/2.$$

The most interesting question for us now is:

What is the difference between bounded-error Monte Carlo algorithms and unbounded-error ones?

For 2MC algorithms, one forces the error probability to have a universal fixed distance from 1/2 for any input. For an MC algorithm, it may happen, that the distance between the error probability and 1/2 tends to 0 with growing input size $|x|$. For instance, a randomized algorithm A can have a uniform choice from $2^{|x|}$ deterministic strategies for any input x. If most of them (i.e., at least $2^{|x|-1} + 1$ many) provide the correct result, then A is an MC algorithm. Hence, one allows that

$$\text{Prob}(A(x) = F(x)) = \frac{1}{2} + \frac{1}{2^{|x|}} > \frac{1}{2}.$$

Thus, the distance $\varepsilon_x = \frac{1}{2^{|x|}}$ between the error probability and 1/2 tends to 0 with an exponential speed in $|x|$.

Now we pose the following principal question:

How many independent runs of A on x are necessary in order to get

$$\text{Prob}(A_k(x) = F(x)) > 1 - \delta$$

for a fixed chosen constant δ?

Since the analysis of the error probability of the 2MC algorithm A_t with respect to ε_x is valid without any change for MC algorithms, one has

$$\text{Prob}(A_t(x) = F(x)) \geq 1 - \left(1 - 4 \cdot \varepsilon_x^2\right)^{\frac{t}{2}}.$$

To achieve $\text{Prob}(A_k(x) = F(x)) \geq 1-\delta$, one obtains the following lower bound on the number of independent runs of A:

[53] The term "unbounded-error" is not used in the sense that the error probability is unbounded, but in the sense that there is no bound on error probability assuring a universal distance from 1/2 for every input.

$$\begin{aligned}
k = k(|x|) &\geq \frac{2 \cdot \ln \delta}{\ln(1 - 4 \cdot \varepsilon_x^2)} \\
&= \frac{2 \cdot \ln \delta}{\ln(1 - 4 \cdot 2^{-2 \cdot |x|})} \\
&\quad \{\text{Since } \varepsilon_x = 2^{-|x|}\} \\
&\geq \frac{2 \cdot \ln \delta}{-2^{-2 \cdot |x|}} \\
&\quad \{\text{Since } \ln(1 - y) \leq -y \text{ for } 0 < y < 1 \\
&\quad\ \ (\text{Lemma A.3.62})\} \\
&= (-2 \cdot \ln \delta) \cdot 2^{2 \cdot |x|}.
\end{aligned}$$

In this way, we obtain

$$\text{Time}_{A_k}(x) \geq (-2 \cdot \ln \delta) \cdot 2^{2 \cdot |x|} \cdot \text{Time}_A(x)\,.$$

Hence, the running time of A_k is exponential in input length $|x|$, even for the case where A is very fast.

Note that this consideration does not imply that no MC algorithm is applicable. For instance, if $\varepsilon_x = \frac{1}{\log_2 |x|}$, then few[54] (not exponentially many) repetitions of the computation on x are sufficient in order to get a useful algorithm for the computing task considered.

Exercise 2.4.55. Let A be an MC algorithm that, for every input x, computes the correct result $F(x)$ with probability $\frac{1}{2} + \varepsilon_x$, where ε_x depends on $|x|$. Let δ be a constant, $0 < \delta < 1/2$. How many repetitions $k = k(|x|)$ of the work of A on x are necessary to achieve $\text{Prob}(A_k(x) = F(x)) \geq 1 - \delta$, if

(i) $\varepsilon_x = \frac{1}{|x|}$,
(ii) $\varepsilon_x = \frac{1}{\log_2 |x|}$?

Exercise 2.4.56. Let A be a randomized algorithm computing a function F with $\text{Prob}(A(x) = F(x)) \geq 1/3$ for every argument x of F. Assume that one is aware of the fact that $\text{Prob}(A(x) = \alpha) \leq 1/4$ for every wrong result α (i.e., that the probability of computing any specific wrong result is at most 1/4). Can this knowledge be used to design a useful randomized algorithm for F?

Example 2.4.57. Consider the following randomized communication protocol between two computers R_{I} and R_{II} for recognizing the language L_{unequal}.

Protocol UMC

Initial situation: R_{I} has n bits $x = x_1 x_2 \ldots x_n$, R_{II} has n bits $y = y_1 y_2 \ldots y_n$, $n \in \mathbb{N} - \{0\}$.

[54] Obviously, a constant number of repetitions is not sufficient, but a low degree polynomial number of repetitions may be acceptable.

{This is the same initial situation as in our motivation example in Section 1.2 (Figure 1.1)}

The input (x, y) has to be accepted if and only if $x \neq y$.

Phase 1: R_{I} uniformly chooses a number $j \in \{1, 2, \ldots, n\}$ at random and sends

$$j \text{ and the bit } x_j$$

to R_{II}.

Phase 2: R_{II} compares x_j with y_j.

If $x_j \neq y_j$, R_{II} accepts the input (x, y).

{In this case, R_{II} is sure that $x \neq y$, i.e., that $(x, y) \in L_{\text{unequal}}$.}

If $x_j = y_j$, then R_{II} accepts (x, y) with probability $\frac{1}{2} - \frac{1}{2n}$, and rejects (x, y) with probability $\frac{1}{2} + \frac{1}{2n}$.

The communication complexity of UMC is exactly

$$\lceil \log_2(n+1) \rceil + 1$$

bits in each computation, and so UMC is always very efficient.

Now we show that UMC is a Monte Carlo protocol. As usual, we handle the two cases $(x, y) \in L_{\text{unequal}}$ and $(x, y) \notin L_{\text{unequal}}$ separately.

(i) Let $(x, y) \notin L_{\text{unequal}}$, i.e., $x = y$.
In this case UMC has exactly $2n$ computations C_{il} for $i \in \{1, 2, \ldots, n\}$ and $l \in \{0, 1\}$, where C_{il} is the computation in which R_{I} chooses the number i at random in Phase 1 and R_{II} outputs "l" (with the usual meaning, $l = 1$ for acceptance and $l = 0$ for rejection).
Thus we have a probability space $(S_{\mathrm{UMC},(x,y)}, \mathrm{Prob})$ with

$$S_{\mathrm{UMC},(x,y)} = \{C_{il} \mid 1 \leq i \leq n,\ l \in \{0, 1\}\}$$

and

$$\begin{aligned} \mathrm{Prob}(\{C_{i0}\}) &= \frac{1}{n} \cdot \left(\frac{1}{2} + \frac{1}{2n}\right) \text{ and} \\ \mathrm{Prob}(\{C_{i1}\}) &= \frac{1}{n} \cdot \left(\frac{1}{2} - \frac{1}{2n}\right) \end{aligned} \tag{2.14}$$

for all $i \in \{1, 2, \ldots, n\}$.
Obviously,

$$A_0 = \{C_{i0} \mid 1 \leq i \leq n\}$$

is the event that UMC outputs the correct answer "reject" ("$x = y$").
Hence,

$$\begin{aligned}
\text{Prob}(A_0) &= \text{Prob}(\text{UMC rejects } (x,y)) \\
&= \sum_{i=1}^{n} \text{Prob}(\{C_{i0}\}) \\
&\underset{(2.14)}{=} \sum_{i=1}^{n} \frac{1}{n} \cdot \left(\frac{1}{2} + \frac{1}{2n}\right) \\
&= n \cdot \frac{1}{n}\left(\frac{1}{2} + \frac{1}{2n}\right) \\
&= \frac{1}{2} + \frac{1}{2n} > \frac{1}{2}.
\end{aligned}$$

(ii) Let $(x, y) \in L_{\text{unequal}}$, i.e., $(x \neq y)$.
Then there exists a $j \in \{1, 2, \ldots, n\}$, such that $x_j \neq y_j$. Without loss of generality we assume that only one j, with $x_j \neq y_j$, exists, clearly the worst case for our analysis. Then we have the computation C_j that accepts (x, y) with certainty (note that $\text{Prob}(\{C_j\}) = 1/n$) and exactly $2(n-1)$ computations C_{il} for $i \in \{1, 2, \ldots, n\} - \{j\}$, $l \in \{0, 1\}$, where C_{il} is the computation in which R_{I} chooses the number i in Phase 1 at random and R_{II} outputs "l" in Phase 2. Thus, our probability space is

$$(S_{\text{UMC},(x,y)}, \text{Prob})$$

with

$$S_{\text{UMC},(x,y)} = \{C_j\} \cup \{C_{il} \mid 1 \leq i \leq n,\ i \neq j,\ l \in \{0,1\}\}$$

and

$$\begin{aligned}
\text{Prob}(\{C_j\}) &= \frac{1}{n}, \\
\text{Prob}(\{C_{i0}\}) &= \frac{1}{n}\left(\frac{1}{2} + \frac{1}{2n}\right), \text{ and} \\
\text{Prob}(\{C_{i1}\}) &= \frac{1}{n}\left(\frac{1}{2} - \frac{1}{2n}\right)
\end{aligned} \tag{2.15}$$

for all $i \in \{1, 2, \ldots, n\}$.
Obviously

$$A_1 = \{C_j\} \cup \{C_{i1} \mid 1 \leq i \leq n,\ i \neq j\}$$

is the event that UMC computes the correct answer "accept" ("$x \neq y''$).
Thus,

$$\begin{aligned}
\text{Prob}(A_1) &= \text{Prob}(\text{UMC accepts } (x,y)) \\
&= \text{Prob}(\{C_j\}) + \sum_{\substack{i=1 \\ i \neq j}}^{n} \text{Prob}(\{C_{i1}\})
\end{aligned}$$

$$\begin{aligned}
&\underset{(2.15)}{=} \frac{1}{n} + \sum_{\substack{i=1\\ i\neq j}}^{n} \frac{1}{n}\left(\frac{1}{2} - \frac{1}{2n}\right)\\
&= \frac{1}{2n} + \frac{1}{2n} + \sum_{i=1}^{n-1}\left(\frac{1}{2n} - \frac{1}{2n^2}\right)\\
&= \frac{1}{2n} + \sum_{i=1}^{n} \frac{1}{2n} - \sum_{i=1}^{n-1} \frac{1}{2n^2}\\
&= \frac{1}{2n} + \frac{1}{2} - \frac{n-1}{2n^2}\\
&= \frac{1}{2} + \frac{1}{2n^2}\\
&\quad \left\{\text{Since } \tfrac{1}{2n} - \tfrac{n-1}{2n^2} = \tfrac{1}{2n^2}\right\}\\
&> \frac{1}{2}.
\end{aligned}$$

□

Exercise 2.4.58. One observes that the protocol UMC is based on a nondeterministic protocol that simply guesses the position j where x and y differ, and then verifies if its guess was correct (i.e., if really $x_j \neq y_j$). We have converted this nondeterministic protocol into an MC protocol by assigning some probabilities to nondeterministic decisions about acceptance and rejection in situations in which the protocol does not know the right answer.[55] The idea is to accept with probability less than, but close to, 1/2, and to reject with the complementary probability.[56] The probability to accept in this uncertain situation must be so close, from below, to 1/2 that one computation accepting with certainty brings the overall probability of acceptance above 1/2. Can one apply this idea for converting any nondeterministic algorithm to an equivalent[57] MC algorithm of the same efficiency?

2.5 Classification of Randomized Algorithms for Optimization Problems

In Section 2.4 we gave a fundamental classification of randomized algorithms for solving decision problems and for computing functions. If one considers optimization problems, then there is no essential reason to classify randomized optimization algorithms in the way described above. This is because if one executes k runs of a randomized algorithm for an optimization problem,

[55]This is the situation where a nondeterministic algorithm rejects its input (in our case, where j has been chosen and $x_j = y_j$).

[56]i.e., larger than, but close to, 1/2

[57]i.e., computing the same problem

then the final output is simply considered the best output of the k outputs computed in the k particular runs, and so one does not need to try to obtain an output that appears as the result in at least half the number of runs. Thus, one does not identify the right output with the frequency of its occurrence in a sequence of runs of the randomized algorithm, but simply takes the best one with respect to the objective function (optimization goal). For instance, if a randomized optimization algorithm A computes an optimal solution only for an input x with probability at least

$$\frac{1}{|x|},$$

it does not mean that A is not useful. One can execute $|x|$ independent runs of A on x and then take the best output. What is then the probability of success? Since the probability of computing no optimal solution in one run is at most

$$1 - \frac{1}{|x|},$$

the probability that $A_{|x|}$ does not find any optimal solution in $|x|$ independent runs is at most[58]

$$\left(1 - \frac{1}{|x|}\right)^{|x|} < \frac{1}{e}.$$

In this way one obtains a constant probability $1 - e^{-1}$ of computing an optimal solution.

Exercise 2.5.59. Show that $|x| \cdot \log_2 |x|$ computation repetitions are sufficient to guarantee the convergence of

$$\mathrm{Prob}\big(A_{|x| \cdot \log_2 |x|} \text{ computes an optimal solution}\big)$$

to 1 with growing $|x|$.

This way, the complexity of the new algorithm $A_{|x|}$ is only $|x|$ times the complexity of A, and so when A is a polynomial-time algorithm, $A_{|x|}$ is a polynomial-time algorithm, too.

Moreover, when dealing with optimization problems, the task is not necessarily to find an optimal solution, but one is usually satisfied with a feasible solution whose cost (quality) does not differ too much from the cost (quality) of an optimal solution. In such a case one can be looking for the probability of finding a relatively good situation, which leads to another classification of randomized algorithms. In what follows, we introduce a classification suitable for randomized optimization algorithms. Since we need this classification only in Chapter 7, one can omit reading this section now and come back to it before starting the study of Chapter 7.

[58]See Lemma A.3.60.

First of all we need to give a definition (a formal description) of optimization problems and to introduce the concept of approximation algorithms. In this way one obtains a reasonable framework in which randomized algorithms can be classified.

Solving an optimization problem cannot in general be considered as computing a function. Solving an optimization problem corresponds more or less to computing a relation R in the sense that, for a given x, it is sufficient to find a y such that $(x, y) \in R$. The reason for this point of view is that there can exist many optimal solutions for an input x, and we are satisfied with any one of them. If one is satisfied even with any "good", but not necessarily optimal, solution, then one has one more reason for regarding an optimization problem as the computing of a relation. We start with a formal definition of an optimization problem.

Definition 2.5.60. *An* **optimization problem** *is a 6-tuple* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \mathrm{cost}, \mathrm{goal})$*, where*

(i) Σ_I *is an alphabet called the* **input alphabet***.*
(ii) Σ_O *is an alphabet called the* **output alphabet***.*
(iii) $L \subseteq \Sigma_I^*$ *is the language of* **feasible inputs** *(as inputs one allows only words that have a reasonable interpretation).*
An $x \in L$ *is called a* **problem instance of** $\boldsymbol{\mathcal{U}}$*.*
(iv) $\mathcal{M}$ *is a function from* L *to* $\mathcal{P}(\Sigma_O^*)$*, and for each* $x \in L$*, the set* $\mathcal{M}(x)$ *is the set of* **feasible solutions for** $\boldsymbol{x}$*.*
(v) cost *is a function* $\mathrm{cost} : \bigcup_{x \in L}(\mathcal{M}(x) \times \{x\}) \to \mathrm{I\!R}^+$*, called the* **cost function***.*
(vi) $\mathrm{goal} \in \{$*minimum, maximum*$\}$ *is the objective.*

A feasible solution $\alpha \in \mathcal{M}(x)$ *is called* **optimal** *for the problem instance* x *of* U *if*

$$\mathrm{cost}(\alpha, x) = \mathbf{Opt}_{\boldsymbol{U}}\boldsymbol{(x)} = \mathrm{goal}\{\mathrm{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}.$$

We say that an algorithm A **solves** $\mathcal{U}$ *if, for any* $x \in L$*,*

(i) $A(x) \in \mathcal{M}(x)$ $\big(A(x)$ *is a feasible solution for the problem instance* x *of* $\mathcal{U}\big)$*, and*
(ii) $\mathrm{cost}(A(x), x) = \mathrm{goal}\{\mathrm{cost}(\beta, x) \mid \beta \in \mathcal{M}(x)\}$*.*

If goal $=$ *minimum,* $\mathcal{U}$ *is called a* **minimization problem***,*
if goal $=$ *maximum,* $\mathcal{U}$ *is called a* **maximization problem***.*

To understand why we define optimization problems this way, let us take a look at the above formal definition of an optimization problem as a 6-tuple $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \mathrm{cost}, \mathrm{goal})$. The input alphabet Σ_I has the same meaning as the input alphabet of decision problems, i.e., Σ_I is used for the representation of input instances of $\mathcal{U}$. Analogously, the output alphabet Σ_O is used for representing the outputs (feasible solutions). The language $L \subseteq \Sigma_I^*$

is the set of correct representations of problem instances. We assume that no word from $L^{\mathsf{C}} = \Sigma_I^* - L$ will occur as an input. This means that we focus on determining the complexity of the optimization, and not on solving the decision problem (Σ_I, L).

A problem instance x usually specifies a set of constraints, and $\mathcal{M}(x)$ is the set of objects (feasible solutions for x) that satisfy these constraints. In the typical case the problem instance x also determines the $\mathrm{cost}(\alpha, x)$ for every solution $\alpha \in \mathcal{M}(x)$. The task is to find an optimal solution in the set $\mathcal{M}(x)$ of feasible solutions for x. The typical difficulty of solving $\mathcal{U}$ lies in the fact that the set $\mathcal{M}(x)$ has such a large cardinality that it is practically[59] impossible to generate all feasible solutions from $\mathcal{M}(x)$ in order to pick the best one.

To make the specification of optimization problems transparent, we often omit the specification of Σ_I and Σ_O, and the specification of coding the data over Σ_I and Σ_O. We assume simply that the typical data such as integers, graphs, and formulae are represented in the usual[60] way described above. This also simplifies the situation in that we can now address these objects directly instead of working with their formal representations. Therefore, one can transparently describe an optimization problem by specifying

- the set L of problem instances,
- the constraints given by any problem instance $x \in L$ and the corresponding set $\mathcal{M}(x)$ for any $x \in L$,
- the cost function, and
- the goal.

Example 2.5.61. **Traveling Salesman Problem (TSP)**

TSP

Input: A weighted complete graph (G, c), where $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ for an $n \in \mathbb{N} - \{0\}$, and $c : E \to \mathbb{N} - \{0\}$.
{More precisely, an input is a word $x \in \{0, 1, \#\}^*$ that codes[61] (represents) a weighted complete graph (G, c).}

Constraints: For any problem instance (G, c), $\mathcal{M}(G, c)$ is the set of all Hamiltonian cycles of G. Each Hamiltonian cycle can be represented as a sequence $v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$ of vertices, where $(i_1, i_2, \ldots, i_n)$ is a permutation of $(1, 2, \ldots, n)$.
{A strictly formal representation of $\mathcal{M}(G, c)$ is the set of all words $y_1\#y_2\#\ldots\#y_n \in \{0, 1, \#\}^* = \Sigma_O^*$ where $y_i \in \{0, 1\}^+$ for $i = 1, 2, \ldots, n$, with

$$\{\mathrm{Number}(y_1), \mathrm{Number}(y_2), \ldots, \mathrm{Number}(y_n)\} = \{1, 2, \ldots, n\},$$

and $\mathrm{Number}(y_1) = 1$.}

[59] in an efficient way

[60] see Chapter 2 in [Hro03]

[61] See [Hro03] for possible string representations of graphs.

Costs: For every Hamiltonian cycle $H = v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1} \in \mathcal{M}(G, c)$,

$$\operatorname{cost}((v_{i_1}, \ldots, v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c\left(\{v_{i_j}, v_{i_{(j \mod n)+1}}\}\right),$$

i.e., the cost of every Hamiltonian cycle is the sum of the weights of all its edges.

Goal: minimum.

For the problem instance of TSP in Figure 2.6, we have

$$\operatorname{cost}((v_1, v_2, v_3, v_4, v_5, v_1), (G, c)) = 8 + 1 + 7 + 2 + 1 = 19 \text{ and}$$
$$\operatorname{cost}((v_1, v_5, v_3, v_2, v_4, v_1), (G, c)) = 1 + 1 + 1 + 1 + 1 = 5.$$

The Hamiltonian cycle $v_1, v_5, v_3, v_2, v_4, v_1$ is the only optimal solution for this problem instance of TSP. □

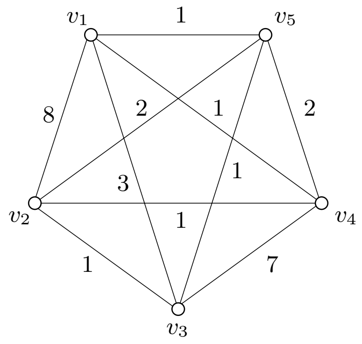


Fig. 2.6.

Exercise 2.5.62. Prove, for any integer $n \geq 2$m, that

$$|\mathcal{M}((G, c))| = (n - 1)!/2$$

for any graph G with n vertices.

A **vertex cover** of a graph $G = (V, E)$ is any set U of vertices of G (i.e., $U \subseteq E$) such that every edge from E is incident[62] to at least one vertex from U. For example, the set $\{v_2, v_4, v_5\}$ is a vertex cover of the graph in Figure 2.7 because each edge of this graph is incident to at least one of there three vertices. The set $\{v_1, v_2, v_3\}$ is not a vertex cover of the graph in Figure 2.7 because the edge $\{v_4, v_5\}$ is not covered by any of the vertices v_1, v_2, and v_3.

[62]An edge is incident to a vertex if this vertex is one of the two endpoints of this edge, i.e., the edge $\{u, v\}$ is incident to the vertices u and v.

Exercise 2.5.63. The **minimum vertex cover problem**, **MIN-VCP**, is a minimization problem, where one searches for a vertex cover of minimal cardinality for a given graph G.

(i) Estimate the set of all vertex covers of the graph in Figure 2.7.
(ii) Give a formal specification of MIN-VCP as a 6-tuple. Use the alphabet $\{0, 1, \#\}$ to represent the input instances and the feasible solutions.

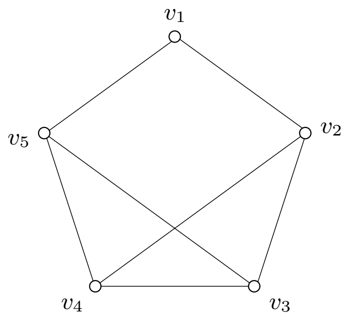


Fig. 2.7.

Example 2.5.64. **The maximum satisfiability problem (MAX-SAT)**

Let $X = \{x_1, x_2, \ldots\}$ be the set of Boolean variables. The set of all literals over X is $Lit_X = \{x, \overline{x} \mid x \in X\}$, where $\overline{x}$ is the negation of x for every variable x. The values 0 and 1 are called Boolean values (constants). A **clause** is any finite disjunction over literals (for instance, $x_1 \vee \overline{x}_3 \vee x_4 \vee \overline{x}_7$). A (Boolean) formula F is in **conjunctive normal form** (**CNF**) if F is a finite conjunction of clauses. A formula F is in k-conjunctive normal form (***k*CNF**) for a positive integer k if every clause of F consists of at most k literals. A formula F is in **E*k*CNF** for a positive integer k if every clause of F consists of exactly k literals over k different variables.

An example of a formula over X in CNF is

$$\Phi = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge \overline{x}_2 \wedge (x_2 \vee x_3) \wedge x_3 \wedge (\overline{x}_1 \vee \overline{x}_3).$$

The maximum satisfiability problem, MAX-SAT, is to find an input assignment to the variables of a given formula in CNF such that the number of satisfied clauses is maximized.

MAX-SAT

Input: A formula $\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ over X in CNF, where F_i is a clause for $i = 1, \ldots, m$, $m \in \mathbb{N} - \{0\}$.

Constraints: For every formula Φ over a set $\{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ of n Boolean variables, the set of feasible solutions is

$$\mathcal{M}(\Phi) = \{0,1\}^n.$$

{Every $\alpha = \alpha_1 \ldots \alpha_n \in \mathcal{M}(\Phi)$, $\alpha_j \in \{0,1\}$ for $j = 1, \ldots, n$, represents an assignment where the value α_j is assigned to the variable x_{i_j}.}

Costs: For every formula Φ and any $\alpha \in \mathcal{M}(\Phi)$, $\text{cost}(\alpha, \Phi)$ is the number of clauses of Φ satisfied by α.

Goal: maximum.

For the formula Φ described above, Table 2.1 presents all eight assignments to the variables x_1, x_2, and x_3 and we can easily observe that the assignments 001, 011, and 101 satisfy five clauses each, and are hence optimal solutions for Φ.

Table 2.1.

x_1 x_2 x_3	$x_1 \vee x_2$	$\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3$	$\overline{x}_2$	$x_2 \vee x_3$	x_3	$\overline{x}_1 \vee \overline{x}_3$	# of satisfied clauses
0 0 0	0	1	1	0	0	1	3
0 0 1	0	1	1	1	1	1	5
0 1 0	1	1	0	1	0	1	4
0 1 1	1	1	0	1	1	1	5
1 0 0	1	1	1	0	0	1	4
1 0 1	1	1	1	1	1	0	5
1 1 0	1	1	0	1	0	1	4
1 1 1	1	0	0	1	1	0	3

If the set of feasible inputs is restricted to formulas in kCNF for a positive integer k, we speak of the MAX-kSAT problem. If one allows only formulas in EkCNF as inputs, then one speaks of the MAX-EkSAT problem.

Example 2.5.65. **Integer linear programming (ILP)**

Given a system of linear equations and a linear function over variables of this linear equation system, the task is to find a solution to the system of equations such that the value of the linear function is minimized. ILP can be phrased as an optimization problem as follows:

ILP

Input: An $m \times n$ matrix

$$A = [a_{ij}]_{i=1,\ldots m, j=1,\ldots,n}$$

and two vectors

$$b = (b_1, \ldots, b_m)^{\mathsf{T}} \text{ and } c = (c_1, \ldots, c_n)$$

for $n, m \in \mathbb{N} - \{0\}$, where a_{ij}, b_i, c_j are integers for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

Constraints: $\mathcal{M}(A,b,c) = \{X = (x_1,\ldots,x_n)^\mathsf{T} \in \mathbb{N}^n \mid AX = b\}$.

$\{\mathcal{M}(A,b,c)$ is the set of all solutions (vectors) that satisfy the system $AX = b$ of linear equations determined by A and $b.\}$

Costs: For every $X = (x_1,\ldots,x_n) \in \mathcal{M}(A,b,c)$,

$$\mathrm{cost}(X,(A,b,c)) = c \cdot X = \sum_{i=1}^{n} c_i x_i.$$

Goal: minimum.

The most discrete optimization problems of our interest are NP-hard, and so one does not hope to solve them in polynomial time. Because of this, we introduce the concept of approximation algorithms for solving hard optimization problems. The idea is to jump from exponential time complexity to a polynomial-time complexity by weakening the requirements. Instead of forcing the computation of an optimal solution, we are satisfied with an "almost optimal" or "nearly optimal" solution. What the term "almost optimal" means is defined below.

Definition 2.5.66. *Let $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \mathrm{cost}, \mathrm{goal})$ be an optimization problem.*

We say that A is a **consistent algorithm for $\mathcal{U}$** *if, for every $x \in L$, the output $A(x)$ of the computation of A on x is a feasible solution for x (i.e., $A(x) \in \mathcal{M}(x)$).*

Let A be a consistent algorithm for $\mathcal{U}$. For every $x \in L$, we define the **approximation ratio, $\mathrm{Ratio}_A(x)$, of A on x** *as*

$$\mathbf{Ratio}_{\boldsymbol{A}}(\boldsymbol{x}) = \max\left\{\frac{\mathrm{cost}(A(x))}{\mathrm{Opt}_{\mathcal{U}}(x)}, \frac{\mathrm{Opt}_{\mathcal{U}}(x)}{\mathrm{cost}(A(x))}\right\},$$

where $\mathrm{Opt}_{\mathcal{U}}(x)$ is the cost of an optimal solution for the instance x of $\mathcal{U}$.

For any positive real number $\delta > 1$, we say that A is a **δ-approximation algorithm for $\mathcal{U}$** *if*

$$\mathrm{Ratio}_A(x) \le \delta$$

for every $x \in L$.

First, we illustrate the concept of approximation algorithms for the minimum vertex cover problem. The idea is to efficiently find a matching[63] in a given graph G, and then to take all vertices incident to the edges of the matching as a vertex cover.

[63] A matching in $G = (V,E)$ is a set $M \subseteq E$ of edges such that there is no vertex incident to more than one edge from M. A matching is maximal if for every $e \in E - M$, the set $M \cup \{e\}$ is not a matching in G.

Algorithm VCA

```
Input: A graph $G = (V, E)$.
Phase 1. $C := \emptyset$;
    {During the computation, $C$ is always a subset of $V$, and at the end of
     the computation $C$ is a vertex cover of $G$.}
    $A := \emptyset$;
    {During the computation, $A$ is always a subset of $E$ (a matching in $G$),
     and when the computation has finished, $A$ is a maximal matching.}
    $E' := E$;
    {During the computation, the set $E' \subseteq E$ contains exactly the edges that
     are not covered by the actual $C$. At the end of the computation, $E' = \emptyset$.}
Phase 2. while $E' \neq \emptyset$ do
            begin
                take an arbitrary edge $\{u, v\}$ from $E'$;
                $C := C \cup \{u, v\}$;
                $A := A \cup \{\{u, v\}\}$;
                $E' := E' - \{$all edges incident to $u$ or $v\}$;
            end
Output: $C$
```

Consider a possible run of the algorithm VCA on the graph in Figure 2.8(a). Let $\{b, c\}$ be the first edge chosen by VCA. Then,

$$C = \{b, c\},\ A = \{\{b, c\}\}, \text{ and } E' = E - \{\{b, a\}, \{b, c\}, \{c, e\}, \{c, d\}\},$$

as depicted in Figure 2.8(b). If the second choice of an edge from E' by VCA is the edge $\{e, f\}$ (Figure 2.8(c)), then

$$C = \{b, c, e, f\}, A = \{\{b, c\}, \{e, f\}\}, \text{ and } E' = \{\{d, h\}, \{d, g\}, \{h, g\}\}.$$

If the last choice of VCA is the edge $\{d, g\}$, then

$$C = \{b, c, e, f, d, g\}, A = \{\{b, c\}, \{e, f\}, \{d, g\}\}, \text{ and } E' = \emptyset.$$

Hence, C is a vertex cover with cost 6. We observe that $\{b, e, d, g\}$ is the optimal vertex cover, and this optimal vertex cover cannot be achieved by any choice of edges by the algorithm VCA.

Exercise 2.5.67. Find a choice of edges in the second phase of VCA such that the resulting vertex cover C contains all vertices of G in Figure 2.8(a).

We show that the algorithm VCA is a 2-approximation algorithm for MIN-VCP and $\text{Time}_{\text{VCA}(G)} \in O(|E|)$ for any instance $G = (V, E)$.

The claim

$$\text{Time}_{\text{VCA}(G)} \in O(|E|)$$

is obvious because every edge from E is manipulated exactly once in VCA.

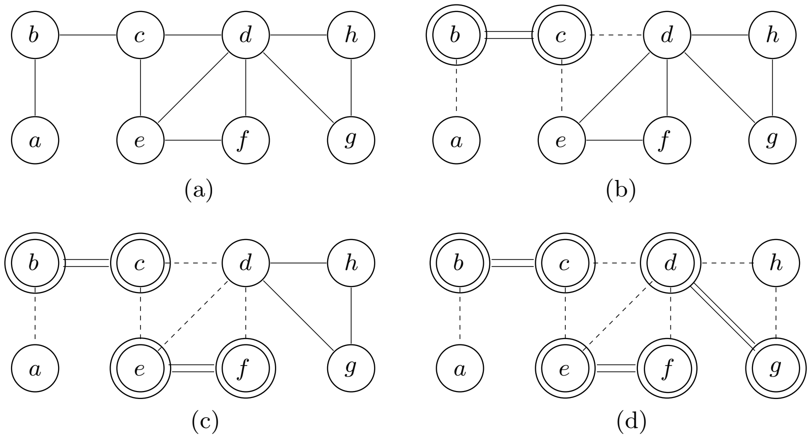


Fig. 2.8.

Since $E' = \emptyset$ at the end of any computation, VCA computes a vertex cover in G (i.e., VCA is a consistent algorithm for MIN-VCP).

To prove

$$\text{Ratio}_{\text{VCA}}(G) \leq 2$$

for every graph G, we observe that

(i) $|C| = 2 \cdot |A|$, and
(ii) A is a matching in G.

To cover the $|A|$ edges of the matching A, one has to take at least $|A|$ vertices. Since $A \subseteq E$, the cardinality of any vertex cover of G is at least $|A|$, i.e.,

$$\text{Opt}_{\text{MIN-VCP}}(G) \geq |A|.$$

Hence

$$\frac{|C|}{\text{Opt}_{\text{MIN-VCP}}(G)} = \frac{2 \cdot |A|}{\text{Opt}_{\text{MIN-VCP}}(G)} \leq 2.$$

Exercise 2.5.68. Construct, for any positive integer n, a graph G_n, such that the optimal vertex cover has cardinality n and the algorithm VCA can compute a vertex cover of cardinality $2n$.

Whether or not the guarantee of an approximation ratio of 2 is sufficient depends on particular applications. Usually one tries to achieve smaller approximation ratios, which requires much more demanding algorithmic ideas. On the other hand, one measures the approximation ratio of an algorithm in the worst-case manner, so a 2-approximation algorithm can provide solutions of essentially better approximation ratios than 2 for typical problem instances.

One of the main goals of applying randomization in the area of discrete optimization is to improve the approximation ratio. Thus, one has to design randomized approximation algorithms that produce feasible solutions whose cost (quality) is not very far from the optimal cost with a high probability. In the analysis of such randomized algorithms one considers the approximation ratio as a random variable.[64] Then the aim is either

(1) to estimate the expected value $\mathrm{E}[\mathrm{Ratio}]$ (or at least to prove a reasonable upper bound for it), or
(2) to guarantee that an approximation ratio δ is achieved with probability at least $1/2$.

These two different aims lead to the following two concepts of defining randomized approximation algorithms.

Definition 2.5.69. *Let* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \mathrm{cost}, \mathrm{goal})$ *be an optimization problem. For any positive real* $\delta > 1$*, a randomized algorithm* A *is called a* **randomized $\mathbf{E}[\delta]$-approximation algorithm for $\mathcal{U}$** *if*

(i) $\mathrm{Prob}(A(x) \in \mathcal{M}(x)) = 1$*, and*
(ii) $\mathrm{E}[\mathrm{Ratio}_A(x)] \leq \delta$

for every $x \in L$*.*

Definition 2.5.70. *Let* $\mathcal{U} = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \mathrm{cost}, \mathrm{goal})$ *be an optimization problem. For any positive real* $\delta > 1$*, a randomized algorithm* A *is called a* **randomized δ-approximation algorithm for $\mathcal{U}$** *if*

(i) $\mathrm{Prob}(A(x) \in \mathcal{M}(x)) = 1$*, and*
(ii) $\mathrm{Prob}(\mathrm{Ratio}_A(x) \leq \delta) \geq 1/2$

for every $x \in L$*.*

The algorithm RSEL designed in Example 2.3.35 for MAX-SAT is clearly a randomized $\mathrm{E}[2]$-approximation algorithm for $\mathcal{U}$.

Let us investigate how much the two proposed classes (concepts) of randomized approximation algorithms really differ. First of all, we claim that these two classes are different in the strongly formal sense, that a randomized $\mathrm{E}[\delta]$-approximation algorithm for $\mathcal{U}$ is not necessarily a randomized δ-approximation algorithm for $\mathcal{U}$, and vice versa.

To recognize this, consider a randomized algorithm A that has exactly 12 runs, $C_1, C_2, \ldots, C_{12}$, on an input x, and all runs have the same probability. Let $\mathrm{Ratio}_{A,x}(C_i)$ be the approximation ratio of the output of C_i. Let us assume that

$$\mathrm{Ratio}_{A,x}(C_i) = 2 \text{ for } i = 1, 2, \ldots, 10 \text{ and } \mathrm{Ratio}_{A,x}(C_j) = 50 \text{ for } j \in \{11, 12\}.$$

Then,

[64] This means that Ratio is here considered as a function that assigns to each run of a randomized algorithm the approximation ratio of the result of this run.

$$\mathrm{E}[\mathrm{Ratio}_{A,x}] = \frac{1}{12} \cdot (10 \cdot 2 + 2 \cdot 50) = 10.$$

On the other hand,

$$\mathrm{Prob}(\mathrm{Ratio}_{A,x} \leq 2) = 10 \cdot \frac{1}{12} = \frac{5}{6} \geq \frac{1}{2}.$$

If the above holds for all input instances x of $\mathcal{U}$, then A is a randomized 2-approximation algorithm for $\mathcal{U}$, but not a randomized $\mathrm{E}[2]$-approximation[65] algorithm for $\mathcal{U}$. If one chooses $\mathrm{Ratio}_{A,x}(C_{11})$ as a very large number (instead of $\mathrm{Ratio}_{A,x}(C_{11}) = 50$), then one can increase the gap between the most "frequent" approximation ratio and the expected approximation ratio to an arbitrarily large extent.

For the opposite direction, we consider a randomized algorithm B that has 1999 runs on any input x, and all these runs are of the same probability. Assume that 1000 of these runs lead to results with approximation ratio 11 and that the remaining 999 runs compute an optimal solution (i.e., 999 runs have approximation ratio 1). Then, $\mathrm{E}[\mathrm{Ratio}_B]$ is slightly greater than 6, but B is not a randomized δ-approximation algorithm for any $\delta < 11$.

Now, it is interesting to observe that even the algorithm[66] B_2 is a randomized 1-approximation algorithm. Thus, the algorithm B is not so bad with respect to the second class (Definition 2.5.70), as it could appear at first glance. Additionally, δ is not too large when compared with $\mathrm{E}[\mathrm{Ratio}_B]$.

> Is it even possible to show a general statement that B is always a randomized δ-approximation algorithm for a $\delta \geq 2 \cdot \mathrm{E}[\mathrm{Ratio}_B]$?

The answer is yes because (as already mentioned) for every random variable X, the probability of getting a value less than or equal to $2 \cdot \mathrm{E}[X]$ is at least $1/2$. This observation offers the following claim.

Lemma 2.5.71. *Let $\delta > 0$ be a real number, and let $\mathcal{U}$ be an optimization problem. For any randomized algorithm B, if B is a randomized $\mathrm{E}[\delta]$-approximation algorithm for $\mathcal{U}$, then B is a randomized γ-approximation algorithm for $\mathcal{U}$ for $\gamma = 2 \cdot \mathrm{E}[\delta]$.*

The following example is not only another illustration of the two concepts of randomized approximation algorithms, but it also deepens our understanding of the relation between these two concepts.

Example 2.5.72. Consider the MAX-EkSAT problem for $k \geq 3$ and the algorithm RSAM presented in Example 2.3.35 that simply generates a random assignment to the variables of a given formula. For each instance

$$F = F_1 \wedge F_2 \wedge \ldots \wedge F_m$$

[65] only a randomized $\mathrm{E}[10]$-approximation algorithm for U

[66] Remember that B_2 consists of two independent runs of the algorithm B on the same input.

of MAX-EkSAT (with F_i in kCNF for $i = 1, \dots, m$), we have defined for $i = 1, \dots, m$ the random variable Z_i as the indicator variable for satisfying F_i. We have shown that

$$\mathrm{E}[Z_i] = 1 - \frac{1}{2^k}$$

is the probability that a random assignment satisfies Z_i. The random variable

$$Z_F = \sum_{i=1}^{m} Z_i$$

counts the number of satisfied clauses in every run of RSEL, and applying linearity of expectation we have proved that

$$\mathrm{E}[Z_F] = \sum_{i=1}^{m} \mathrm{E}[Z_i] = m \cdot \left(1 - \frac{1}{2^k}\right). \tag{2.16}$$

Since the cost of an optimal solution is bounded by m, for every instance F of MAX-EkSAT one obtains[67]

$$\begin{aligned}\mathrm{E}[\mathrm{Ratio}_{\mathrm{RSEL}}(F)] &\le \frac{\mathrm{Opt}_{\text{MAX-E}k\text{SAT}}(F)}{\mathrm{E}[Z_F]}\\ &\le \frac{m}{\mathrm{E}[Z_F]}\\ &\le \frac{m}{m \cdot (1 - 2^{-k})} = \frac{2^k}{2^k - 1}.\end{aligned}$$

Thus, the algorithm RSEL is a randomized $\mathrm{E}\left[2^k/(2^k - 1)\right]$-approximation algorithm for MAX-EkSAT.

Our next aim is to apply (2.16) in order to show that RSEL is a randomized $2^{k-1}/(2^{k-1} - 1)$-approximation algorithm for MAX-EkSAT. First, we observe that

$$\mathrm{E}[Z_F] = m \cdot (1 - 1/2^k)$$

also says that $m \cdot (1 - 1/2^k)$ is the average number[68] of satisfied clauses over all assignments to the variables of F. The number $m \cdot (1 - 1/2^k)$ is the average of m and $m \cdot (1 - 2/2^k) = m \cdot (1 - 1/2^{k-1})$. Hence, the number $m \cdot (1 - 1/2^k)$ lies in the middle between n and $m \cdot (1 - 1/2^{k-1})$ on the real axis (Figure 2.9).

To complete our argument we need to show that at least half the assignments satisfy at least $m \cdot (1 - 1/2^{k-1})$ clauses. Again, the combinatorial idea says that if more than half the clauses satisfy fewer than $m \cdot (1 - 1/2^{k-1})$ clauses, then one cannot achieve the average value $m \cdot (1 - 2^{-k})$, even if all remaining assignments satisfy all m clauses (Figure 2.9).

If one wants to verify this claim by counting, denote

[67]Observe that $\mathrm{Ratio}_{\mathrm{RSEL}}(F) = \mathrm{Opt}_{\text{MAX-E}k\text{SAT}}(F) \cdot (1/Z_F)$ is a random variable, where $\mathrm{Opt}_{\text{MAX-E}k\text{SAT}}(F)$ is a constant for a given, fixed F, and F is fixed in our analysis (in our probability space).

[68]Because all assignments have the same probability of being chosen.

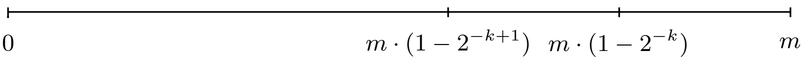


Fig. 2.9.

(i) by l the number of assignments that satisfy fewer than $m \cdot (1 - 1/2^{k-1})$ clauses, and
(ii) by $u = 2^n - l$ the number of assignments that satisfy at least $m \cdot (1 - 1/2^{k-1})$ clauses.

Then,

$$\mathrm{E}[Z_F] \leq \frac{1}{2^n} \cdot (l \cdot [m \cdot (1 - 2^{-k+1}) - 1] + u \cdot m). \tag{2.17}$$

The inequalities (2.16) and (2.17) imply that

$$m \cdot (1 - 2^{-k}) \leq \frac{1}{2^n}(l \cdot \left[m \cdot (1 - 2^{-k+1}) - 1\right] + u \cdot m),$$

and so

$$2^n \cdot m \cdot (1 - 2^{-k}) \leq l \cdot \left[m \cdot (1 - 2^{-k+1}) - 1\right] + u \cdot m,$$

which can be true for $u > l$ only. □

Exercise 2.5.73. Let c be a real number and let X be a random variable in a probability space (S, Prob). Let Y be a random variable defined by $Y = c \cdot 1/X$, i.e.,

$$Y(s) = c \cdot \frac{1}{X(s)}$$

for all $s \in S$. Prove that

$$\mathrm{E}[Y] = c \cdot \frac{1}{\mathrm{E}[X]}.$$

Example 2.5.74. Consider the **MAX-CUT** problem defined as follows. The instances of MAX-CUT are (unweighted) graphs. For any given graph $G = (V, E)$, a pair (V_1, V_2) is a cut of G if

$$V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset,$$

and every cut is considered as a feasible solution for the MAX-CUT problem. The cost of a cut (V_1, V_2) is the number of edges between the vertices from V_1 and the vertices from V_2, i.e.,

$$\mathrm{cost}(V_1, V_2) = |E \cap (V_1 \times V_2)|.$$

The objective is to find a cut with the maximal possible cost. We show that a random choice of cut results in a randomized E[2]-approximation algorithm.

Algorithm RC (Random Cut)

```
Input: A graph $G = (V, E)$.
Step 1: $V_1 = \emptyset$, $V_2 = \emptyset$.
Step 2:
    for every vertex $v \in V$ do
        assign $v$ to $V_1$ or to $V_2$ at random (with the same probabilities)
Output: The cut $(V_1, V_2)$.
```

For every edge $e = \{u, v\} \in E$, we define the indicator variable X_e as

$$X_e(C) = \begin{cases} 0 \text{ if the output of } C \text{ is the cut } (V_1, V_2) \text{ and both } u, v \in V_1 \text{ or} \\ \quad \text{both } u, v \in V_2, \\ 1 \text{ if } e \text{ lies in the cut } (V_1, V_2) \text{ computed in } C\text{, i.e., if } u \in V_1 \text{ and} \\ \quad v \in V_2 \text{ or vice versa} \end{cases}$$

for every run C of RC. Since X_e is an indicator variable, we have

$$\mathrm{E}[X_e] = 1 \cdot \mathrm{Prob}(X_e = 1) + 0 \cdot \mathrm{Prob}(X_e = 0) = \mathrm{Prob}(X_e = 1)\,,$$

and so $\mathrm{E}[X_e]$ is the probability of the event $X_e = 1$. Clearly, for every edge $e = \{u, v\}$

$$\begin{aligned} \mathrm{Prob}\big(X_{\{u,v\}} = 1\big) &= \mathrm{Prob}(u \in V_1 \wedge v \in V_2) + \mathrm{Prob}(u \in V_2 \wedge v \in V_1) \\ &= \mathrm{Prob}(u \in V_1) \cdot \mathrm{Prob}(v \in V_2) \\ &\quad + \mathrm{Prob}(u \in V_2) \cdot \mathrm{Prob}(v \in V_1) \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2} \end{aligned}$$

and so

$$\mathrm{E}[X_e] = \frac{1}{2}.$$

Let $X = \sum_{e \in E} X_e$ be the random variable counting the number of edges between V_1 and V_2. Due to linearity of expectation, one obtains

$$\mathrm{E}[X] = \sum_{e \in E} \mathrm{E}[X_e] = \frac{1}{2} \cdot |E|.$$

Since $|E| \geq \mathrm{Opt}_{\text{MAX-CUT}}(G)$ and $\mathrm{Ratio}_{\mathrm{RC}}(G)$ is the random variable

$$\mathrm{Opt}_{\text{MAX-CUT}}(G) \cdot \frac{1}{X},$$

one obtains

$$\mathrm{E}[\mathrm{Ratio}_{\mathrm{RC}}(G)] = \frac{\mathrm{Opt}_{\text{MAX-CUT}}(G)}{\mathrm{E}[X]} \leq \frac{|E|}{\mathrm{E}[X]} = 2.$$

Thus we have shown that RC is a randomized E[2]-approximation algorithm for MAX-CUT. □

Exercise 2.5.75. Is it possible to show for a $\delta < 4$ that RC is a randomized δ-approximation algorithm?

2.6 Paradigms of the Design of Randomized Algorithms

In the previous sections we have already indicated that randomized algorithms may be essentially more efficient than their deterministic counterparts, and so randomization can be a source of unbelievable efficiency. The fundamental questions attending us during the course on randomization are the following:

- *What is the nature of (the reasons for) the success of randomization?*
- *Do there exist robust design methods for randomized algorithms that are helpful in the search for an efficient randomized algorithm for a given problem?*

In this section we introduce the fundamentals of the most famous concepts and methods for the design of randomized algorithms, and provide the initial ideas that explain the power of randomization this way. The following chapters of this book are devoted to an involved study and applications of these particular methods.

FOILING THE ADVERSARY

The method of foiling the adversary, also called the method of avoiding the worst-case problem instances, lies behind the design of any randomized algorithm. One considers the design of an algorithm as a game between two players. One player is the designer, who tries to design an efficient algorithm for a given problem, and the other player is an adversary, who for any designed algorithm tries to construct an input on which the algorithm does not work efficiently or even correctly. This is the typical game for designing and analyzing deterministic algorithms, where the classic adversary argument establishes a lower bound on the complexity of a given algorithm by constructing an input instance on which the algorithm fares poorly. It corresponds to our view of complexity as worst-case complexity, which means that only those algorithms, that run efficiently on every input[69] are considered to be efficient. An adversary has an advantage in this game because the designer has first to present his algorithm and, knowing the algorithm, it is not often difficult to find a hard problem instance for the algorithm. This game can essentially change when the designer designs a randomized algorithm A (instead of a deterministic one). For sure, an adversary learns the designed randomized algorithm too, but one does not know which of the possible runs of the algorithm A will be chosen at random. While an adversary may still be able to construct an input that is hard for one run, or a small fraction of runs of A, it is difficult to devise a single input that is likely to defeat most of the runs of A.

[69]For instance, the time complexity of a deterministic algorithm is defined as $\text{Time}(n) = \max\{\text{Time}(C) \mid C \text{ is a computation on an input of length } n\}$.

Learning this game and switching to the first model of randomized algorithms, one can recognize the main idea for the design of randomized algorithms. It may happen for a problem U that on the one hand there exists a hard problem instance for each deterministic algorithm solving U, but on the other hand, for every instance x of U, there are sufficiently many deterministic algorithms that work correctly and efficiently on x. If one designs a randomized algorithm as a probability distribution over a convenient set of deterministic algorithms, then this algorithm works correctly and efficiently on every problem instance with high probability.[70]

As we have already mentioned, this design paradigm is common for all randomized algorithms. We illustrate this by considering two examples of randomized algorithms already presented. First we consider the randomized protocol R comparing the data on two computers R_{I} and R_{II}. This protocol corresponds to a uniform probability distribution over $\mathit{Prim}\left(n^2\right)$ deterministic protocols P_q for all $q \in \mathrm{PRIM}\left(n^2\right)$. Each of these protocols is efficient on all inputs and works correctly on most of the inputs. But, for some inputs (x, y) with $x \neq y$, it may compute the wrong answer "$x = y$". The crucial fact is that, for every input (x, y), at least

$$\mathit{Prim}\left(n^2\right) - (n-1) \geq 1 - \frac{2 \cdot \ln n}{n}$$

of the protocols P_q provide the correct answer. Taking a random choice of a protocol for a given input (x, y), one achieves a high probability of getting the right answer. Here, it is interesting to mention that one can prove even more than that every deterministic protocol solving this problem has a hard input (x, y) on which it must exchange $|x|$ communication bits. It is true that every deterministic protocol for this task has a high communication complexity on most inputs[71]. Thus, it is provable that the designed randomized protocol R for L_{equal} is exponentially more efficient than any deterministic protocol for L_{equal}, and this holds not only with respect to the worst-case complexity, but also with respect to the average complexity of all inputs of the same size.

In the case of the design of the randomized communication protocol R, the aim was to find a set of efficient deterministic protocols, each of them working correctly on almost all inputs. Designing the randomized Quicksort, one is looking for a set of deterministic strategies such that

(i) each of these strategies works correctly on every input, and
(ii) each strategy proceeds efficiently on most inputs.

Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set to be sorted. If one considers A as an unsorted sequence $a_1, a_2, \ldots, a_n$, a possible strategy could be to take the element a_1 as a pivot. This strategy in the framework of the "divide and

[70]i.e., there is no hard input for this algorithm, and so no adversary has a chance to defeat this algorithm.

[71]Most inputs require the exchange of at least $n - 1$ bits.

conquer" method leads to the following recurrence for estimating the number of comparisons executed.

$$\begin{aligned}\mathrm{Time}(1) &= 0\\ \mathrm{Time}(n) &= n-1+\mathrm{Time}(|A_<|)+\mathrm{Time}(|A_>|)\,.\end{aligned}$$

If an adversary takes the input $a_1, \ldots, a_n$ with $a_1 > a_2 > \ldots > a_n$, then $A_> = \emptyset$ and the complexity of the work on this input is in $\Theta(n^2)$. Clearly, an adversary can construct a hard input for every deterministic strategy of choosing a pivot (without precomputation[72]). The kernel of the success of the randomized Quicksort lies in the fact that each strategy of choosing a pivot leads to complexity in $O(n \cdot \log n)$ for most of the inputs.

Exercise 2.6.76. Let $T : \mathbb{N} \to \mathbb{N}$ be a function that satisfies the following recurrence.

$$\begin{aligned}T(1) &= 0\\ T(n) &= n-1+T\left(\left\lfloor\frac{n}{10}\right\rfloor\right)+T\left(\left\lceil\frac{9}{10}\cdot n\right\rceil\right)\end{aligned}$$

Prove that $T(n) \in O(n \cdot \log n)$. A consequence of this fact is that, from the asymptotical point of view, at least $8/10 = 4/5$ of the elements of A are good pivots. How many elements in A are good pivots?

We show the power of the paradigm of foiling the adversary in Chapter 3 by outlining possible contributions of randomization for hashing (data management) and for online problems (optimization procedures without any knowledge about the future).

ABUNDANCE OF WITNESSES

The method of abundance of witnesses is especially suitable for solving decision problems. Generally, a decision problem is a task of deciding whether a given input[73] has a special property of interest or not. For instance, taking positive integers as possible inputs, one can ask whether a given n is prime or not. In our example for the motivation in Section 1.2, the inputs were pairs (x, y), and the task was to decide whether $x \neq y$ or $x = y$.

One can describe the application of the method of abundance of witnesses for solving a decision problem U as follows. Assume that we do not have any efficient deterministic algorithm for this problem, or that there does not even exist any efficient deterministic algorithm solving U. The first step of

[72] If one uses some comparisons for estimating a pivot, then one can assure a reasonable ratio between $|A_<|$ and $|A_>|$, and there is no hard input then for the pivot precomputed in a reasonable way.

[73] more precisely, the object represented by this input

the application of the method of abundance of witnesses is to search for a suitable definition of witnesses. A **witness** has to be an additional information (represented by a string) to a given input x, that helps to efficiently prove that x has the required property (or does not have the required property). For instance, a factor m of a given positive integer n is a witness of the fact "n is prime." In the design of the randomized protocol R for L_{unequal}, a prime p is a witness of the fact[74] "$x \neq y$" if

$$\text{Number}(x) \mod p \neq \text{Number}(y) \mod p.$$

If one has got such a prime p for free, then one can prove the fact "x differs from y" by an efficient communication. Obviously, in reality is no one giving us a witness for free, and in many cases there is no possibility of efficiently computing a witness by a deterministic procedure (if this were possible, then one would have an efficient deterministic algorithm for the original problem).

To design an efficient randomized algorithm one needs for every input a set of witness candidates that contain reasonably many witnesses. Considering our communication protocol R, PRIM $\left(n^2\right)$ is the set of candidates for witnessing "$x \neq y$" for any input (x, y). For every input (x, y) with $x \neq y$, at least $\frac{n^2}{\ln n^2} - (n-1)$ candidates from the approximately $\frac{n^2}{\ln n^2}$ candidates in PRIM $\left(n^2\right)$ are witnesses of "$x \neq y$".

Hence, the probability of uniformly choosing a witness from the set PRIM $\left(n^2\right)$ of witness candidates at random is at least

$$\frac{\frac{n^2}{\ln n^2} - (n-1)}{\frac{n^2}{\ln n^2}} \geq 1 - \frac{\ln n^2}{n}.$$

This is very promising because the probability is very close to 1. But even if the probability of choosing a witness at random would be only 1/2, we would be satisfied. In such a case it is sufficient to perform several random choices, because the probability of getting a witness grows very fast with the number of random attempts.

Now, one can be surprised that, despite of the high abundance of witnesses, we are not able to efficiently find a witness in a deterministic way. A possibility would be to systematically search in the set of all candidates in such a way that a witness has to be found in a short time. The core of the problem is that, for any input, the distribution of witnesses among candidates for witnessing may be completely different. Thus, if one fixes a search strategy in the set of candidates, an adversary can always find input for which this strategy does not work efficiently.

Consider the protocol R. Here, one can even prove that there is no strategy that efficiently finds a witness for any input (x, y). To give a transparent example, consider the simple strategy that tries to look on all primes from PRIM $\left(n^2\right)$ in order, from the smallest prime to the largest one.

[74] of the difference between x and y

Obviously, one must find a witness after at most n tests, because there are at most $n-1$ non-witnesses among the candidates. Unfortunately n probes means communication complexity $n \cdot 4 \cdot \log_2 n$, which is more than we are willing to accept. Why is there no assurance of finding a witness after a few probes? The explanation is that in order to find a witness our strategy needs $k+1$ probes for any input (x,y) with

$$\text{Number}(x) - \text{Number}(y) = p_1 \cdot p_2 \cdot \ldots \cdot p_k,$$

where $k = \frac{n}{2(\log_2 n)^2}$ and $p_1 < p_2 < \ldots < p_k$ are the smallest primes.

One can easily imagine that, for any other order of primes in $\text{PRIM}\left(n^2\right)$, one can find inputs (x,y) for which too many probes are necessary to obtain a witness.

The art of applying the method of abundance of witnesses is in the search for a promising definition of a witness for the property to be verified, and in an appropriate choice of the set of candidates for witnessing. Hence, a useful specification of witnesses has to satisfy the following conditions:

(i) If one has a witness for an input, then one can efficiently prove that the input has the required property (or does not have the required property).
(ii) If one has a witness candidate for an input x, then one can efficiently verify whether or not the candidate is a witness of the claim "x has the required property".
(iii) The set of candidates for an input contains many witnesses. One aims to have a constant ratio between the cardinality of the subset of witness and the cardinality of the set of all candidates.

If (i), (ii), and (iii) hold, then we have the guarantee that a random choice of a candidate provides a good basis for the design of an efficient randomized algorithm solving the given problem.

The method of abundance of witnesses can be applied to computing functions, too. If F is a function and x is an argument of F, a witness for x can be considered as additional information y such that, knowing (x,y), one can compute $F(x)$ more efficiently than without y.[75] But in this book we restrict our attention on designing randomized algorithms by the method of abundance of witnesses to solving decision problems only. In Chapter 6 we show the power of this design paradigm by designing efficient randomized algorithms for primality testing.

FINGERPRINTING

Fingerprinting (also called **Freivalds' technique**) can be viewed as a special case of the method of abundance of witnesses for solving equivalence problems.

[75] For instance, the quadratic roots for an integer $a \in \{2, \ldots, n-1\}$ modulo n build a witness for factorizing n. Although one does not know any polynomial-time algorithm for factorization of integers, having this witness we can factorize n efficiently.

An instance of an equivalence problem consists of two complete representations of two objects, and the task is to decide whether or not these two representations describe the same object. For example, an instance of the problem of equivalence of two polynomials is to decide whether the two polynomials $(x_1-2x_2)^2\cdot(x_1+x_3)^3\cdot(x_2-x_3)$ and $(x_1^5-7x_1^4x_2^2+6x_2x_3)\cdot(x_2-x_3+x_1)^2$ represent the same polynomial. Our protocol R for L_{unequal} also solves an equivalence problem, because the task is to decide for any input (x, y) whether $x = y$ or $x \neq y$.

The idea of fingerprinting is to view witnesses as mappings that map the complex, full, and hardly comparable descriptions of objects in some partial descriptions. On the one hand, we require that these partial descriptions be short and easy to compare. On the other hand, the partial descriptions have to be representative, in the sense, that, though their incompleteness, they save the most essential characteristics of the objects represented. Comparing partial descriptions determined by a randomly chosen mapping instead of comparing full, complex descriptions has to be a way of solving a given equivalence problem correctly with high probability.

A good way to transparently present the fingerprinting method is the following schema.

Schema of the Fingerprinting Method

Objective: To decide the equivalence of two objects O_1 and O_2 with complex representations.

Phase 1: Let M be a "suitable" set of mappings that map the full representation of the objects considered to some partial representations of these objects.
Choose a mapping h from M at random.

Phase 2: Compute $h(O_1)$ and $h(O_2)$.
The partial representation $h(O_i)$ of O_i is called the **fingerprint** of O_i for $i = 1, 2$.

Phase 3: Compare the fingerprints $h(O_1)$ and $h(O_2)$.

```
if $h(O_1) = h(O_2)$ then
    output "$O_1$ and $O_2$ are equivalent";
else
    output "$O_1$ and $O_2$ are not equivalent";
```

In the randomized protocol R for L_{equal}, the objects O_1 and O_2 were two large numbers of n bits ($n = 10^{16}$). The set M was specified as

$$\{h_p \mid h_p(m) = m \bmod p \text{ for all } m \in \mathbb{N},\ p \text{ is a prime}, p \leq n^2\}.$$

For a randomly chosen prime p,

$$h_p(O_1) = O_1 \bmod p \text{ and } h_p(O_2) = O_2 \bmod p$$

were the fingerprints of O_1 and O_2.

The kernel of fingerprinting is that $h_p(O_i)$ is an essentially shorter representation of the number O_i than its full binary representation, and so the comparison of $h_p(O_1)$ and $h_p(O_2)$ can be substantially more efficient than directly comparing O_1 and O_2. But this gain is achievable only due to incompleteness of the representation $h_p(O_i)$ of O_i, and so one has to take the possibility of an error into account. The set M is the set of candidates for witnesses of the non-equivalence of O_1 and O_2 (i.e., if "$O_1 \neq O_2$"). If, for every pair of different objects O_1 and O_2, there are many[76] witnesses of "$O_1 \neq O_2$" in M, then one can reduce the error probability to an arbitrarily small value.

The art of applying the fingerprinting method for the design of randomized algorithms lies in a suitable choice of the set M. On the one hand, the fingerprints should be as short as possible in order to assure an efficient comparison. On the other hand, the fingerprints have to involve as much information about their objects as possible[77] in order to reduce the probability of losing the information about the characteristics of O_1 and O_2 in which they differ.

Thus, searching for a suitable set M, one takes care at the trade-off between the degree of "compression" from O to $h(O)$ and the error probability. Designing the randomized protocol R for L_{equal} by the fingerprinting method, we were able to achieve an exponential representation reduction, namely $|h(O)| \in O(\log_2 |O|)$, and we paid for this big gain with an error probability tending to O with growing $|O|$ only. This is an exemplary success of fingerprinting, because one can hardly expect more than an exponential reduction of complexity for negligible error probability.

Further elegant and strong applications of fingerprinting are presented in Chapter 4.

RANDOM SAMPLING

A random sample from an object set is often representative of the whole set. Sometimes it is not easy to deterministically construct objects with some given properties (or satisfying some special constraints) despite the known fact that there are many such objects in a set. One or a few random samples from this set provide, with a high probability, an object having the required properties. As an illustration one can also consider the random generation of witnesses. One is unable to efficiently generate a witness deterministically, but one can get a witness by a random choice from a set of candidates that contains many witnesses. In this way, one may view the design of the randomized protocol R as an application of the method of random sampling, too.

A special case of random sampling is the so-called **probabilistic method**, based on the following two facts.

[76]relative to $|M|$

[77]This requirement is responsible for the term "fingerprinting", because fingerprints are considered unambiguous identifications of men.

Fact 1: For every random variable X, there is a value that is not smaller than the expectation $E[X]$, and there is a value that is not larger than $E[X]$.

Fact 2: If a randomly chosen object of a universe[78] has a property with positive probability, then there exists an object having this property.

Although these two facts are trivial observations, they may have surprisingly strong applications. We have applied them in order to design a simple randomized algorithm for MAX-SAT, based on the fact that the expected number of clauses satisfied by a random assignment is at least half the total number of clauses. If one considers MAX-EkSAT for a positive integer k, then the expected number of satisfied clauses is $2^k/(2^k-1)\cdot m$ for any formula of m clauses. In this case, Fact 1 says that there exists an assignment that satisfies $2^k/((2^k-1)\cdot m)$ clauses of any given formula of m clauses. We showed in Example 2.5.72 that few random samples suffices to get with high probability an assignment that satisfies $m \cdot (2^{k-1}/(2^{k-1}-1))$ clauses from m clauses.

More applications of the method of random sampling are presented in Chapter 5, where we even design an efficient randomized algorithm for an algebraic problem that is not known to be in P.

AMPLIFICATION

The paradigm of the amplification of the success probability says nothing else than one can increase the success probability (the probability of computing the correct result) by repeating independent computations on the same input. The classification of randomized algorithm presented is based on this paradigm, and we have learned how much amplification is helpful to different classes of randomized algorithms.

There are also circumstances under which repeating entire runs of the randomized algorithm on the same input does not help very much, or even does not help at all. But repetitions of some critical parts of the computation only[79] may be helpful in such cases. A part of a computation may be viewed as critical when the success probability is essentially decreased in this part relative to other parts of the computation. This more involved version of the amplification method will be presented in Chapter 5 for the problem of finding a minimum cut in a graph.

RANDOM ROUNDING

One uses the method of random rounding in combination with the method of relaxation to linear programming to solve optimization problems. The method of relaxation first relaxes a hard discrete optimization problem to an efficiently

[78]set of objects

[79]instead of repeating complete runs

solvable problem instance of linear programming by increasing the size of the solution space (more precisely, by removing some constraints[80] on the variables domains). For instance, integer linear programming and 0/1-linear programming are NP-hard problems, but allowing real solutions (i.e., removing the requirement that the values of the elements of the solutions must be integers or Boolean values) one gets the problem of linear programming which is solvable in polynomial time. Solving the relaxed problem instance, one obtains a solution that is not necessarily a feasible solution of the original discrete problem. To get a feasible solution of the original problem, one can try to round the real values of the computed solution of the relaxed problem instance. The hope is that the solution obtained by rounding is of a good quality, because the computed solution of the relaxed problem instance is optimal for the real domain.

One can sometimes view random rounding as a special case of random sampling. Instead of picking up samples from a set S by uniform probability distribution over S, one applies the relaxation method in order to compute another probability distribution over S, and then uses this probability distribution for random sampling in S. We illustrate this view on random rounding in Chapter 7 by designing a randomized approximation algorithm for MAX-SAT.

GENERAL OBSERVATIONS

We introduced above some basic paradigms for the design of randomized algorithms, and we aim to present them as successful methods for designing randomized algorithms for concrete problems in the following chapters. We call attention to the fact these design methods do not partition the class of randomized algorithms into disjoint subclasses. More often than not, the contrary is true. Usually, several of the above design methods are behind the design of a randomized algorithm. The best illustration of this statement is the randomized protocol R. It is based on fingerprinting (as a special case of the method of abundance of witnesses) as well as on the method of foiling the adversary and random sampling. Amplification can be additionally used to increase the success probability (to decrease the error probability). However, we relate the design of the protocol R, without any doubt, to fingerprinting, because this method is essential for the design idea.

This is the approach we will use in the following chapters. For every design paradigm, we present randomized algorithms, for whose design this paradigm is the most helpful instrument.

[80]This constraints deletion is called relaxation.

2.7 Summary

Probability theory is the language and the basic instrument for modeling and analyzing randomized algorithms. In the name of transparency, in this introductory course we restrict our attention to finite probability spaces. Modeling a random (probabilistic) experiment begins with fixing the sample space S of all possible outcomes (results) of the experiment. Every element of S is considered an elementary event in the sense that in this model no elementary event can be viewed as a collection of even smaller ones, i.e., elementary events are the atoms of the model, and so each elementary event (outcome of the experiment) excludes the others. An event is an arbitrary subset of the sample space S of all elementary events. After fixing S, one determines the probabilities of particular elementary events.[81] The probability of any A is then determined by the sum of the probabilities of all elementary events in A.

The notion[82] of conditional probability was introduced in order to get an instrument for analyzing experiments in which one has partial information about the outcome of the experiment before it is finished. In this framework such a partial information alway corresponds to an event B that occurs with certainty. The conditional probability of an event A when an event B occurs with certainty (i.e., the probability of A given B) is the probability of the event $A \cap B$ in a new probability space, where B is the sample space of all possible elementary events. The probability distribution on B is unambiguously determined by the requirement that, for any two elementary events from B, the ratio of their probabilities in the new probability space is the same as in the original probability space.

We say that two events A and B are independent if the probability of A is equal to the probability of A given B. Thus, the meaning of independence is that the occurrence of B with certainty does not have any influence on the probability of A, and vice versa. A consequence of this concept of independence and, in fact, an equivalent definition of independence of A and B is that A and B are independent if and only if the probability of $A \cap B$ is equal to the product of the probabilities of A and B.

Other fundamental instruments for the analysis of probabilistic experiments are the concepts of random variables as functions from the sample space to $\mathbb{R}$, and the expectation of a random variable as the weighted[83] average of the values of the random variable.

For randomized algorithms, one requires efficiency and correctness with high probability for every input. Therefore the behavior of a given randomized algorithm has to be analyzed on every input. For a given randomized algorithm A and an input x, we consider the set $S_{A,x}$ of all computations (runs) of A on

[81] This must be done in a way such that the sum over the probabilities of all elementary events is equal to 1.

[82] concept

[83] where the weights are determined by the probabilities of having the corresponding values of the random variables

x as the sample space. If one chooses the random variable (called the indicator variable in this case) that assigns 1 to any run with the correct output and 0 to any run with a wrong output, then the expectation of this random variable is exactly the probability that A computes the correct output for the input x. This probability is called the success probability of A on x. The probability of the complementary event (i.e., of the event that A computes a wrong output on x) is called the error probability of A on x. If one takes a random variable that assigns to every computation its complexity, then the expectation of this random variable equals the expected complexity of A on x.

Whenever possible, we view a randomized algorithm as a probability distribution over a set of deterministic algorithms. In general, one can view randomized algorithms as nondeterministic algorithms with a probability distribution over each nondeterministic computation splitting.

We classify randomized algorithms with respect to the speed at which their error probabilities are reduced by executing independent runs of the algorithm on the same input. Las Vegas algorithms are the most convenient ones, because they never produce a wrong output. We consider two versions of them. In the first version all computations end with the correct result, and in the second version the output "?" ("I don't know") is allowed with bounded probability. In the first version, we always consider (analyze) the expected complexity.

One-sided-error Monte Carlo algorithms are considered only for decision problems, i.e., for problems of deciding whether or not a given input has the required property. For any input having the property, one forces the randomized algorithm to recognize this fact with a probability of at least $1/2$ (or with a probability of at least ϵ for an $\epsilon > 0$). A strong requirement is that, for every input without this property, the algorithm rejects this input with certainty (i.e., in all runs). A positive consequence of this requirement is that the error probability of one-sided-error Monte Carlo algorithms tends to 0 with exponential speed in the number of independent runs executed.

A bounded-error Monte Carlo algorithm is a randomized algorithm for which there exists a constant ϵ such that, for every input x, the algorithm computes the correct output with probability at least $1/2 + \epsilon$. The crucial point is the requirement of a fixed, constant[84] distance ϵ from $1/2$ for the error probability that assures that constantly[85] many run executions are sufficient for reducing the error probability to an arbitrarily small given constant. If one relaxes this requirement to the constraint that the success probability is greater than $1/2$ for every input (i.e., if for different inputs the distance of the success probability to $1/2$ may be different, and may converge to $1/2$ with input length), then exponentially[86] many runs may be necessary to reduce the error probability below a given constant. The randomized algorithms with this

[84]independent of the input

[85]The number of the executed runs on the same input is, in this sense, a constant.

[86]in the input length

relaxed constraint in the success probability are called unbounded-error Monte Carlo algorithms, or simply (general) Monte Carlo algorithms. This algorithm class has a large expressive power because all nondeterministic polynomial-time algorithms can be simulated by polynomial-time Monte Carlo algorithms.

The classification presented is suitable for solving decision and equivalence problems, and for computing functions. Solving optimization problems, one usually does not search for a unique correct solution, but one is satisfied with any of the possibly many optimal solutions, or even with a nonoptimal solution with a reasonable quality. This quality is measured by the approximation ratio, which may be viewed as a random variable. One may be interested in analyzing the expectation of the approximation ratio, or in guaranteeing an approximation ratio with probability at least 1/2.

Presenting several examples, we have seen that randomized algorithms may be more efficient than their best known, or even the best possible, deterministic counterparts. In our exemplary motivation for the study of randomization in Section 1.2, we showed that the gap between determinism and randomization can be of exponential size. The general robust ideas behind randomization are called here paradigms of the design of randomized algorithms. The most important, recognized paradigms are "foiling the adversary," "abundance of witnesses," "fingerprinting," "random sampling," "random rounding," and "amplification."

The paradigm "foiling the adversary," also called "the method of avoiding worst-case inputs," can be recognized behind the design of every randomized algorithm. The idea is to overcome the situation where, for each deterministic algorithm, there exist hard inputs (on which the algorithm does not run efficiently or correctly) in the following way. One finds a set of deterministic algorithms (strategies) such that, for each instance of the problem considered, most of these algorithms compute the correct result efficiently. Then, the randomized algorithm is simply designed as a probability distribution over this set of deterministic algorithms.

The method of "abundance of witnesses" is especially suitable for solving decision problems. A witness y for an input x is information with which one can compute the correct output for x more efficiently than without. If one has sufficiently many witnesses for every instance x of the problem, then one can try to get a witness by choosing an element for a set of witness candidates at random. If there is an abundance of witnesses in the set of candidates, and the witnesses are randomly distributed in this candidate set, there is no possibility of efficiently constructing a witness in the deterministic way. But a random sample from the set of candidates leads to a witness with reasonable probability.

The method of random sampling helps construct objects having a special property when one knows that there are many such objects; but, because of their random distribution in the set of all objects of the type considered, one is unable to find or to construct such an object efficiently in a deterministic way. Because of the abundance of such objects, a random choice is an ideal way of creating objects having some desired properties. In fact, one applies this paradigm for creating a witness to each application of the method of abundance of witnesses.

The paradigm of “amplification” says that one can increase the success probability of a randomized algorithm by executing several independent runs of the algorithm on the same input. In fact, we use this paradigm for classifying randomized algorithms with respect to the speed of the error probability reduction in the number of executed runs.

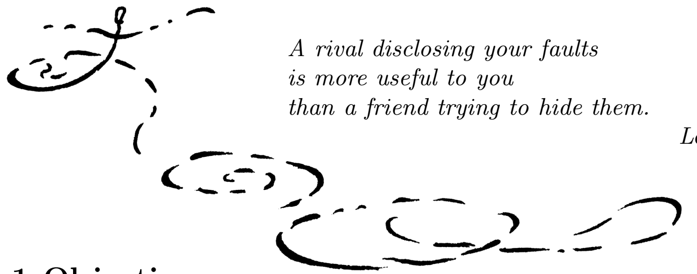
The method of “random rounding” is applied to solving hard optimization problems. First, one relaxes an NP-hard discrete optimization problem to an efficiently solvable problem like linear programming, and then one rounds the real optimal solution of the relaxed problem instance to get a feasible solution of the original optimization problem.

Karp was the first computer scientist who explicitly formulated the paradigms of the design of randomized algorithms in his milestone article [Kar91]. The most exhaustive source of randomization in algorithmics is the excellent book of Motwani and Raghavan [MR95]. Several parts of this book are too difficult to be read by non-specialists, and so we recommend this source as an advanced course, deepening the knowledge provided in our textbook. A simple introduction to applying randomization for solving hard problems is presented in Chapter 5 of the textbook [Hro03]. Pardalos, Rajasekaran, Reif, and Rolim edited an extensive handbook [PRRR00] on randomized computing with many excellent contributions by leading researchers in this area. The bestseller on algorithmics by Cormen, Leiserson, Rivest, and Stein [CLRS01] (the first edition [CLR90] by Cormen, Leiserson, and Rivest) provides also several nicely presented contributions to the design of randomized algorithms. For further reading we also warmly recommend the textbooks of Brassard and Bradley [BB96], Schöning [Sch01], and Wegener [Weg03] (the last two in German). A fascinating source about the famous PCP Theorem, which provides a fascinating endorsement of the power of randomized computing, is the book edited by Mayr, Prömel, and Steger [MPS98].

An excellent, transparent introduction to probability theory is provided in the German textbook [SS01] by Schickinger and Steger. We also warmly recommend, for the fundamentals of discrete mathematics, the book by Graham, Knuth, and Patashnik [GKP94], the successful textbook by Ross [Ros00], and the introductory German bestseller by Steger [Ste01].

This page intentionally left blank

3 Foiling the Adversary



A rival disclosing your faults
is more useful to you
than a friend trying to hide them.

Leonardo da Vinci

3.1 Objectives

The method of foiling the adversary lies behind all randomized algorithms. The aim of this chapter is to present its applications in situations in which it is crucial in the algorithm design process. This is the case exactly for problems for which

(i) every deterministic algorithm (strategy) has some worst-case instances at which the algorithm is not able to efficiently compute the correct output,
(ii) but there exists a class of deterministic algorithms such that, for every problem instance, most algorithms of this class efficiently compute the correct result.

If (ii) holds, then, for any given input instance, one can pick an algorithm from the class at random, and expect to get the correct result efficiently with reasonable probability. Because of the point of view presented above this method is also called the **method of avoiding the worst-case inputs**. We speak preferably about foiling the adversary because one can view the entire process of algorithm design as a game between the algorithm designer and her/his adversary. The goal of the designer is to design an efficient algorithm for a given problem. The task of the adversary is to investigate any algorithm presented by the designer in order to find instances at which this algorithm behaves poorly. This is a typical game for analyzing algorithms or for proving lower bounds on the amount of computing resources necessary to solve a given problem. In situations where the adversary is successful in constructing hard inputs for any given deterministic algorithm, the designer can try to beat the adversary by designing a randomized algorithm. Then the task of the adversary becomes harder than in the deterministic case. Though the adversary can learn the designed randomized algorithm, she/he does not know which of the possible runs will be chosen at random. And finding an input that is hard for all, or at least for many, runs is a much more complex task than finding a hard problem instance for a specific run.

The art of applying the method of foiling the adversary lies in

finding a suitable set of deterministic algorithms (strategies) such that, for any problem instance, most algorithms efficiently compute the correct result.

To give transparent presentations of successful applications of this method, we consider hashing and online algorithms. In the case of hashing, one can easily observe that, for each hash function, one can find a set of data records (keys), such that the distribution of data in the hash table is very unsatisfactory. This is transparently shown in Section 3.2. In Section 3.3 it is shown that one can find a set H of hash functions (hashing strategies) such that, for each set S of data records, a randomly chosen hash function from H distributes the actual data records of S very well in the hash table with high probability. Section 3.4 introduces the online problems. Solving an online problem, one gets only part of the problem instances, and is required to process it. After the completion of this partial task, the following part of the problem instance is made available, etc. The crucial point is that the decision made while working on an available part of the input must be executed, and may not be changed after reaching subsequent parts. As we will document later, these kinds of tasks are very natural in practice, especially in scheduling and logistics. This is an ideal situation for the adversary who always waits for the decisions of an online algorithm on an input part and, after decisions have failed, creates the next part of the input. Therefore it is not surprising that for many online problems deterministic algorithms do not have any chance of finding a reasonably good[1] solution for the entire task. This can happen even for optimization problems for which deterministic algorithms are able to provide good solutions efficiently, assuming that the entire input instance is available from the very beginning of the computation. Section 3.5 shows that by applying randomization (especially the design paradigm of foiling the adversary) one can guarantee reasonably good solutions to online problems for which no comparably good solution can be assured by deterministic algorithms working in unbounded time.

3.2 Hashing

Hashing is a method of dynamic data management. The considered data records have unique names, and one uses the names to request them. The unique name of a data record is called the **key** (of that data record) and the whole data management is determined by the keys, i.e., the lengths and the contents of the data records do not have any influence on data management. Because of this, we identify data records with their keys, and work only with the keys in what follows.

[1]close to an optimal solution

Dynamic data management consists of executing a sequence of the following three dictionary operations:

- *Search* (T, k): Search for the key (data record) k in the data structure T.
- *Insert* (T, k): Insert the new key (data record) k in the data structure T.
- *Delete* (T, k): Delete the key (data record) k in the data structure T.

Our aim is to find a dynamic data structure such that these three operations can be efficiently performed. Basic courses on data structures and algorithms usually introduce AVL-trees and B-trees. These dynamic data structures enable us to execute any of these three kinds of operations in

$$\Theta(\log n)$$

time, and this holds in the worst case as well as in the average case.

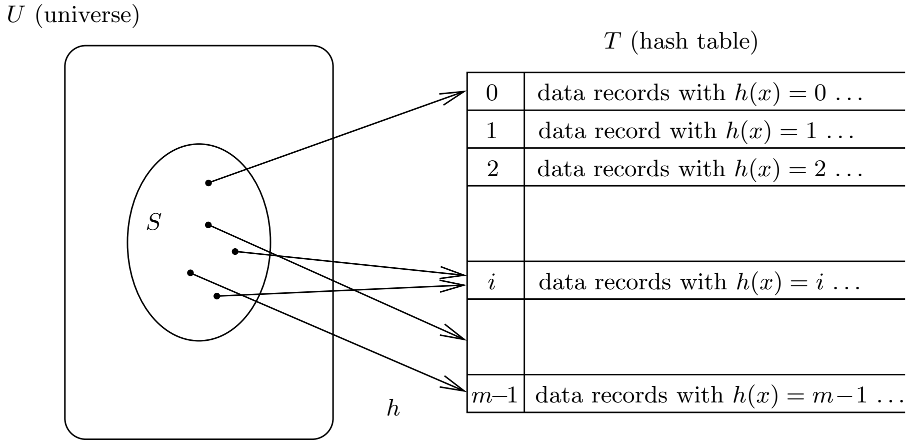

U (universe)
T (hash table)
S
h
0 | data records with $h(x) = 0 \ldots$
1 | data record with $h(x) = 1 \ldots$
2 | data records with $h(x) = 2 \ldots$
i | data records with $h(x) = i \ldots$
m–1 | data records with $h(x) = m-1 \ldots$

Fig. 3.1.

The first goal of hashing is to reduce the expected complexity to $O(1)$. We assume we have memory T with direct access[2] to each memory cell of T. The memory cells are called **slots**, and T is called a **hash table** (Figure 3.1) or **direct address table**. The size $|T|$ is the number of slots of T and we fix the notation

$$|T| = m$$

for a positive integer m for the rest of this chapter. We set

$$T = \{0, 1, 2, \ldots, m-1\},$$

[2]One is able to examine any memory cell (position) of T in time $O(1)$.

where $0, 1, 2, \ldots, m-1$ are the names of the slots. The whole scenario is depicted in Figure 3.1). We have a large set U, called the **universe**, that contains all possible keys.[3] Usually, $|U| \gg |T|$, and so it is impossible to embed the whole universe U in T. But this is not bad because U does not correspond to any actual set of data records (keys), it is only the set of all keys that may occur in an application. Thus, instead of trying to manage U, we aim to save an actual set S of keys (data records) in T. The set S is given by a considered application and its size is usually comparable with $|T| = m$.

In what follows, we always consider U as a large finite subset $\{0, 1, 2, \ldots, d\}$ of $\mathbb{N}$, or even $U = \mathbb{N}$. Our task is to save a set $S \subseteq U$ of actual keys (data records) in T. The main point is that we cannot influence the choice of S, and we do not have any preliminary information about S. The set S is given completely by a considered application (by a user), and we are not able to predict anything about it. Typically, S is approximately as large as T, and we fix

$$|S| = n$$

for the rest of this chapter. The task is to determine a mapping

$$h : U \to T = \{0, 1, \ldots, m-1\}$$

such that the set $S \subseteq U$ is "well dispersed" in T. To be "well dispersed" means that the elements of S are uniformly distributed in the slots $\{0, 1, \ldots, m-1\}$ of T; in the ideal case each slot has approximately

$$\frac{|S|}{|T|} = \frac{n}{m}$$

keys of S. If a slot b has l keys of S, we build a linear list of l keys (data records) and use a pointer from the slot b to this list (as depicted in Figure 3.2) to make this list available. The consequence is that the search for a key in the list costs at most time l, and in the average case time $l/2$. Hence it is clear that the maximal[4] number of keys assigned to a slot is decisive for the complexity of the execution of dictionary operations *Search* (T, k), *Insert* (T, k), and *Delete* (T, k).

Once again we call attention to the fact that we do not have the possibility of choosing h as a mapping from S to T. We have to choose h without any knowledge of S. Moreover, the set S may substantially change during data management by applying the operations *Insert* (T, k) and *Delete* (T, k).

The function h from U to T is called a **hash function**. Since U and T are given, and S is first unknown, and we cannot influence its choice, the only instrument we have in this game is the choice of the hash function h from U to T. We pose the following requirements on h:

(i) h can be efficiently computed,

[3] all possible names of data records

[4] over all $i \in \{0, 1, \ldots, m-1\}$

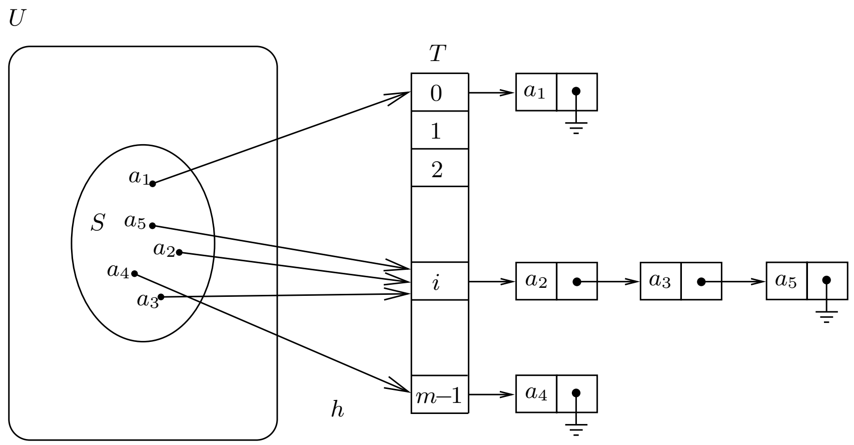


Fig. 3.2.

(ii) h maps most data sets $S \subseteq U$ to T in such a way, that, for all $i \in \{0, 1, \ldots, m-1\}$, the cardinality of the set

$$T(i) = \{a \in S \mid h(a) = i\}$$

is in $O(|S|/|T|)$.

Note, that the choice of the hash function h can also be viewed as an application of fingerprinting.[5] From this point of view, an $h(a) \in \{0, 1, \ldots, m-1\}$ is a fingerprint of a possibly larger key (number) a. The requirement (ii) says besides other things, that the fingerprint $h(a)$ saves so much essential information about a that using $h(a)$ one can distinguish a from almost all other keys in S for most sets S (or that using $h(a)$ one can distinguish a from most other keys in the universe).

It is important to observe that one cannot strengthen the requirement (ii) for the choice of h by requiring a uniform distribution of S in T for every set $S \subseteq U$. For every hash function $h : U \to T$ and every slot $i \in \{0, 1, \ldots, m-1\}$, there exists the set

$$U_{h,i} = \{a \in U \mid h(a) = i\},$$

all of whose elements are mapped by h to the slot i of T. Hence, for any actual key set $S \subseteq U_{h,i}$, or S consisting primarily of elements of $U_{h,i}$, the hash function h does not provide any good distribution of the keys in T.

In order to fulfill at least (ii), one has to have at least a hash function h that uniformly distributes all keys of the universe U in T. This is nothing else than forcing, that, for every $i \in \{0, 1, \ldots, m-1\}$,

$$\text{Prob}(h(x) = i) = \frac{1}{m} \tag{3.1}$$

[5]See Section 2.6 for more details.

for every randomly chosen element[6] x of U. An example of such a hash function is the function $h_m : U \to T$, defined by

$$h_m(x) = x \bmod m.$$

Exercise 3.2.1. Let U be a finite set, $|U| = r \cdot m$ for a positive integer r. Estimate the number of functions $h : U \to T$ that satisfy the property (3.1).

Our first aim is to show that all functions satisfying (3.1) fulfill the requirement (ii), too. Let $\mathcal{P}_n(U) = \{S \subseteq U \mid |S| = n\}$ for any opsitive integer n.

Lemma 3.2.2. *Let $U = \mathbb{N}$ be the universe, and let $T = \{0, 1, \ldots, m-1\}$, $m \in \mathbb{N} - \{0, 1\}$. Let n be a positive integer, and let $h : U \to T$ be a hash function that satisfies (3.1). Then, for every slot l of T,*

(i) the expected number of elements of a random $S \in \mathcal{P}_n(U)$ assigned to the slot l (i.e., the number of elements x of S with $h(x) = l$) is smaller than

$$\frac{n}{m} + 1$$

(ii) and, if $n = m$, then

$$\text{Prob}(\textit{more than one key from a random } S \in \mathcal{P}_n(U) \textit{ is in the } l\textit{-th slot}) < \frac{1}{2}.$$

Proof. We consider the experiment of choosing n elements (keys) of U at once in order to obtain a random set $S \subseteq U$ with $|S| = n$. One can model this experiment by the probability space $(\mathcal{P}_n(U), \text{Prob})$, where

$$\mathcal{P}_n(U) = \{S \subseteq U \mid |S| = n\}$$

and Prob is the uniform probability distribution over $\mathcal{P}_n(U)$. We use the notation $S = \{s_1, s_2, \ldots, s_n\}$, where $s_1, \ldots, s_n$ is the n-tuple of randomly chosen keys from U.

For all $i, j \in \{1, \ldots, n\}$, $i < j$, and all $l \in \{0, 1, \ldots, m-1\}$, we consider the random variable X_{ij}^l, defined by

$$X_{ij}^l(S) = \begin{cases} 1 \text{ if } h(s_i) = h(s_j) = l \\ 0 \text{ else.} \end{cases}$$

If $h(s_i) = h(s_j) = l$, then we say that s_i and s_j have a **collision** in the l-th slot. Since X_{ij}^l is an indicator variable, the expectation $\mathrm{E}\left[X_{ij}^l\right]$ is the probability that $h(s_i) = h(s_j) = l$ (i.e., that both s_i and s_j are assigned to the l-th slot). But this is nothing other than

[6] i.e., the sample space considered here is the universe U, and the probability distribution over U is the uniform one.

$$\begin{aligned}\mathrm{Prob}(h(s_i) = l \wedge h(s_j) = l) &= \mathrm{Prob}(h(s_i) = l) \cdot \mathrm{Prob}(h(s_j) = l) \\ &\underset{(3.1)}{=} \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m^2}. \end{aligned} \tag{3.2}$$

Now, we estimate the expected number of collisions for each slot l in T. The indicator variable

$$X^l = \sum_{1 \le i < j \le n} X_{ij}^l = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}^l$$

counts the number of collisions in the l-th slot. Applying the linearity of expectation and inserting (3.2), we obtain

$$\begin{aligned}\mathrm{E}\left[X^l\right] &= \mathrm{E}\left[\sum_{1 \le i < j \le n} X_{ij}^l\right] \\ &= \sum_{1 \le i < j \le n} \mathrm{E}\left[X_{ij}^l\right] \\ &\underset{(3.2)}{=} \sum_{1 \le i < j \le n} \frac{1}{m^2} \\ &= \frac{\binom{n}{2}}{m^2} = \frac{n(n-1)}{2m^2} < \frac{n^2}{2 \cdot m^2} \end{aligned} \tag{3.3}$$

for every $l \in \{0, 1, \ldots, m-1\}$. For $n = m$, one obtains

$$\mathrm{E}\left[X^l\right] < \frac{1}{2},$$

i.e., the probability that more than one key of S is assigned to the l-th slot of T is smaller than $1/2$.

Next, we use (3.3) to estimate the expected number of keys from S assigned to the l-th slot for an arbitrary $n > m$. If exactly k elements of S are assigned to the l-th slot, then one has exactly

$$\binom{k}{2} = \frac{k \cdot (k-1)}{2}$$

collisions in the l-th slot. Setting $\mathrm{E}\left[X^l\right] = \frac{k \cdot (k-1)}{2}$, one obtains

$$\frac{n^2}{2m^2} > \mathrm{E}\left[X^l\right] \underset{(3.3)}{=} \frac{n \cdot (n-1)}{2 \cdot m^2} = \frac{k \cdot (k-1)}{2} > \frac{(k-1)^2}{2},$$

and so

$$k < \frac{n}{m} + 1.$$

□

Exercise 3.2.3. Assume that the assumptions of Lemma 3.2.2 are satisfied. Estimate the expected number of collisions in all slots of T.

Exercise 3.2.4. Let S be a randomly chosen group of persons. How large has S to be in order to assure

$$\text{Prob}(\text{two persons from } S \text{ were born on the same day}) \geq \frac{1}{2}?$$

To answer this question, consider the indicator variable X defined by

$$X(S) = \begin{cases} 1 \text{ if there are two persons in } S \text{ with the same birthday} \\ 0 \text{ else} \end{cases}$$

and assume that the birthdays are uniformly distributed over the year.

Estimate $\text{E}[X]$ as a function of $|S|$.

Exercise 3.2.5. Assume U, T, and S and X_{ij}^l, and X^l for $1 \leq i < j \leq n$, and $0 \leq l \leq m-1$ have the same meaning as in the proof of Lemma 3.2.2. Define a new random variable $\text{MAX} : \mathcal{P}_n(U) \to \{0,1\}$ by

$$\text{MAX}(S) = \max\{X^0(S), X^1(S), \ldots, X^{m-1}(S)\}.$$

Estimate $\text{E}[\text{MAX}]$ and explain its meaning.

Lemma 3.2.2 assures that the expected[7] complexity of the execution of any dictionary operation is in $O\left(\frac{n}{m}\right)$, and so in $O(1)$ for $n \in O(m)$. This holds because the expected number of keys assigned to any slot of T is less than $\frac{n}{m} + 1$. Remember that the expectation is considered as the average over all possible sets S.

In Lemma 3.2.2 we assumed $U = \mathbb{N}$. If $|U|$ is a finite multiple of $|T|$ and one tries to fix S by n consecutive random choices of an element of U, we have the following problem. Let x with $h(x) = i$ for an $i \in T$ be the first element chosen. If (3.1) holds, then

$$\Big|\{a \in U - \{x\} \mid h(a) = i\}\Big| < \Big|\{b \in U - \{x\} \mid h(b) = j\}\Big| = \frac{|U|}{m}$$

for all $j \in \{0, 1, \ldots, m-1\} - \{i\}$. Hence, the probability of getting an element y with $h(y) = i$ in the random choice of the second element from $U - \{x\}$ is less than $1/m$. If $|U| \gg |S| \geq |T|$, one may take $1/m$ as a good approximation of $\text{Prob}(h(a) = s \mid a \in U - A)$ for every $s \in T$ and every set A with $|A| < |S|$. Another way to overcome this difficulty is considered in the following exercise.

Exercise 3.2.6. Consider the following random generation of $S \subseteq U$. We consider the experiment of n consecutive random choices of elements from U in the way in which each chosen element is returned to U before the next

[7]with respect to a randomly chosen set S of n elements

random choice, i.e., one has n random choices from the complete universe U. In this way one allows the choosing of the same element several times. The result of this random experiment is a set

$$S \in \bigcup_{r=1}^{n} \mathcal{P}_r(U).$$

Estimate the probability of generating a set of cardinality n (i.e., the probability that no element has been chosen more than once). Formulate and prove an assertion similar to Lemma 3.2.2 for a random choice of a set S from a finite universe U.

3.3 Universal Hashing

In the previous section we have shown that

(i) for every hash function h, there exist "bad" sets S (for instance, a set $S \subseteq U_{h,i} = \{a \in U \mid h(a) = i\}$)), and (on the other hand)
(ii) every hash function satisfying (3.1) successfully distributes the keys of a random set S (and so the most subsets of the universe U) in T.

Though the fact (ii) may evoke optimism, one has to note that the choice of data (keys) from the universe is very far from being random according to the uniform distribution. For instance, when the data records of employees of an institute contain a bit for the sex indicator, and almost all persons working in the institute are women, then one cannot model and represent this bit by a random choice. Also, the birthdays are not uniformly distributed over the year, and problems can occur when considering age. Therefore we aim to improve the hashing strategy.

Considering facts (i) and (ii), we have an exemplary situation calling for applying the method of foiling the adversary. For every choice of h (i.e., for every deterministic hashing), (i) says that the adversary can construct arbitrarily bad inputs S. On the other hand (ii) says that there are many hash functions that assure the best possible distribution for most inputs S. Following the idea of foiling the adversary, one has to choose a hash function at random from a set of hash functions such that every input set S is uniformly distributed in T with high probability.

Hence, we need a set H of hash functions from which one can uniformly choose h at random. This means that our probability space is

$$(H, \text{Prob}_H),$$

and H should be created in such a way that (H, Prob_H) is good for every possible S. In searching for H, we follow an analogy to condition (3.1) saying that two randomly chosen keys x and y from U are mapped by h on the same slot with probability at most

$$\frac{1}{m}.$$

Here, we require for *each* pair of keys $x, y \in U$, $x \neq y$, that the set H has the property

$$\text{Prob}_H(h(x) = h(y)) \leq \frac{1}{m}. \tag{3.4}$$

This leads to the following definition:

Definition 3.3.7. *Let H be a finite set of hash functions from U to $T = \{0, 1, \ldots, m-1\}$. The set H is called* **universal** *if for each pair of elements x, y from U, $x \neq y$,*

$$|\{h \in H \mid h(x) = h(y)\}| \leq \frac{|H|}{m}$$

holds, i.e., at most every m-th hash function from H maps x and y to the same slot of T.

We immediately observe that every universal set of hash functions satisfies condition (3.4) in the probability space (H, Prob_H). The next theorem shows that each universal set of hash functions corresponds to our main goal to guarantee a uniform distribution of S in T for every subset S of U.

Theorem 3.3.8. *Let $S \subseteq U$ be an arbitrary set of keys, $|S| = n$. Let H be a universal class of hash functions from U to $T = \{0, 1, \ldots, m-1\}$.*

Then, for every $x \in S$ and a randomly chosen $h \in H$ the expected cardinality of the set $S_x(h) = \{a \in S \mid a \neq x,\ h(a) = h(x)\}$ is at most

$$\frac{|S|}{|T|} = \frac{n}{m}.$$

Proof. Let S be an arbitrary subset of U, with $|S| = n$. Consider the probability space (H, Prob_H). Let, for all $x, y \in S$, $x \neq y$, $Z_{x,y}$ be the indicator variable defined by

$$Z_{x,y}(h) = \begin{cases} 1 \text{ if } h(x) = h(y) \\ 0 \text{ if } h(x) \neq h(y). \end{cases}$$

Since $Z_{x,y}$ is an indicator variable, applying (3.4) we obtain

$$\text{E}[Z_{x,y}] = \text{Prob}_H(h(x) = h(y)) \underset{(3.4)}{\leq} \frac{1}{m}. \tag{3.5}$$

The random variable

$$Z_x = \sum_{y \in S, y \neq x} Z_{x,y}$$

counts the number of elements in $S_x(h)$ for every $x \in S$. Due to the linearity of expectation we obtain

$$\mathrm{E}[Z_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathrm{E}[Z_{xy}] \underset{(3.5)}{\leq} \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{m} = \frac{|S|-1}{m} < \frac{n}{m}$$

for every key $x \in S$. □

The assertion of Theorem 3.3.8 says that the expected number of keys in any slot of T is smaller than $1 + \frac{n}{m}$. Therefore, the concept of universal sets of hash functions is an instrument for solving the problem of dynamic data management, assuming there exist universal sets of hash functions. The next step is to show that we have not introduced an "empty" concept.

Lemma 3.3.9. *Let U be a finite set. The class*

$$H_{U,T} = \{h \mid h : U \to T\}$$

of all functions from U to T is universal.

Proof. The number of all functions in $H_{U,T}$ is[8]

$$m^{|U|},$$

and each of these functions has the same probability of being chosen. Let

$$H(x, y, i) = \{h \mid h : U \to T \text{ and } h(x) = i = h(y)\}$$

for all $x, y \in U$, $x \neq y$, and all $i \in T$. Clearly,[9]

$$|H(x, y, i)| = m^{|U|-2}.$$

What we need is to estimate the cardinality of the set

$$H(x, y) = \{h : U \to T \mid h(x) = h(y)\}.$$

Observe that

$$H(x, y) = \bigcup_{i=0}^{m-1} H(x, y, i)$$

and $H(x, y, i) \cap H(x, y, j) = \emptyset$ for all $i \neq j$, $i, j \in T$. Therefore,

$$|H(x, y)| = \sum_{i=0}^{m-1} |H(x, y, i)| = \sum_{i=0}^{m-1} m^{|U|-2} = m^{|U|-1} = \frac{|H_{U,T}|}{m}$$

for all $x, y \in U$, $x \neq y$. From Definition 3.3.7, $H_{U,T}$ is a universal set of hash functions. □

[8] One has to assign one of the m values from T to each element of U.

[9] The functions in $H(x, y, i)$ have a fixed value i for the arguments x and y, and so $|H(x, y, i)|$ equals the number of functions from $U - \{x, y\}$ to T.

Exercise 3.3.10. Let us consider the set

$$M = \{h : U \to T \mid h \text{ fulfills } (3.1)\}.$$

Is M a universal set of hash functions?

Now, we are sure that the concept of universality is not empty, but unfortunately this does not mean that one has a guarantee of a successful application. There are at least two reasons why we cannot use $H_{U,T}$ in real applications.

(1) $H_{U,T}$ is too large[10] to be able to perform a random choice of an h from $H_{U,T}$ efficiently.
(2) Most functions from $H_{U,T}$ can be viewed as random mappings in the sense[11] that they do not have any essentially shorter description than their full table representation. Such functions are not efficiently computable.

The reasons (1) and (2) explaining why one is not satisfied with $H_{U,T}$ help us formulate what one really wants. We would like to have a universal class H of hash functions such that

(i) H is small (at most polynomial in $|U|$), and
(ii) every hash function from H is very efficiently computable.

In what follows, we show that the above formulated wishes are satisfiable.

Consider $U = \{0, 1, 2, \ldots, p-1\}$ for a prime p. For any natural numbers $a, b \in U$, we define the linear hash function $h_{a,b} : U \to T$ by

$$\boldsymbol{h_{a,b}(x)} = ((ax + b) \bmod p) \bmod m$$

for every $x \in U$. Since we compute modulo m, it is obvious that $h_{a,b}$ really maps elements from U to T. Let

$$H_{\text{lin}}^p = \{h_{a,b} \mid a \in \{1, 2, \ldots, p-1\} \text{ and } b \in \{0, 1, \ldots, p-1\}\}$$

be a set of hash functions. Obviously,

$$|H_{\text{lin}}^p| = p \cdot (p-1).$$

Theorem 3.3.11.* *For any prime p, the class H_{lin}^p is a universal class of hash functions from $U = \{0, \ldots, p-1\}$ to $T = \{0, 1, \ldots, m-1\}$.*

Proof. Let x and y by arbitrary distinct elements from U. We have to show that

$$|\{h_{a,b} \mid h_{a,b} \in H_{\text{lin}}^p \text{ and } h_{a,b}(x) = h_{a,b}(y)\}| \leq \frac{|H_{\text{lin}}^p|}{m} = \frac{p \cdot (p-1)}{m}.$$

[10]Observe that U is already very large, and so $m^{|U|}$ is usually substantially larger than the number of protons in the known universe.

[11]See the concept of Kolmogorov complexity as presented in [Hro03].

To achieve this goal, we start with the investigation of the linear functions

$$\boldsymbol{h'_{a,b}(x)} = (ax + b) \bmod p$$

in $\mathbb{Z}_p$. The algebra $(\mathbb{Z}_p, \oplus_{\bmod p}, \odot_{\bmod p})$ is a field because p is a prime.[12] Let

$$r = h'_{a,b}(x) = (ax + b) \bmod p \text{ and } s = h'_{a,b}(y) = (ay + b) \bmod p \qquad (3.6)$$

be the remainders modulo p after applying the linear mapping $h'_{a,b}$ on x and y. We show in indirect way that $x \neq y$ implies $r \neq s$. Clearly,

$$r - s \equiv ax - ay \equiv a \cdot (x - y) \pmod{p}. \qquad (3.7)$$

If $r = s$, then

$$a \cdot (x - y) \equiv 0 \pmod{p}.$$

Since $c \cdot d = 0$ in a field implies that $c = 0$ or $d = 0$, we obtain that

$$a \equiv 0 \pmod{p} \text{ or } (x - y) \equiv 0 \pmod{p}$$

hold.[13] The element $a \in \{1, \ldots, p-1\}$ is smaller than p, and so cannot be divisible by p. Hence, the only possibility is that $x - y \equiv 0 \pmod{p}$. Since $x, y \in \{1, \ldots, p-1\}$, we obtain $x = y$.

Since we assumed $x \neq y$, r and s must be different. Hence, we have proved that for any fixed different keys $x, y \in U$, every[14] function $h'_{a,b}$ unambiguously determines a pair (r, s) from $U \times U$, with $r \neq s$. In other words, we have proved that the function $f_{x,y}$, defined by

$$f_{x,y}(a, b) = ((ax + b) \bmod p,\ (ay + b) \bmod p) = (h'_{a,b}(x), h'_{a,b}(y)),$$

is a mapping from $(U - \{0\}) \times U$ to $\{(r, s) \mid r, s \in U, r \neq s\}$ (Figure 3.3).

We claim that $f_{x,y}$ is a bijection.[15] Since

$$|(U - \{0\}) \times U| = (p - 1) \cdot p = |\{(r, s) \mid r, s \in U, r \neq s\}|,$$

it is sufficient to show that one of the functions $f_{x,y}$ or $f^{-1}_{x,y}$ is injective.

We show that $f^{-1}_{x,y}$ is injective. Let r and s belong to U, $r \neq s$. Since one computes in a field $\mathbb{Z}_p$, the linear equalities (3.6) for known values of x, y, r, and s have the following unique solution in $\mathbb{Z}_p$:

$$a \underset{(3.7)}{=} (r - s) \cdot (x - y)^{-1} \bmod p$$

[12] $\mathbb{Z}_p$ is a field if and only if p is a prime. This assertion is a direct consequence of Theorem A.2.27.

[13] In other words, if a prime p divides $a \cdot (x - y)$, then, following the Fundamental Theorem of Arithmetics (Theorem A.2.3), p must divide a or $(x - y)$.

[14] $h'_{a,b}$ for all $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$

[15] i.e., $f_{x,y}$ and $f^{-1}_{x,y}$ are injective functions

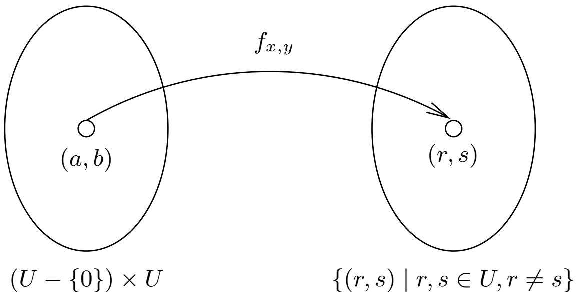


Fig. 3.3.

and

$$b = (r - ax) \bmod p,$$

where $(x-y)^{-1}$ is the unique inverse of the element $x-y$ with respect to $\odot_{\bmod p}$ in $\mathbb{Z}_p$. Therefore the mapping $f_{x,y}^{-1}$ defined by

$$f_{x,y}^{-1}(r,s) = (a,b) = ((r-s)\cdot(x-y)^{-1} \bmod p, (r-ax) \bmod p)$$

is an injective function, too.

Since the random choice of $h_{a,b}$ (i.e., the random choice of (a,b)) unambiguously determines the pair

$$(r,s) = (h'_{a,b}(x), h'_{a,b}(y)) = ((ax+b) \bmod p, (ay+b) \bmod p),$$

the number of hash functions in

$$H^p_{\text{lin}}(x,y) = \{h_{a,b} \in H^p_{\text{lin}} \mid h_{a,b}(x) = h_{a,b}(y)\}$$

is equal to the cardinality of the set

$$M(x,y) = \{(r,s) \in U \times U \mid r \neq s \text{ and } r \equiv s \pmod m\},$$

and so

$$|H^p_{\text{lin}}(x,y)| = |M(x,y)|. \tag{3.8}$$

The number of elements in U is p, and U can be partitioned into the remainder classes modulo m. Each of these remainder classes has at most

$$\left\lceil \frac{p}{m} \right\rceil \leq \frac{(p+m-1)}{m} = \frac{(p-1)}{m} + 1$$

elements. This means that, for every fixed r, there are at most

$$\frac{(p-1)}{m}$$

elements s in U, with

$$r \equiv s \pmod m \text{ and } r \neq s.$$

In this way, we obtain[16]

[16]The element r can be chosen from p elements of U.

$$|M(x,y)| \leq \frac{p \cdot (p-1)}{m} \tag{3.9}$$

for all pairs $(x,y) \in U \times U$, with $x \neq y$, and so

$$|H^p_{\text{lin}}(x,y)| \underset{(3.8)}{=} |M(x,y)| \underset{(3.9)}{\leq} \frac{p \cdot (p-1)}{m} = \frac{|H^p_{\text{lin}}|}{m}$$

for all $(x,y) \in U$, with $x \neq y$. □

Exercise 3.3.12. In Theorem 3.3.11 we have chosen $U = \{0, 1, \ldots, p-1\}$ for a prime p. Assume that $U = \{0, 1, \ldots, l\}$ for an arbitrary positive integer l, and choose a prime p such that $p > l$. Is the set H^p_{lin} still a universal set of hash functions?

Next, we give another useful example of a universal set of hash functions. For this purpose we consider another appropriate representation of keys. For every prime m and every positive integer r, we define

$$U(m,r) = \{x = (x_0, x_1, \ldots, x_r) \mid 0 \leq x_i \leq m-1 \text{ for } i = 1, \ldots, r\} = T^{r+1}$$

for $T = \{0, 1, \ldots, m-1\}$.

For every vector $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_r) \in T^{r+1}$, we define the hash function $h_\alpha : U(m,r) \to T$ by

$$h_\alpha(x_0, x_1, \ldots, x_r) = \left(\sum_{i=0}^{r} \alpha_i \cdot x_i \right) \bmod m$$

for all $(x_0, x_1, \ldots, x_r) \in U(m,r)$. Let

$$\text{Vec} = \{h_\alpha \mid \alpha \in T^{r+1}\}.$$

Lemma 3.3.13. Vec *is a universal set of hash functions from the universe* $U(m,r)$ *to* T.

Proof. Since $|T^{r+1}| = m^{r+1}$,

$$|\text{Vec}| = m^{r+1}.$$

Let $x = (x_0, x_1, \ldots, x_r)$ and $y = (y_0, y_1, \ldots, y_r)$ be two arbitrary different elements from $U(m,r)$. We have to show that the number of hash functions h_α from Vec with the property $h_\alpha(x) = h_\alpha(y)$ is at most

$$m^r = \frac{|\text{Vec}|}{m}.$$

Since the vectors x and y are different, they differ in at least one position. To simplify the notation we assume without loss of generality that $x_0 \neq y_0$.

The following holds:

$$h_\alpha(x) = h_\alpha(y)$$

for an $\alpha \in T^{r+1}$ if and only if

$$\sum_{i=0}^{r} \alpha_i \cdot x_i \equiv \sum_{i=0}^{r} \alpha_i \cdot y_i \pmod{m},$$

which is equivalent to

$$\alpha_0 \cdot (x_0 - y_0) \equiv \sum_{i=1}^{r} \alpha_i \cdot (y_i - x_i) \pmod{m}. \tag{3.10}$$

Since m is a prime, the algebra $(\mathbb{Z}_m, \oplus_{\bmod m}, \odot_{\bmod m})$ is a field. Hence, for every element from $\mathbb{Z}_m - \{0\}$, there exists a unique inverse with respect to $\odot_{\bmod p}$. Since $x_0 - y_0 \neq 0$, there exists an inverse $(x_0 - y_0)^{-1}$ for $x_0 - y_0$. Multiplying both sides of the equality (3.10) by $(x_0 - y_0)^{-1}$, one obtains

$$\alpha_0 \equiv \left[\sum_{i=1}^{r} \alpha_i \cdot (y_i - x_i)\right] \cdot (x_0 - y_0)^{-1} \pmod{m}.$$

In this way α_0 is unambiguously determined by x, y and $\alpha_1, \alpha_2, \ldots, \alpha_r$, and

$$|\{h_\alpha \in \text{Vec} \mid h_\alpha(x) = h_\alpha(y)\}| \leq m^r = \frac{m^{r+1}}{m} = \frac{|\text{Vec}|}{m}.$$

□

We observe that Vec is an appropriate set of hash functions because of the following.

(i) Every hash function h_α from Vec is completely determined by α, and α corresponds exactly to the number of keys in $U(m, r)$. Hence, h_α can be efficiently generated at random.
(ii) The function h_α is efficiently computable for every α, namely in linear time with respect to the length of its argument.

Exercise 3.3.14.* Our last example of a universal class of hash functions is based on the fact that $T = \{0, 1, \ldots, m-1\}$ for a prime m. Consider $m = p^a$ for a prime p and a positive integer $a \geq 2$. For such an m, does there exist a universal class of hash functions?

3.4 Online Algorithms

The computing problems considered until now are the classical tasks for which, for any given input (question), one has to compute an output (answer). But in practice one also has to deal with the following tasks. One obtains only part of the input, and is forced to process this part. After one has solved it, one gets another part of the input that also has to be immediately processed. The input may be arbitrarily long. These kinds of tasks are called **online problems**, and the algorithms solving online problems are called **online algorithms**. The fundamental question posed in this framework is the following:

> *How good can an online algorithm (that does not know the future*[17]*) be in comparison to an algorithm that knows the whole input (the future) from the very beginning?*

What the adjective "good" in this context means depends on the measure used to estimate the quality of produced solutions. Hence, one has a similar situation when dealing with optimization problems. We simply compare the costs of solutions computed by an online algorithm with the costs of corresponding optimal solutions.

In order to highlight the interest in online problems, we present two simple examples. Consider a hospital, with surgeons, each with an ambulance. In the case of a medical call, a surgeon has to drive to the patient's flat or place of accident to help. After some time, some surgeons will be distributed in the city and some may still be in the hospital. In the case of another call, the hospital center has to decide which surgeon has to drive to the new patient.[18] Clearly, the center has to take this decision without knowledge of the future (sources and the order of forthcoming calls). Though the center aims to make decisions that optimize some cost measure. For example, one can try to minimize the sum of the lengths of all distances driven by the ambulances, or the sum of waiting times of the patients, or the maximal waiting time of a patient, or a cost measure that appropriately combines several simple cost measures. A similar type of tasks can be considered by a police station, or by a taxi company, or by many other customer services. The most challenging problem is determining whether or not one has a real chance of making reasonable decisions with respect to the unknown future, i.e., whether or not there exists an online strategy that produces decisions that are not too poor in comparison to results produced by an optimal algorithm that knows the future (the whole input).

Another class of frequently occurring online problems is the class of scheduling problems. Typically, one considers a factory with a number of different machines. From time to time, requests come in that can be viewed as a sequence of some tasks. The execution of every task requires the reservation of a machine type for a certain period of time. The factory has to decide about the scheduling of the machines, i.e., about assigning machines for some time intervals to the tasks. Usually, one aims to minimize the makespan that is the time of completion of all tasks. One can also be interested in maximizing the average load of the machines or in minimizing the average waiting time

[17]the input parts that are still not available

[18]for instance, the surgeon who is closest to the new patient

of all customers. Tasks of this type appear not only in factories, but also in the hardware management of a computer, especially of a parallel computer or an interconnection network. The running processes continuously request some hardware resources, such as the CPU, data storage, communication channels, memory, etc. The computing system has to determine the scheduling of computer hardware without any knowledge of future requests.

In what follows we define the competitive ratio of online algorithms in order to get a reasonable measure of their quality.

Definition 3.4.15. *Let* $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ *be an optimization problem, that can be viewed as an online problem.*[19] *An algorithm* A *is an online algorithm for* U *if, for every input* $x = x_1x_2\dots x_n \in L$*, the following conditions are satisfied:*

(i) For all $i \in \{1, \dots, n\}$, $x_1x_2\dots x_i$ *is a feasible input.*
(ii) $A(x) \in \mathcal{M}(x)$*, i.e.,* A *always computes a feasible solution.*
(iii) For all $i \in \{1, \dots, n\}$, $A(x_1x_2\dots x_i)$ *is part*[20] *of* $A(x)$*, i.e., the decisions made for the prefix* $x_1x_2\dots x_i$ *of* x *cannot be changed any more.*

For every input $x \in L$*, the* **competitive ratio** $\mathbf{comp}_A(x)$ **of** A **on** x *is the number*

$$\mathbf{comp}_{\boldsymbol{A}}(\boldsymbol{x}) = \max\left\{\frac{\text{Opt}_U(x)}{\text{cost}_A(x)}, \frac{\text{cost}_A(x)}{\text{Opt}_U(x)}\right\},$$

where $\text{Opt}_U(x)$ *denotes the cost of an optimal solution for the instance* x *of the problem* U*.*

Let $\delta \geq 1$ *be a real number. We say that* A *is a* **δ-competitive algorithm** *for* U *if*

$$\text{comp}_A(x) \leq \delta$$

for all $x \in L$*.*

Definition 3.4.16. *Let* $\delta > 1$ *be a real number. We say that an online problem* U *is* **δ-hard** *if there does not exist any* d*-competitive online algorithm for* U *with* $d < \delta$*.*

Note that, in contrast with approximation algorithms, we do not take care on the complexity of online algorithms. We focus primarily on comparing what is achievable without knowing the future and with full knowledge of the future.

[19]For instance, an optimization problem can be viewed as an online problem, when each prefix y of every input x can be viewed as a problem instance, and one is required to provide a solution for y that has to remain unchanged as part of the solution for the whole input x. We say "for instance" because sometimes it is reasonable to view inputs as two dimensional objects, and then the notion of a prefix is not unambiguous.

[20]To avoid a too complex, opaque formalism, we try not to formally specify what the term "part" means for the formal output representation.

Example 3.4.17. Here, we introduce the so-called **paging** problem. Let us consider a computer with fast, small direct access memory called **Cache** and large, slow main memory called **Main**. The size of the Cache is bounded by k pages of data, and the Main contains all the data. One has fast, direct access only on data in the Cache. If one wants to access data that is not in the Cache, the corresponding pages have first to be loaded from Main into the Cache. If the Cache is full and one needs to place new pages there, then first some old pages have to be removed from the Cache in order to create free space. Removing data (pages) from the Cache can be performed by sending the corresponding pages back to Main, or simply by erasing them. In what follows, we consider direct access to the Cache for free and for a cost of 1 for the exchange of two pages between the Cache and Main (or for erasing a page in the Cache and moving a new page from Main to the Cache).

Let us consider an initial situation, where the first k pages $s_1, s_2, \ldots, s_k$ reside in the Cache, and Main contains the remaining pages $s_{k+1}, s_{k+2}, \ldots, s_n$ for an $n \gg k$. An instance of the paging problem is a sequence $i_1, i_2, \ldots, i_m$ of integers from $\{1, 2, \ldots, n\}$ that determines the requirement to read the pages $s_{i_1}, s_{i_2}, \ldots, s_{i_m}$ in the given order. In the case of the online version of the problem, one at first gets only i_1, without any knowledge of the forthcoming requests. If s_{i_1} is not in the Cache, then a page has to be removed from the Cache, and the page s_{i_1} has to be loaded from Main to the Cache.

Hence, the online algorithm has to decide which page of the Cache has to be removed without knowing which pages will be required later. This is an ideal situation for the adversary, who can be very mean by simply requesting exactly the removed page in the following step.

We aim to show that there is no d-competitive online algorithm for the paging problem for any $d < k$, i.e., that the paging problem is k-hard for caches of size k. To create a hard input for a given online algorithm, we use the mean adversary.

Let A be an arbitrary online algorithm solving the paging problem for a Cache of size k. The adversary starts with request $k+1$. Since the Cache contains the pages $s_1, s_2, \ldots, s_k$, A must exchange a page for s_{k+1}. Assume that A decides to remove the page s_{j_1} for a $j_1 \in \{1, \ldots, k\}$. Then, the next request of the mean adversary is j_1. Now A has to load the page s_{j_1} back to the Cache and remove a page s_{j_2} for a $j_2 \in \{1, \ldots, k+1\} - \{j_1\}$. Next, the adversary requests the last removed page, s_{j_2}, and so on. In this way the input

$$x_A = k+1, j_1, j_2, \ldots, j_{k-1}$$

is created. The input x_A is hard for A, because A has to perform k exchange operations between the Cache and Main during the processing of x_A, i.e., $\text{cost}(A(x_A)) = k$.

Now, we show that there is a solution for the input x_A that forces only one page exchange. Let i be a number from $\{1, 2, \ldots, k\} - \{j_1, j_2, \ldots, j_{k-1}\}$. The optimal strategy for x_A is to exchange s_i for s_{k+1} in the Cache in the first step. After that, the Cache contains all the pages, $s_{j_1}, s_{j_2}, \ldots, s_{j_{k-1}}$, that are required in the next $k-1$ steps. Thus, we obtain

$$\mathrm{comp}_A(x_A) = \frac{\mathrm{cost}(A(x_A))}{\mathrm{Opt}(x_A)} = \frac{k}{1} = k$$

for every deterministic online algorithm A for the paging problem, and so the paging problem is k-hard[21] (Cache-size hard). $\square$

Exercise 3.4.18. Let A be an arbitrary online algorithm for the paging problem. Find infinitely many inputs y_A of the paging problem with the property $\mathrm{comp}_A(y_A) \geq k$.

Hence, the first fundamental question related to online problems is the question of whether or not there exists a possibility of solving them with a reasonable competitive ratio. If this is possible, then one can start to look for an efficient online algorithm for the online problem considered. Example 3.4.17 shows that there are online problems that cannot be reasonably solved by deterministic online strategies. In such cases one can pose the question of whether randomization can help. The following section is devoted to the attempt of answering this question for a specific scheduling problem.

3.5 Randomized Online Algorithms

As observed in Section 3.4, exactly the online formulations of a computing task provide the adversary the ability to construct hard problem instances for each strategy. Similarly, as in the case of hashing, the randomization can be very helpful. Though the adversary knows the designed randomized algorithm, it does not know which random decision will be taken. In other words, if one has sufficiently many suitable deterministic strategies for every problem instance, then, for any given input, it suffices to randomly choose a deterministic strategy. This way one can get a reasonable expected competitive ratio for each input. Then, there are no hard instances for the randomized algorithm, and so the adversary does not have any chance of beating the randomized strategy.

In what follows, we first define the concept of randomized online algorithms[22] and then document its power by investigating a special online problem.

[21] Note, that it cannot be worse, because k is the input length.

[22] Again we restrict ourselves to one-dimensional input representations of considered problems, because taking multidimensional inputs (which may be natural for some problems) makes the formalism of the following definition too complex, and hardly understandable. For problems with a natural two-dimensional input representation, one can usually easily recognize which parts of the input can be considered as meaningfull prefixes.

Definition 3.5.19. *Let $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$ be an online optimization problem. A randomized algorithm A is a* **randomized online algorithm for U** *if, for every input $x_1 x_2 \ldots x_n \in L$ and every $i \in \{1, \ldots, n-1\}$,*

(i) the output of every run of A on $x_1 x_2 \ldots x_i$ is a feasible solution for the instance $x_1 x_2 \ldots x_i$ of the problem U, and
(ii) for every input $(C(x_1 \ldots x_i), x_{i+1})$, where $C(x_1 \ldots x_i)$ is a feasible solution for the instance $x_1 \ldots x_i$ of U (i.e., $C(x_1 \ldots x_i) \in \mathcal{M}(x_1 \ldots x_i)$), all runs of A compute a feasible solution for the instance $x_1 x_2 \ldots x_{i+1}$ of U and all these feasible solutions involve $C(x_1 \ldots x_i)$ as a partial solution.

For every problem instance x, let $S_{A,x}$ be the set of all computations of A on x, and let $\text{Prob}_{A,x}$ be the corresponding probability distribution on $S_{A,x}$. Let Z_x be the random variable in $(S_{A,x}, \text{Prob}_{A,x})$ defined by

$$Z_x(C) = \text{comp}_C(x)$$

for each computation $C \in S_{A,x}$. We define the **expected competitive ratio of A on x** *as*

$$\textbf{Exp-Comp}_{\boldsymbol{A}}\boldsymbol{(x)} = \mathrm{E}[Z_x]\,.$$

Let δ be a real number. We say, that the randomized algorithm A is a **E [$\boldsymbol{\delta}$]-competitive** *algorithm for U if*

$$\text{Exp-Comp}_A(x) \leq \delta$$

for every $x \in L$.

Let $h : \mathbb{N} \to \mathbb{R}^{\geq 1}$ be a function. We say that A is an **Exp ($\boldsymbol{h}$)-competitive** *algorithm for U if*

$$\text{Exp-Comp}_A(x) \leq h(|x|)$$

for each $x \in L$.

Our next aim is to show that there is an online problem such that applying randomization one can get a competitive ratio approaching 1 with input length, but this approximation ratio is not achievable by any deterministic online strategy.

We consider the following scheduling problem. There are m different machine types, $M_1, M_2, \ldots, M_m$, one piece of each type being available. Each feasible job consists of m tasks, $A_1, A_2, \ldots, A_m$, and can be represented by a permutation $(i_1, i_2, \ldots, i_m)$ of $(1, 2, \ldots, m)$. The meaning of job $(i_1, i_2, \ldots, i_m)$ is as follows:

(i) The tasks must be performed in the order $A_1, A_2, \ldots, A_m$ (one cannot start with work on A_{i+1} before work on A_i has finished).
(ii) For every $j \in \{1, 2, \ldots, m\}$, task A_j must be performed on machine M_{i_j}.
(iii) The execution of a task on its machine costs exactly one time unit.

If one has m pairwise different machines and d tasks, we speak of the so-called Unit (m,d) Job problem. For every positive integer d, we denote by Unit (d) Job the problem

$$\bigcup_{m=1}^{\infty} \text{Unit}\,(m,d)\,\text{Job}.$$

A feasible solution of an instance of Unit (m,d) Job corresponds to a distribution of the tasks to the machines in discrete time. More precisely, a machine and a time unit are assigned to every task of every job in such a way that in each time unit each machine is assigned to at most one task. The cost of a feasible solution is the number of time units used in this solution in order to completely execute all jobs.

In what follows, we consider the simplest Unit (2) Job version of this scheduling problem. To present our considerations in a transparent way, we develop a geometric representation of the problem instances. Let

$$\alpha = (i_1, \ldots, i_m) \text{ and } \beta = (j_1, \ldots, j_m)$$

be an instance of Unit $(m,2)$ Job for a positive integer m. We consider an $m \times m$ grid, $\text{Grid}_m(\alpha,\beta)$, in which, for all integers $k,l \in \{1,\ldots,m\}$, the k-th row is labeled j_k and the l-th column is labeled i_l. The square (cell) (k,l) is the intersection of the row i_k with the column j_l. The square (k,l) is called an **obstacle** (or a **collision**) iff $i_l = j_k$, i.e., iff the same machine is asked to perform the i_l-th task of δ and the j_k-th task of β. These squares are depicted in Figure 3.4(a) as hatched squares. Figure 3.4(a) shows $\text{Grid}_9(\alpha,\beta)$ for $\alpha = (1,2,3,4,5,6,7,8,9)$ and $\beta = (1,3,2,6,5,4,8,7,9)$. The motivation to introduce the term obstacle (collision) is as follows. Assume that the executions of the first $l-1$ tasks of job α and of the first $k-1$ tasks of job β have finished. Now, if $i_l \neq j_k$, one can continue by executing the l-th task of α on the i_l-th machine M_{i_l} and the k-th task of β on the j_k-th machine M_{j_k} in parallel.

But, if

$$i_l = j_k,$$

then both tasks require the same machine M_{i_l}. Therefore, one of these tasks must be postponed until the other one is finished on M_{i_l}.

We assign the graph $G_m(\alpha,\beta) = (V,E)$ to any grid $\text{Grid}_m(\alpha,\beta)$ as follows:

(i) V consists of the vertices of the grid $\text{Grid}_m(\alpha,\beta)$, and
(ii) E contains
- all vertical and all horizontal edges of $\text{Grid}_m(\alpha,\beta)$, and
- all **diagonal** edges of the grid squares between the upperleft corner and the lower-right corner of the square for the squares that are not obstacles.

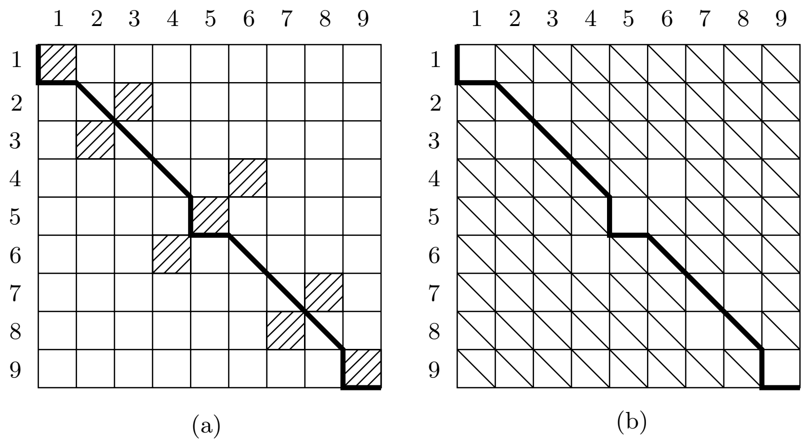


Fig. 3.4.

Figure 3.5(b) shows the graph $G_9(\alpha, \beta)$ for the Grid$_9(\alpha, \beta)$. We observe that every feasible solution for an instance (α, β) corresponds to a path in $G_m(\alpha, \beta)$ that leads from the upper left corner of the grid to the lower right corner of the grid, and uses only edges going to the right or down. Each edge of this path corresponds to one time unit. If the edge is diagonal, it means that both jobs are executed in parallel, and so none of them is delayed. A vertical edge corresponds to the situation where the first job is not executed (i.e., the first job is delayed), and so only the second job is performed in this time unit. In this case we say that the first job has got a **delay**. Analogously, a horizontal edge corresponds to the situation in which only the first job is executed and the second one is delayed.

Obviously, an optimal solution corresponds to the shortest path from the upper left corner to the lower right corner of the graph. The bold edges in Figure 3.5(a) and Figure 3.5(b) build a path that represents an optimal solution for the problem instance considered. In this solution there are 6 delays, that are uniformly distributed over both jobs. Hence, the cost of this optimal solution is

$$m + 6/2 = 9 + 3 = 12.$$

The geometric (graphic) representation of problem instances teaches us that the optimization problem Unit (2) Job is efficiently solvable by computing the shortest path between the two corners of the graph $G_m(\alpha, \beta)$. But we are interested in the online version of this problem. Here, one obtains only the first elements of the permutations (the first tasks of the jobs) and the next element will not be known before all its previous tasks in this job have been performed. To analyze this online problem, we introduce the following terminology.

Let (α, β) be an instance of Unit $(m, 2)$ Job for a positive integer m. Let S be a path[23] in $G_m(\alpha, \beta)$. The number $\mathbf{del}_{\boldsymbol{\alpha}}\boldsymbol{(S)}$ of the vertical edges in S is called the **delay of the first job $\boldsymbol{\alpha}$ in $\boldsymbol{S}$** and the number $\mathbf{del}_{\boldsymbol{\beta}}\boldsymbol{(S)}$ of the horizontal edges in S is called the **delay of the second job $\boldsymbol{\beta}$ in $\boldsymbol{S}$**. The **delay of $\boldsymbol{S}$**, for short **delay $\boldsymbol{(S)}$**, is the maximum of $\mathrm{del}_\alpha(S)$ and $\mathrm{del}_\beta(S)$. If S corresponds to a finite feasible solution for (α, β), then clearly[24]

$$\mathrm{delay}\,(S) = \mathrm{del}_\alpha(S) = \mathrm{del}_\beta(S)\,,$$

and so

$$\mathrm{cost}(S) = m + \mathrm{delay}\,(S) = m + \frac{\mathrm{del}_\alpha(S) + \mathrm{del}_\beta(S)}{2}. \tag{3.11}$$

Let $\mathrm{Opt}_{\mathrm{Job}}(\alpha, \beta)$ denote the cost of the optimal solutions for the problem instance (α, β).

In what follows, we show that using deterministic online algorithms one cannot guarantee solutions with costs arbitrarily close to the optimal cost. We start with a simple observation.

Observation 3.5.20. Let m be a positive integer. Every instance (α, β) of Unit $(m, 2)$ Job contains exactly m conflicts, and each column and each row of the grid $\mathrm{Grid}_m(\alpha, \beta)$ contains exactly one conflict.

Proof. Every integer $k \in \{1, 2, \ldots, m\}$ appears exactly once in each of the permutations α and β.

Lemma 3.5.21. *Let m be a positive integer divisible by 8 ($m \bmod 8 = 0$). For every online algorithm A for the* Unit $(m, 2)$ Job *problem, there exists an instance $I = ((1, 2, \ldots, m), \beta)$ such that*

$$\mathrm{cost}(A(I)) \geq m + \frac{m}{8}.$$

Proof. Let A be an arbitrary online algorithm for the Unit (2) Job problem. We aim to prove Lemma 3.5.21 by showing that the adversary can place the obstacles (i.e., can choose β) in such a way that at least half the edges reached before column $\left(\frac{m}{2} + 1\right)$ or row $\left(\frac{m}{2} + 1\right)$ are not diagonal edges. This would mean that

$$\mathrm{del}_{(1,2,\ldots,m)}(A(I)) + \mathrm{del}_\beta(A(I)) \geq \frac{m}{4},$$

and so (from (3.11))

$$\mathrm{delay}\,(A(I)) \geq \frac{m}{8}.$$

The permutation β starts with 1 in order to place a conflict (an obstacle) in the square $(1, 1)$. Hence, A cannot start with a diagonal edge. After A has used a diagonal edge, the path has always reached, for the first time,

[23] a schedule for the input (α, β)

[24] since S leads from the upper left corner to the lower right corner of a square

a new row and a new column. The adversary places the next obstacle on the square at the intersection of this column and this row. The adversary can work in this way until half the jobs have been executed, because it can use elements $1, 2, \ldots, m/2$ for β for its purpose. If A makes vertical steps,[25] then the adversary uses elements $m, m-1, m-2, \ldots, m/2+1$ for β at the corresponding positions. $\square$

The following lemma shows that every instance of Unit $(m, 2)$ Job can be solved in $m + \sqrt{m}$ time units, and so that the Unit (2) Job problem is $\left(1 + \frac{1}{8} - \varepsilon\right)$-hard for any $\varepsilon > 0$.

Lemma 3.5.22. *Let m be a positive integer. For every instance I of the* Unit $(m, 2)$ Job *problem,*

$$\mathrm{Opt}_{\mathrm{Job}}(I) \leq m + \left\lceil \sqrt{m} \right\rceil .$$

Proof. To simplify the proof, we show it for $m = k^2$ only. Let I be an arbitrary instance of Unit $(m, 2)$ Job. We visualize the proof idea by our graphic representation. For every $i = 0, 1, \ldots, \sqrt{m}$ we denote by D_i the i-th diagonal of the grid $\mathrm{Grid}_m(I)$ that goes from position[26] $(0, i)$ to position $(m - i, m)$, and we denote by D_{-i} the diagonal from position $(i, 0)$ to position $(m, m - i)$, as depicted in Figure 3.5.

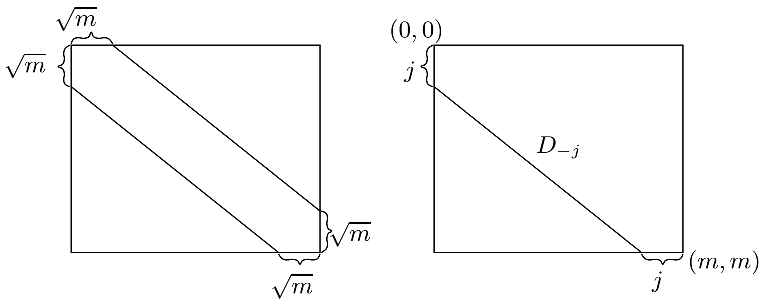


Fig. 3.5.

For every $i \in \{-\sqrt{m}, -\sqrt{m}+1, \ldots, 0, 1, \ldots, \sqrt{m}\}$ we consider the following deterministic strategy, $A(D_i)$, to assign the machines. The strategy $A(D_i)$ first uses i (horizontal[27]/vertical[28]) grid edges in order to reach the starting position of the diagonal D_i. Then $A(D_i)$ runs via the diagonal edges of the diagonal D_i. If this path hits an obstacle, then $A(D_i)$ goes around it by taking

[25] independently of the content of β

[26] The upper left corner vertex is at position $(0, 0)$.

[27] if i is nonnegative

[28] if i is negative

one horizontal and one vertical edge. When $A(D_i)$ has reached the endpoint of the diagonal D_i, then it runs via the i (vertical/horizontal) grid edges to the corner (m, m).

The cost of the solution achieved by this strategy is exactly

$$m + i + \text{ the number of obstacles on } D_i,$$

because the length[29] of D_i is exactly $m - i$, and $A(D_i)$ needs i steps to reach the starting point of D_i and i steps to reach the corner (m, m) from the end of the diagonal. Hence,

$$i + \text{ the number of obstacles on } D_i$$

is the delay of the solution computed by $A(D_i)$.

Since the number of all obstacles is exactly m (Observation 3.5.20), the sum of all delays over all $2 \cdot \sqrt{m} + 1$ solutions computed by the diagonal strategies is at most

$$m + \sum_{i=-\sqrt{m}}^{\sqrt{m}} |i| = m + 2 \cdot \sum_{i=1}^{\sqrt{m}} i = m + \sqrt{m} \cdot (\sqrt{m} + 1).$$

Because we take $2 \cdot \sqrt{m} + 1$ solutions into account, the average delay over all solutions is

$$\frac{m + \sqrt{m} \cdot (\sqrt{m} + 1)}{2 \cdot \sqrt{m} + 1} \leq \sqrt{m} + \frac{1}{2}.$$

Thus, there exists an $i \in \{-\sqrt{m}, \ldots, \sqrt{m}\}$ such that the solution computed by the strategy $A(D_i)$ has at most $\sqrt{m}$ delays, and so the cost of this solution is at most $m + \sqrt{m}$. □

Lemma 3.5.21 and 3.5.22 together imply the following theorem.

Theorem 3.5.23. *For every real $\varepsilon > 0$, the* Unit (2) Job *problem is $\left(\frac{9}{8} - \varepsilon\right)$-hard.*

Proof. From Lemma 3.5.21 we have that, for each deterministic online algorithm A, there exists an input instance I such that

$$\text{cost}(A(I)) \geq \frac{9}{8} \cdot m.$$

From Lemma 3.5.22, we have

$$\text{Opt}_{\text{Job}}(I) \leq m + \sqrt{m}.$$

Hence,

$$\text{comp}_A(I) = \frac{\text{cost}(A(I))}{\text{Opt}_{\text{Job}}(I)} \leq \frac{\frac{9}{8} \cdot m}{m + \lceil \sqrt{m} \rceil} \leq \frac{9}{8} \cdot \frac{1}{1 + \frac{1}{\sqrt{m}}}.$$

□

[29]the number of diagonal edges

Exercise 3.5.24. Let k be a positive integer, Show that, for every $m = \binom{k+1}{2}$, there exists an instance I of Unit $(m, 2)$ Job such that

(i) $\mathrm{Opt}_{\mathrm{Job}}(I) \geq m + \sqrt{\frac{m}{2}} - \frac{1}{2}$, and
(ii)* $\mathrm{Opt}_{\mathrm{Job}}(I) \geq m + \sqrt{m}$.

Though the Unit (2) Job problem can be solved in a simple and efficient way, it cannot be solved optimally, or almost optimally, by any deterministic online algorithm. This claim is independent of the complexity of online algorithms. But the proof of Lemma 3.5.22 already provides an idea as to how we can expect to approach optimal costs. Let us simply consider a randomized online algorithm as a uniform probability distribution over the $2 \cdot \sqrt{m} + 1$ diagonal strategies $A(D_i)$.

Algorithm DIAG

Input: An instance I of Unit $(m, 2)$ Job for a positive integer m.
Step 1: Choose uniformly an $i \in \{-\lceil\sqrt{m}\rceil, -\lceil\sqrt{m}\rceil + 1, \ldots, \lceil\sqrt{m}\rceil\}$.
Step 2: Compute the solution for I by the deterministic online strategy $A(D_i)$.

Theorem 3.5.25. DIAG *is a randomized online algorithm for the* Unit (2) Job *problem with*

$$\text{Exp-Comp}_{\mathrm{DIAG}}(I) \leq 1 + \frac{1}{\sqrt{m}} + \frac{1}{2 \cdot m}$$

for every instance I of Unit $(m, 2)$ Job.

Proof. In the proof of Lemma 3.5.22 we have shown that, for any instance I of Unit $(m, 2)$ Job, the expected[30] delay over all $2 \cdot \sqrt{m} + 1$ diagonal strategies considered is at most

$$\lceil\sqrt{m}\rceil + \frac{1}{2}.$$

A consequence is that the expected cost of a solution over all $2 \cdot \sqrt{m} + 1$ diagonal strategies is at most

$$m + \lceil\sqrt{m}\rceil + \frac{1}{2}.$$

Since $\mathrm{Opt}_{\mathrm{Job}}(I) \geq m$ for all instances of Unit $(m, 2)$ Job, we obtain

$$\begin{aligned}\text{Exp-Comp}_{\mathrm{DIAG}}(I) &= \frac{\text{expected cost}}{\mathrm{Opt}_{\mathrm{Job}}(I)} \\ &\leq \frac{m + \sqrt{m} + \frac{1}{2}}{m} \\ &= 1 + \frac{1}{\sqrt{m}} + \frac{1}{2 \cdot m}.\end{aligned}$$ □

[30] A very formal proof of this fact would start by defining a random variable X that counts the number of obstacles in the randomly chosen diagonals, and continue by applying $\sum_{i=-\lceil\sqrt{m}\rceil}^{\lceil\sqrt{m}\rceil} X(D_i) \leq m$ from Observation 3.5.20

Exercise 3.5.26. Consider the Unit (3) Job problem. Estimate the hardness of this problem for deterministic online algorithms and generalize the randomized algorithm DIAG for instances of the Unit (3) Job problem.

3.6 Summary

In this chapter we have shown that a random choice of an algorithm from a suitable class of deterministic algorithms can lead to successful processing of problems for which no deterministic algorithm alone is able to solve the problem efficiently or with a required solution quality. The method of avoiding the worst-case inputs is usually called the method of foiling the adversary, because the algorithm design can be viewed as a game between an algorithm designer and her/his adversary, who tries to construct hard input instances for every algorithm designed. The art of successfully applying this method lies in searching for a suitable set of deterministic strategies. The term "suitable" means that, for every problem instance I, most of the algorithms of this class behave well[31] on I despite the fact that none of them is able to behave reasonably on all feasible inputs. If one knows that this is possible, then one tries to find a class of such algorithms, with a cardinality that is as small as possible in order to guarantee an efficient execution of the random choice from this class.

In the case of hashing, we call such classes of hash functions universal. We have seen that one can construct universal sets of hash functions, whose cardinality is acceptable for applications.

Online algorithms are algorithms that have to process a given request without having any information of future requests. Without knowing the future, it is very hard to compete against algorithms that have complete information about future requests, i.e., about the whole input. In the case of online problems, the adversary is in a very good position because it can construct hard instances by waiting for the decision of the online algorithm and then determining the next part of the input. By considering the Unit-Job problem we have shown that there are hard[32] problem instances for every deterministic online algorithm, but there is a randomized online algorithm with very good behavior for every feasible input.

A detailed study of hashing strategies is contained in most textbooks on algorithms and data structures. Here, we recommend the books of Cormen, Leiserson, Rivest and Stein [CLRS01], Ottmann and Widmayer [OW02], Schöning [Sch01], Gonnet, [Gon84], and Knuth [Knu73]. The concept of universal hashing has been proposed by Carter and Wegman [CW79]. Further improvements

[31] compute efficiently the correct result, or whatever one can expect from an algorithm

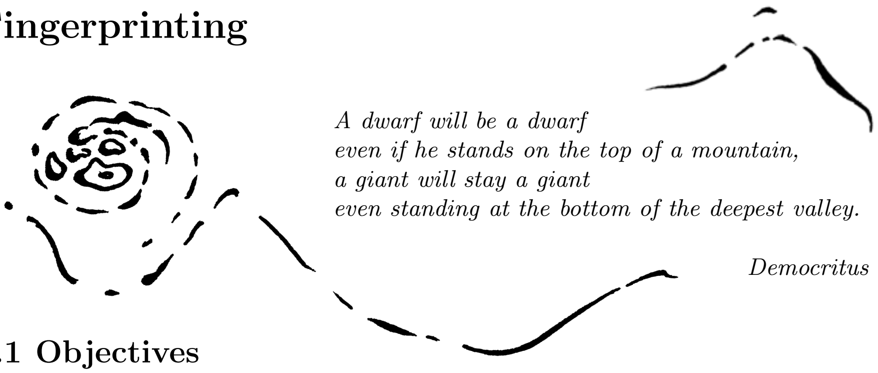
[32] in the sense that the algorithm is not able to compute a solution whose cost is very close to the optimal cost

of this concept are presented by Fredman, Komlós, and Szemerédi [FKS84] and Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, and Tarjan [DKM$^+$94].

The most comprehensive sources on online algorithms are the books of Fiat and Woeginger [FW98] and Borodin and El-Yaniv [BEY98]. The randomized online algorithm presented in Section 3.5 is a simplified version of an algorithm for job scheduling by Hromkovič, Steinhöfel, and Widmayer [HSW01].

This page intentionally left blank

4 Fingerprinting



A dwarf will be a dwarf
even if he stands on the top of a mountain,
a giant will stay a giant
even standing at the bottom of the deepest valley.

Democritus

4.1 Objectives

This chapter is devoted to applications of the fingerprinting method. We introduced this method in Section 2.6 as an approach for solving equivalence problems. The basic idea is to compare two (full) representations of some objects by the fingerprints of these representations. An important point is that one has several different ways for making fingerprints. For instance, in the randomized protocol R in Section 1.2, computing modulo p for each prime $p < n^2$ is a way of getting fingerprints (i.e., a kind of fingerprinting). The method of creating fingerprints is chosen at random from a finite set of possibilities. The main idea behind this is that, for any two different objects, most ways of creating fingerprints save the differences between the two objects. Thus, in general, we pose the following two requirements for fingerprints.

(i) The fingerprints have to be simple and short in order to assure the possibility of being able to compare them efficiently. To achieve this, the fingerprints may be not only compressed representations of given objects, but even incomplete representations of them.
(ii) Despite its bounded size, the fingerprint of an object has to contain as much essential information about the object as possible.

Hence, the basis of any application of the fingerprinting method is a set M of functions, each of them considered as a way of fingerprinting. More precisely, each element of M is a mapping from the given, full representation of the considered objects to their fingerprints. Our aim is to find a set M, such that for any two different objects O_1 and O_2 there are sufficiently many mappings f with $f(O_1) \neq f(O_2)$. Because of this, one considers fingerprinting as a special case of the method of abundance of witnesses. The set M can be viewed as the set of witness candidates for proving $O_1 \not\equiv O_2$, and a mapping $h \in M$ is a witness of $(O_1) \not\equiv (O_2)$ if $h(O_1) \neq h(O_2)$.

The art of applying the method of fingerprinting lies in a suitable choice of M. On the one hand, one aims to have fingerprints as representative as possible, and so assure a large number of witnesses among the witness candidates. On the other hand, we wish to have fingerprints as short as possible in order to guarantee their efficient comparison. Obviously, these two aims contradict each other. Small fingerprints also mean a small set of fingerprints, and so the situation where several different objects are mapped to the same fingerprint may occur more often (Figure 4.1).

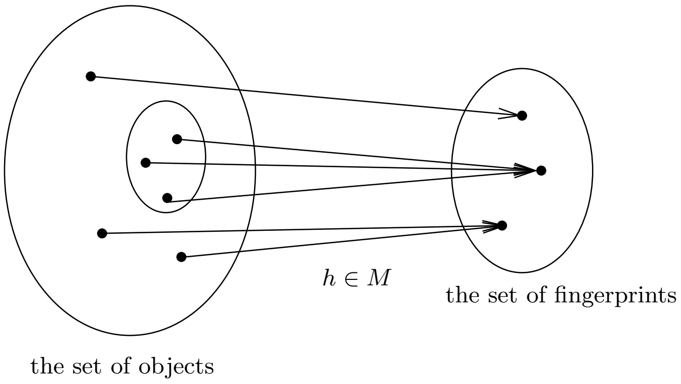


Fig. 4.1.

Hence, the crucial point in applying fingerprints lies in searching for a reasonable compromise between the efficiency of the fingerprints comparison (the size of fingerprints) and the abundance of witnesses in M. The objective of this chapter is to present searching for a suitable set M for some fundamental equivalence problems in a transparent way, and to provide the first experience in applying the fingerprinting method.

This chapter is organized as follows. In Section 4.2 we first generalize the concept of comparing two long strings (from Section 1.2) to decide whether a given long string is in a set of strings, and then to decide whether or not two string sets are disjoint. The second important goal of Section 4.2 is to show how to amplify the success probability by an improved choice of M, i.e., to show how to perform amplification in a way different from the execution of independent runs of a randomized algorithm on the same input (Section 2.6). The idea is to significantly decrease the error probability by an acceptable increase in the size of fingerprints. In Section 4.3 we apply this kind of fingerprinting once again in order to design an efficient randomized algorithm for the so-called substring problem.

In Section 4.4 we show how to efficiently verify the equality $A \cdot B = C$ for three $(n \times n)$ matrices, A, B, and C, in time $O(n^2)$ by the fingerprinting method. This application of fingerprinting is extended to a randomized polynomial-time algorithm for testing the equivalence of two polynomials.

Note, that there is no deterministic polynomial-time algorithm known for this equivalence problem.

4.2 Communication Protocols

In our initial example presenting the power of randomization in Section 1.2, we considered two computers R_{I} and R_{II} that were asked to verify whether the contents of their memories are identical. These contents were considered binary strings of length n, for an $n \in \mathbb{N}$. Next, we will consider some generalizations of this equivalence problem.

First, assume that R_{I} has a string $x \in \{0,1\}^n$ and that R_{II} has a set $U = \{u_1, u_2, \ldots, u_k\}$ of strings $u_i \in \{0,1\}^n$ for $i = 1, \ldots, k$. The computer R_{II} does not know any bit of the string x, and R_{I} does not have any knowledge of U. The computers have to find out whether or not $x \in U$. One can prove that deterministically this is not possible to do more efficiently than by sending the whole x from R_{I} to R_{II}, which finally checks whether $x \in U$ or $x \notin U$. Designing a randomized protocol for this communication task, we show that the same idea that is used for the comparison of two strings can again be applied. The question is, for which cardinalities of U does this approach still provide an efficient communication protocol.

We propose the following randomized protocol PSet that uses the set

$$M = \{h_p \mid p \in \mathrm{PRIM}\left(n^2\right)\}$$

of ways of fingerprinting.

PSet

Initial situation: R_{I} has a string $x \in \{0,1\}^n$. R_{II} has k different strings $u_1, u_2, \ldots, u_k \in \{0,1\}^n$. Let $U = \{u_1, u_2, \ldots, u_k\}$. The two computers R_{I} and R_{II} have to decide whether $x \in U$ or $x \notin U$.

Phase 1: R_{I} uniformly chooses a prime $p \in \mathrm{PRIM}\left(n^2\right)$ at random.

Phase 2: R_{I} computes the number

$$s = \mathrm{Number}(x) \mod p$$

and sends s and p to R_{II}.

Phase 3: After receiving s and p, the computer R_{II} computes the numbers

$$q_i = \mathrm{Number}(u_i) \mod p$$

for $i = 1, \ldots, k$.

If $s \in \{q_1, q_2, \ldots, q_k\}$, then R_{II} outputs "$s \in U$".

If $s \notin \{q_1, q_2, \ldots, q_k\}$, then R_{II} "$s \notin U$".

Clearly, the communication complexity of the protocol PSet is $4 \cdot \lceil \log_2 n \rceil$, i.e., the same as that of protocol P from Section 1.2.

In what follows, we analyze the error probability of PSet. As usual, we distinguish two cases.

(i) Let $x \in U$.
Then there exists a $j \in \{1, \ldots, k\}$ such that $x = u_j$. Similarly, as for the protocol P in Section 1.2, we have

$$\text{Number}(x) = \text{Number}(u_j) \pmod{m}$$

for all $m \in \mathbb{N} - \{0\}$, and so PSet outputs "$x \in U$" with certainty, i.e., the error probability is 0 in this case.

(ii) Let $x \notin U$.
Let $S_{\text{PSet},x} = \{P_r \mid r \in \text{PRIM}\left(n^2\right)\}$ be the set of all $\textit{Prim}\left(n^2\right)$ runs of PSet on x. Let A_i be the event[1] that

$$\text{Number}(x) \bmod r = \text{Number}(u_i) \bmod r$$

for each $i \in \{1, \ldots, k\}$. Obviously,

$$A = \bigcup_{i=1}^{k} A_i$$

is the event that PSet outputs the wrong answer "$x \in U$". Hence, Prob(A) is the error probability of PSet on x.

Analyzing our exemplary protocol in Section 1.2, we have shown that

$$\text{Prob}(A_i) \le \frac{n-1}{\textit{Prim}\,(n^2)} \le \frac{2 \cdot \ln n}{n} \tag{4.1}$$

for every integer $i \in \{1, \ldots, k\}$. In this way, we obtain

$$\begin{aligned} \text{Error}_{\text{PSet}}(x, U) &= \text{Prob}(A) \\ &= \text{Prob}\left(\bigcup_{i=1}^{k} A_i\right) \\ &\le \sum_{i=1}^{k} \text{Prob}(A_i) \\ &\underset{(4.1)}{\le} \sum_{i=1}^{k} \frac{2 \cdot \ln n}{n} \\ &= k \cdot \frac{2 \cdot \ln n}{n}. \end{aligned} \tag{4.2}$$

[1]Thus, $A_i = \{P_m \mid m \in \text{PRIM}\left(n^2\right) \text{ and } s = q_i \text{ in the run } P_m\}$.

A direct consequence is that

$$\text{Error}_{\text{PSet}}(x, U) \leq \frac{1}{2}$$

for $k \leq \frac{n}{4 \cdot \ln n}$.

Hence, PSet is a one-sided-error Monte Carlo protocol for deciding the membership of x to U for $k \leq n/(4 \cdot \ln n)$.

Exercise 4.2.1. Apply the amplification method in order to enable a randomized test of $x \in U$ for larger sets U. Estimate the maximal possible cardinality of U for which the achieved error probability still approaches 0 with growing n when the communication complexity is

(i) in $O(\log n \cdot \log \log n)$,
(ii) in $O\big((\log n)^d\big)$ for any constant $d \in \mathbb{N}$,
(iii) polylogarithmic.

Now, we consider another generalization of this communication task, where one has to decide whether or not two sets of strings are disjoint. This task is called the **disjointness problem** in what follows. The following randomized protocol again uses $M = \{h_p \mid p \in \text{PRIM}\,(n^2)\}$ as the set of fingerprinting ways.

Protocol PDisj

Initial Situation: Computer R_{I} has a set $V = \{v_1, v_2, \ldots, v_l\} \subseteq \{0,1\}^n$ and computer R_{II} has a set $U = \{u_1, u_2, \ldots, u_k\} \subseteq \{0,1\}^n$ for some positive integers l and k. Neither computer knows the data of the other. The computers R_{I} and R_{II} have to decide whether $V \cap U = \emptyset$ or $V \cap U \neq \emptyset$.

Phase 1: R_{I} chooses a prime $p \in \text{PRIM}\,(n^2)$ at random.

Phase 2: R_{I} computes the numbers

$$s_i = \text{Number}(v_i) \mod p$$

for $1, 2, \ldots, l$, and sends

$$p, s_1, s_2, \ldots, s_l$$

to R_{II}.

Phase 3: After receiving $p, s_1, s_2, \ldots, s_l$, the computer R_{II} computes the numbers

$$q_m = \text{Number}(u_m) \mod p$$

for all $m = 1, 2, \ldots, k$.
If $\{s_1, s_2, \ldots, s_l\} \cap \{q_1, q_2, \ldots, q_k\} \neq \emptyset$, R_{II} outputs "$U \cap V \neq \emptyset$".
If $\{s_1, s_2, \ldots, s_l\} \cap \{q_1, q_2, \ldots, q_k\} = \emptyset$, R_{II} outputs "$U \cap V = \emptyset$".

Clearly, the communication complexity of PDisj is

$$(l+1) \cdot 2 \cdot \lceil \log_2 n \rceil.$$

If $k < l$, then one can exchange the roles of R_{I} and R_{II}, and so remain within the communication complexity of $(k+1) \cdot 2 \cdot \lceil \log_2 n \rceil$ bits.

In what follows, we investigate the error probability of the protocol PDisj.

Lemma 4.2.2. PDisj *is a* 1MC* *protocol for the disjointness problem of two sets* $U, V \subseteq \{0,1\}^n$ *for*

$$|U| \cdot |V| = o(n/\ln n)\,.$$

Proof. In the analysis of the error probability of PDisj we handle separately the two possibilities $U \cap V \neq \emptyset$ and $U \cap V = \emptyset$.

(i) Let $U \cap V \neq \emptyset$.
Then there exist $i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, k\}$ such that $v_i = u_j$. Hence, $s_i = q_j$ for all primes p, and so PDisj outputs "$U \cap V \neq \emptyset$" with certainty.

(ii) Let $U \cap V = \emptyset$.
Let, for $i \in \{1, \ldots, l\}$, B_i be the event that $s_i \in \{q_1, q_2, \ldots, q_k\}$ through the fact $v_i \notin \{u_1, \ldots, u_k\}$. In (4.2) we have calculated

$$\mathrm{Prob}(B_i) \leq k \cdot \frac{2 \cdot \ln n}{n} \tag{4.3}$$

for all $i \in \{1, \ldots, l\}$. Clearly,

$$B = \bigcup_{i=1}^{l} B_i$$

is the event that PDisj provides the wrong answer "$U \cap V \neq \emptyset$". Hence,

$$\begin{aligned}
\mathrm{Error}_{\mathrm{PDisj}}(V,U) &= \mathrm{Prob}(B) \\
&= \mathrm{Prob}\left(\bigcup_{i=1}^{l} B_i\right) \\
&\leq \sum_{i=1}^{l} \mathrm{Prob}(B_i) \\
&\underset{(4.3)}{\leq} \sum_{i=1}^{l} k \cdot \frac{2 \cdot \ln n}{n} \\
&= l \cdot k \cdot \frac{2 \cdot \ln n}{n}.
\end{aligned}$$

Thus, the error probability tends to 0 with growing n, if

$$l \cdot k = o(n/\ln n)\,.$$

Hence, PDisj is a 1MC* protocol for the disjointness problem for $l \cdot k = o(n/\ln n)$. □

Exercise 4.2.3. Let $U, V \subseteq \{0,1\}^n$ and let $|U| \in O(n^3)$, and $|V| \in O(n^2)$. How many independent runs of PDisj are necessary (i.e., how large is the necessary communication complexity) to successfully decide the disjointness problem for U and V with an error probability tending to 0 with growing n? What happens when $U, V \in \Theta(2^{\sqrt{n}})$?

Exercise 4.2.4. The disjointness problem for an instance (U, V) can be solved by a deterministic protocol by sending the complete data of one computer to the other. In this way, the communication complexity is $n \cdot |U|$ or $n \cdot |V|$, but it is never necessary to exchange more than 2^n bits. Explain why 2^n bits always suffice. Do there exist cardinalities of U and V such that the randomized protocol PDisj is not better than an optimal deterministic protocol?

Exercise 4.2.5. Transform the protocol PDisj to a Las Vegas protocol. How large is its expected communication complexity?

So far, we have always amplified the success probability[2] by executing independent runs on the same input. Analyzing the protocol R for the equality problem in Section 1.2 we have recognized that for every input (x, y) with $x \neq y$, there are at most $n - 1$ bad[3] primes. Following the proof of this fact we see that this upper bound on the number of bad primes does not depend on considering PRIM (n^2) as a basis for choosing primes. For instance, this means that when exchanging PRIM (n^2) for PRIM (n^3) it remains true that there are at most $n - 1$ bad primes in PRIM (n^3), but the number of good primes has increased substantially. And this is very good news, because the consequence is an essential increase in the success probability. Thus, let us consider the following protocols d-R for all integers $d \geq 2$.

Protocol *d*-R

Initial Situation: Computer R_{I} has an $x \in \{0,1\}^n$ and computer R_{II} has a $y \in \{0,1\}^n$.

Phase 1: R_{I} uniformly chooses a prime p from the set

$$\text{PRIM}\left(n^d\right) = \{p \mid p \leq n^d \text{ is a prime}\}$$

at random.

Phase 2: R_{I} computes the number

$$s = \text{Number}(x) \mod p$$

and sends s and p to R_{II}.

[2]It does not matter whether one views the success as computing the correct output or as an efficient computation of the correct output.

[3]Remember that a bad prime is a prime that is no witness of "$x \neq y$".

Phase 3: After receiving s and p, R_{II} computes the number

$$q = \mathrm{Number}(y) \mod p.$$

If $q \neq s$, then R_{II} outputs "$x \neq y$".
If $q = s$, then R_{II} outputs "$x = y$".

Since s and p are smaller than n^d, the communication complexity of d-R is bounded by

$$2 \cdot \lceil \log_2 n^d \rceil \leq 2 \cdot d \cdot \lceil \log_2 n \rceil.$$

Obviously, for all inputs (x, y) with $x = y$ the error probability is (as before) equal to 0.

For inputs (x, y) with $x \neq y$, the error probability is equal to

$$\frac{\text{the number of bad primes for } (x, y) \text{ in PRIM}\left(n^d\right)}{|\mathrm{PRIM}\,(n^d)\,|},$$

i.e., at most

$$\frac{n-1}{\mathit{Prim}\,(n^d)} \leq \frac{n}{n^d/\ln n^d} = \frac{d \cdot \ln n}{n^{d-1}}$$

for all sufficiently large n.

Observe that the result of our analysis is very appreciative because the communication complexity grows only linearly in d, but the error probability tends to 0 with exponential speed with respect to d.

Exercise 4.2.6. Apply this success amplification method to the protocols PSet and PDisj, and estimate the error probability as a function of n, d, and the cardinalities of U and V.

Exercise 4.2.7. Compare the amplification method of executing independent runs with the method of increasing the size of the set of witness candidates (the set of fingerprinting methods). How many communication bits are needed by each of these methods in order to get an error probability tending to 0 with growing n? By what size is the error probability reducible by these methods when an upper bound $c(n)$ on the number of communication bits is given?

Exercise 4.2.8. Consider a network of l computers $R_1, R_2, \ldots, R_l$ in which each computer is directly connected via a communication link to each other. Let k be a positive integer. Assume that each R_i possesses a set $S_i \subseteq \{0,1\}^n$, where $|S_i| \leq k$ for all $i \in \{1, 2, \ldots, l\}$. Design and analyze a randomized communication protocol for deciding whether or not $\bigcap_{i=1}^{l} S_i$ is empty. The communication complexity is measured as the number of bits communicated via all links between the l computers.

Exercise 4.2.9. Let us consider the following communication task. R_{I} has got a sequence $x_1, x_2, \ldots, x_n$ of strings, $x_i \in \{0,1\}^n$ for $i = 1, 2, \ldots, n$. R_{II}

has got a sequence $y_1, y_2, \ldots, y_n$ of strings, $y_i \in \{0,1\}^n$ for $i = 1, 2, \ldots, n$. The question is whether there exists a $j \in \{1, 2, \ldots, n\}$ such that $x_j = y_j$. One can prove that any deterministic protocol solving this task has a communication complexity of at least n^2. Design a Las Vegas protocol that solves this problem with expected communication complexity in $O(n \cdot \log n)$.

4.3 The Substring Problem

An important class of algorithmic problems are the tasks of pattern recognition that frequently occur in text processing and graphics. Typical tasks are the recognition of patterns in a complex scenario or the classification of objects with respect to a set of given patterns. Here we consider the simplest basic task of this kind – the substring problem. Given a pattern $x = x_1x_2 \ldots x_n$ and a text $y = y_1y_2 \ldots y_m$ as strings over an alphabet Σ ($x_i \in \Sigma$ for $i = 1, \ldots, n$ and $y_j \in \Sigma$ for $j = 1, \ldots, m$), the task is to decide whether or not x is a substring[4] of y. Moreover, if x is a substring of y, then one has to compute the smallest index r such that

$$x_1x_2 \ldots x_n = y_r \ldots y_{r+n-1}.$$

This task is of interest not only in text processing, but occurs frequently as one of the basic tasks of molecular biology.

In what follows, we simplify the matter by choosing $\Sigma = \{0,1\}$. Let, for every $k \in \{1, \ldots, m-1\}$ and every $r \in \{1, \ldots, m-k+1\}$,

$$y(r,k) = y_ry_{r+1} \ldots y_{r+k-1}$$

be the substring of y of length k that begins at the r-th bit position of y. For given $x = x_1x_2 \ldots x_n$ and $y = y_1y_2 \ldots y_m$, a naive deterministic algorithm compares x with all substrings $y(r,n)$ for $r = 1, 2, \ldots, m-n+1$ in order to find x in y. This naive algorithm executes $O(n+m)$ operations over the symbols of $\Sigma = \{0,1\}$. Note that there exist faster deterministic algorithms that run in the optimal time $O(n+m) = O(m)$. In what follows we present a simple Las Vegas algorithm for this task. Let $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a mapping.

Algorithm STRING (f)

Input: Two strings $x = x_1x_2 \ldots x_n$ and $y = y_1y_2 \ldots y_m$ over $\{0,1\}$, $n \leq m$.
Step 1: Choose uniformly a prime p from PRIM $(f(n,m))$ at random.
Step 2: Compute

$$\text{Finger}_p(x) := \text{Number}(x) \mod p.$$

[4]Remember that a string x is called a substring (or a subword) of a string y, if $y = uxv$ for some strings u and v.

Step 3: Compute, sequentially,

$$\text{Finger}_p(y(r,n)) := \text{Number}\,(y,(r,n)) \mod p$$

for all $r \in \{1, 2, \ldots, m-n+1\}$, and check whether

$$\text{Finger}_p(y(r,n)) = \text{Finger}_p(x)\,.$$

For every $j \in \{1, 2, \ldots, m-n+1\}$ such that $\text{Finger}_p(y(j,n)) = \text{Finger}_p(x)$, compare $y(j,n)$ and x.
If $y(j,n) = x$, then halt and output "j".
Else continue computing $\text{Finger}_p(y(j+1,n))$.
If no r with $y(r,n) = x$ has been found, output "$\emptyset$".

Clearly, STRING (f) is a Las Vegas algorithm, because STRING (f) always computes the correct output.

In what follows, we analyze the expected time complexity of STRING (f). Here, we count one time unit for the comparison of two fingerprints and n time units for the bit-by-bit comparison of x and $y(r,n)$. Hence, the time complexity is measured in the number of operations in $\mathbb{Z}_p$, or in bit comparisons.

Clearly, one can compute $\text{Finger}_p(x)$ and $\text{Finger}_p(y(1,n))$ in time $O(|x|) = O(n)$. We show now that one can compute all fingerprints

$$\text{Finger}_p(y(r,n)) \text{ for } r = 1, \ldots, m-n+1$$

in an overall time of $O(m)$. This is possible because

$$\text{Number}\,(y(r+1,n)) = 2 \cdot \left[\text{Number}\,(y(r,n)) - 2^{n-1} \cdot y_r\right] + y_{r+n},$$

and so one can compute $\text{Finger}_p(y(r+1,n))$ from $\text{Finger}_p(y(r,n))$ as follows:

$$\text{Finger}_p(y(r+1,n)) = \left(2 \cdot \left[\text{Finger}_p(y(r,n)) - (2^{n-1} \cdot y_r) \mod p\right] + y_{r+n}\right) \mod p.$$

Hence, $O(1)$ operations over the field $\mathbb{Z}_p$ are sufficient for computing the fingerprint $\text{Finger}_p(y(r+1,n))$ from the fingerprint $\text{Finger}_p(y(r,n))$.

Let (x,y) be an input where x is not a substring of y. Next we give an upper bound on the expected time complexity of STRING (f) on (x,y). Let $(\text{PRIM}\,(f(n,m))\,, \text{Prob})$ be the corresponding probability space, where Prob is the uniform probability distribution over all primes not greater that $f(n,m)$. Let, for $r = \{1, \ldots, m-n+1\}$,

$$A_r \text{ be the event that } \text{Finger}_p(x) = \text{Finger}_p(y(r,n))\,.$$

In the same way as for the protocol in Section 1.2, one can show that

$$\text{Prob}(A_r) \leq \frac{n-1}{\mathit{Prim}\,(f(n,m))} \leq \frac{n \cdot \ln(f(n,m))}{f(n,m)}.$$

Then the expected time complexity of STRING (f) is

$$\begin{aligned}\text{Exp-Time}_{\text{STRING}(f)}((x,y)) &= O(m) + \sum_{r=1}^{m-n+1} (1 + \text{Prob}(A_r) \cdot n) \\ &\quad \{\text{The complexity } O(m) \text{ is sufficient to compute Finger}_p(x) \text{ in step 2 and, as shown above, the complexity } O(m) \text{ is also sufficient to compute all fingerprints Finger}_p(y(r,n)).\} \\ &\leq O(m) + \sum_{r=1}^{m-n+1} \frac{n \cdot \ln(f(n,m))}{f(n,m)} \cdot n \\ &\leq O(m) + m \cdot n^2 \cdot \frac{\ln(f(n,m))}{f(n,m)}.\end{aligned}$$

Choosing $f(n,m) = n^2 m \cdot \ln(n^2 m)$, one obtains

$$\text{Exp-Time}_{\text{STRING}(n^2 m \cdot \ln(n^2 m))}(x,y) \in O(m)\,.$$

We close our analysis by observing that the expected time complexity of STRING (f) on inputs (x, uxv) cannot be larger than the expected complexity on inputs in which the pattern does not occur in the text.

Exercise 4.3.10. Let (x,y) be an input with $y(j,n) = x$, and let j be the smallest number from $\{1, 2, \ldots, m-n+1\}$ with this property. Give a detailed analysis of the expected time complexity of STRING $\big(n^2 m \cdot \ln(n^2 m)\big)$ on (x,y).

4.4 Verification of Matrix Multiplication

The multiplication of two matrices over a field $\mathbb{F}$ is one of the most basic mathematical tasks.[5] The execution of the school algorithm for matrix multiplication needs $O\big(n^3\big)$ arithmetical operations over $\mathbb{F}$. Based on the "divide and conquer" design method, Strassen developed an $O\big(n^{\log_2 7}\big)$-algorithm for matrix multiplication. A further development of this design idea provided a sequence of several improvements, and the currently best known algorithm for matrix multiplication runs in time $O\big(n^{2.376}\big)$.

The task considered here is a little bit simpler than matrix multiplication. Given three $(n \times n)$-matrices A, B, and C over a field $\mathbb{F}$, one has to decide whether

$$A \cdot B = C$$

[5]It is well known that the complexity of matrix multiplication is asymptotically the same as the complexity of computing an inverse matrix, and that it is strongly related the complexity of solving a system of linear equations.

i.e., whether $A \cdot B$ is the same matrix as C. A naive deterministic approach to solve this equivalence problem is based on computing $A \cdot B$, and then comparing the result with C. The complexity of this approach is asymptotically the same as the complexity of multiplying the matrices A and B. Our aim is to apply the fingerprinting method to design a randomized $O(n^2)$-algorithm for this equivalence problem. The idea is to take fingerprints as vectors and to consider the set

$$M = \{f_\alpha : \mathbb{F}^{n\times n} \to \mathbb{F}^n \mid f_\alpha(A) = A \cdot \alpha \text{ and } \alpha \in \{0,1\}^n\},$$

where $\mathbb{F}^{n\times n}$ is the set of all $(n\times n)$-matrices over $\mathbb{F}$, and $\mathbb{F}^n$ is the set of all n-dimensional vectors[6] $(\alpha_1, \alpha_2, \ldots, \alpha_n)^\top$ over $\mathbb{F}$. Taking the set M as the base of the fingerprinting method, one obtains the following algorithm discovered by Freivalds for the verification of the matrix multiplication.

Algorithm FREIVALDS

```
Input: Three (n × n)-matrices, A, B, and C over a field F.
Step 1: Choose uniformly a vector $\alpha \in \{0,1\}^n$ at random.
Step 2: Compute the vectors
    $\beta := A \cdot (B \cdot \alpha)$
    $\gamma := C \cdot \alpha$
Step 3:
    if $\beta = \gamma$ then output "$A \cdot B = C$",
    if $\beta \neq \gamma$ then output "$A \cdot B \neq C$".
```

To compute β, one has to perform two multiplications of an $(n\times n)$-matrix with an n-dimensional vector. To compute γ, it is sufficient to perform one multiplication of an $(n\times n)$ matrix C with a vector α. The comparison between β and γ can be performed in linear time, and so the overall complexity of the algorithm FREIVALDS is in $O(n^2)$.

In what follows, we show that FREIVALDS is a 1MC algorithm for the equivalence problem considered.

If $A \cdot B = C$, then

$$A \cdot B \cdot \alpha = C \cdot \alpha$$

for all $\alpha \in \{0,1\}^n$. Hence, FREIVALDS provides the correct answer with certainty.

To analyze the case when $A \cdot B \neq C$ we use the following lemma.

Lemma 4.4.11. *Let A, B, and C be $(n \times n)$-matrices over a field $\mathbb{F}$, such that $A \cdot B \neq C$. Then at least 2^{n-1} vectors α from $\{0,1\}^n$ are witnesses of "$A \cdot B \neq C$" in the sense that*

$$A \cdot B \cdot \alpha \neq C \cdot \alpha.$$

[6] $(n \times 1)$-matrices over $\mathbb{F}$

Proof. If $A \cdot B \neq C$, the matrix

$$D = A \cdot B - C$$

contains at least one element different from 0. This means that it is sufficient to show that, for at least half the vectors[7] α from $\{0,1\}^n$, the vectors

$$D \cdot \alpha = \delta = (\delta_1, \delta_2, \ldots, \delta_n)$$

are different from the 0-vector, assuming D is not a 0-matrix.

Let $D = [d_{ij}]_{i,j=1,2,\ldots,n}$. Let $i \in \{1,2,\ldots,n\}$ be an index such that the i-th row d_i of D is not a 0-vector. Let $d_{is_1}, d_{is_2}, \ldots, d_{is_l}, l \geq 1$, be all elements from d_i that are different from 0. Obviously, it is sufficient to show that $\delta_i \neq 0$ for at least 2^{n-1} vectors α.

Observe that

$$\delta_i = \sum_{j=1}^{n} d_{ij} \cdot \alpha_j = \sum_{k=1}^{l} d_{is_k} \cdot \alpha_{s_k}.$$

Clearly,

$$\delta_i = 0 \quad \Leftrightarrow \quad \sum_{k=1}^{l} d_{is_k} \cdot \alpha_{s_k} = 0 \quad \Leftrightarrow \quad \alpha_{s_1} = -\frac{1}{d_{is_1}} \cdot \sum_{k=2}^{l} d_{is_k} \cdot \alpha_{s_k}.$$

This means that the value of α_{s_1} is unambiguously determined by the values of the remaining elements of α. Hence, there exist at most 2^{n-1} vectors α such that $\delta_i = 0$ in $\delta = D \cdot \alpha$. □

Lemma 4.4.11 assures that at least half the vectors from $\{0,1\}^n$ are witnesses of the difference between $A \cdot B$ and C. Therefore, the error probability of FREIVALDS on any input (A, B, C) with $A \cdot B \neq C$ is at most 1/2. Hence, FREIVALDS is a 1MC algorithm for the verification of the matrix multiplication.

Exercise 4.4.12. Let p be a prime. Let $\mathbb{Z}_p$ be the finite field of p elements with the operations $\oplus_{\bmod p}$ and $\odot_{\bmod p}$. Let A, B, and C be $(n \times n)$-matrices over $\mathbb{Z}_p$ such that $A \cdot B \neq C$. How many vectors from $(\mathbb{Z}_p)^n$ are witnesses of this inequality? What is the effect of exchanging $\{0,1\}^n$ for $(\mathbb{Z}_p)^n$ on the error probability of our algorithm, and how does one pay for the reduction of the error probability? Why does one not try to reduce the error probability in the case of an infinite field $\mathbb{F}$ by choosing a vector from $\mathbb{F}^n$ at random instead of a random choice from $\{0,1\}^n$?

[7] 2^{n-1} many

4.5 Equivalence of Two Polynomials

In this section we consider an equivalence problem for which no deterministic polynomial-time algorithm is known, and that can be solved efficiently by the method of fingerprinting. The problem is to decide the equivalence of two polynomials of several variables over a finite field $\mathbb{Z}_p$ for a prime p. Two polynomials $P_1(x_1, \ldots, x_n)$ and $P_2(x_1, \ldots, x_n)$ are said to be equivalent over $\mathbb{Z}_p$, if for all $(\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$

$$P_1(\alpha_1, \ldots, \alpha_n) \equiv P_2(\alpha_1, \ldots, \alpha_n) \pmod{p}.$$

One does not know any polynomial-time algorithm for this problem. A naive observer may deny this by pointing out that the comparison of two polynomials is easy because it is done simply by comparing the coefficients of the corresponding terms. We know that two polynomials P_1 and P_2 are identical iff the coefficients of P_1 are the same as those of P_2. But the real problem is that in order to perform this coefficient comparison, one has to transform the polynomials to their normal form first. The normal form of a polynomial of n variables $x_1, x_2, \ldots, x_n$ and a degree[8] d is

$$\sum_{i_1=0}^{d} \sum_{i_2=0}^{d} \cdots \sum_{i_n=0}^{d} c_{i_1, i_2, \ldots, i_n} \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n}.$$

However, the input polynomials for our equivalence test may be in an arbitrary form, for instance, as

$$P(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 + x_2)^{10} \cdot (x_3 - x_4)^7 \cdot (x_5 + x_6)^{20}.$$

Applying the binomial formula

$$(x_1 + x_2)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot x_1^k \cdot x_2^{n-k},$$

it is obvious that $P(x_1, x_2, x_3, x_4, x_5, x_6)$ can have $10 \cdot 7 \cdot 20 = 1400$ terms with nonzero coefficients. Thus the normal form of a polynomial can be exponentially longer than its representation, and hence, in general, one cannot compute the normal form from a given form in polynomial time.[9] If one wants to be efficient, one has to find a way of comparing two polynomials without creating their normal forms.

To do this we use a very simple strategy that is based on the concept of Freivalds, presented in Section 4.4. One can view the application of the fingerprinting method for comparing $A \cdot B$ with C in Section 4.4 in the following way:

[8]The degree of a polynomial of several variables is the maximum degree of all variables.

[9]Note the similarity to the situation when one wants to compare the matrices $A \cdot B$ and C without computing $A \cdot B$.

(i) $A \cdot B$ and C are two linear functions f_{AB} and f_C from $\mathbb{F}^n$ to $\mathbb{F}^n$, where $f_{AB}(\alpha) = (A \cdot B) \cdot \alpha$ and $f_C(\alpha) = C \cdot \alpha$, and
(ii) an argument $\alpha \in \mathbb{F}^n$ for f_{AB} and f_C is a witness of $AB \neq C$ if $f_{AB}(\alpha) \neq f_C(\alpha)$.

If one applies this idea for the comparison of two polynomials, one obtains the following definition of witnesses. For two polynomials $P_1(x_1, \ldots, x_n)$ and $P_2(x_1, \ldots, x_n)$, $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ is a witness of

$$P_1(x_1, \ldots, x_n) \not\equiv P_2(x_1, \ldots, x_n)$$

if

$$P_1(\alpha_1, \ldots, \alpha_n) \bmod p \neq P_2(\alpha_1, \ldots, \alpha_n) \bmod p.$$

In the language of fingerprinting,

$$h_\alpha(P_i) = P_i(\alpha_1, \ldots, \alpha_n) \bmod p$$

is the fingerprint of P_i for $i = 1, 2$. This way, one obtains the following algorithm.

Algorithm AQP

Input: A prime p and two polynomials P_1 and P_2 over $\mathbb{Z}_p$ in n variables $x_1, \ldots, x_n$, $n \in \mathbb{N} - \{0\}$, with a maximal degree of d, $d \in \mathbb{N}$.
Step 1: Choose uniformly an $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ at random.
Step 2: Compute the fingerprints
$h_\alpha(P_1) = P_1(\alpha_1, \ldots, \alpha_n) \bmod p$, and
$h_\alpha(P_2) = P_2(\alpha_1, \ldots, \alpha_n) \bmod p$.
Step 3:

```
if $h_\alpha(P_1) = h_\alpha(P_2)$ then
    output "$P_1 \equiv P_2$"
else
    output "$P_1 \not\equiv P_2$"
```

The complexity of the algorithm AQP is clearly in $O(m \cdot \log_2 d)$ with respect to the operations over $\mathbb{Z}_p$, where m is the length of the representation of the input (P_1, P_2).

Now we analyze the error probability of the algorithm AQP. If P_1 and P_2 are equivalent over $\mathbb{Z}_p$, then

$$P_1(\alpha_1, \ldots, \alpha_n) \equiv P_2(\alpha_1, \ldots, \alpha_n) \pmod{p}$$

for all $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$. Hence, the error probability for inputs (P_1, P_2) with $P_1 \equiv P_2$ is equal to 0.

Let P_1 and P_2 be two polynomials that are not equivalent over $\mathbb{Z}_p$. In what follows, we aim to show that if $p \geq 2nd$, then the error probability of AQP is less than $1/2$.

The question of whether

$$P_1(x_1, \ldots, x_n) \equiv P_2(x_1, \ldots, x_n)$$

is equivalent to the question of whether

$$Q(x_1, \ldots, x_n) = P_1(x_1, \ldots, x_n) - P_2(x_1, \ldots, x_n) \equiv 0.$$

This means that if P_1 and P_2 are not equivalent, the polynomial Q is not identical to 0 (zero polynomial). Now, we show that the number of roots of a polynomial $Q \not\equiv 0$ over n variables and of a degree d is bounded. This means that there are sufficiently many witnesses $\alpha \in (\mathbb{Z}_p)^n$ with

$$Q(\alpha) \not\equiv 0 \pmod{p}, \text{ i.e., } P_1(\alpha) \not\equiv P_2(\alpha) \pmod{p}.$$

We start with the well known theorem about the number of roots of polynomials over one variable.

Theorem 4.5.13. *Let d be a nonnegative integer. Every polynomial $P(x)$ of a singe variable x over any field and of degree d has either at most d roots or is equal to 0 everywhere.*[10]

Proof. We prove Theorem 4.5.13 by induction with respect to the degree d.

(i) Let $d = 0$. Then $P(x) = c$ for a constant c. If $c \neq 0$, then $P(x)$ does not have any root.

(ii) Assume that Theorem 4.5.13 holds for $d - 1$, $d \geq 1$. Now, we prove it for d.

Let $P(x) \not\equiv 0$ and let a be a root of P. Then

$$P(x) = (x - a) \cdot P'(x)$$

where $P'(x) = \frac{P(x)}{(x-a)}$ is a polynomial of degree $d - 1$. From the induction hypothesis $P'(x)$ has at most $d - 1$ roots. Therefore $P(x)$ has at most d roots.

□

Now we are ready to prove that there are sufficiently many witnesses[11] of the nonequivalence of different P_1 and P_2 over $\mathbb{Z}_p$ for a sufficiently large prime p.

[10] i.e., is a zero polynomial

[11] nonroots of $Q(x_1, \ldots, x_n) = P_1(x_1, \ldots, x_n) - P_2(x_1, \ldots, x_n)$

Theorem 4.5.14. *Let p be a prime, and let d and n be positive integers. Let $Q(x_1, \ldots, x_n) \not\equiv 0$ be a polynomial over $\mathbb{Z}_p$ in n variables $x_1, \ldots, x_n$, where each variable has degree of at most d in Q. Then, Q has at most*

$$n \cdot d \cdot p^{n-1}$$

roots.

Proof. We prove Theorem 4.5.14 by induction with respect to the number n of variables in Q.

(i) Let $n = 1$. Then Theorem 4.5.13 implies that $Q(x_1)$ has at most

$$d = n \cdot d \cdot p^{n-1} \text{ (for } n = 1)$$

roots.

(ii) Assume that the assertion of Theorem 4.5.14 is true for $n-1$, $n \in \mathbb{N} - \{0\}$. We prove it for n. We can express Q as

$$\begin{aligned} Q(x_1, x_2, \ldots, x_n) &= Q_0(x_2, \ldots x_n) + x_1 \cdot Q_1(x_2, \ldots, x_n) + \ldots \\ &\quad + x_1^d \cdot Q_d(x_2, \ldots, x_n) \\ &= \sum_{i=0}^{d} x_1^i \cdot Q_i(x_2, \ldots, x_n) \end{aligned}$$

for some polynomials

$$Q_0(x_2, \ldots x_n),\ Q_1(x_2, \ldots, x_n),\ \ldots,\ Q_d(x_2, \ldots, x_n)$$

in the $n-1$ variables $x_2, \ldots, x_n$.

If

$$Q(\alpha_1, \alpha_2, \ldots, \alpha_n) \equiv 0 \bmod p$$

for an $\alpha = (\alpha_1, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$, then either

(a) $Q_i(\alpha_2, \ldots, \alpha_n) \equiv 0 \bmod p$ for all $i = 0, 1, \ldots, d$, or

(b) there exists a $j \in \{0, 1, \ldots, d\}$ with $Q_i(\alpha_2, \ldots, \alpha_n) \not\equiv 0 \bmod p$, and α_1 is a root of the polynomial

$$\begin{aligned} \overline{Q}(x_1) &= Q_0(\alpha_2, \ldots \alpha_n) + x_1 \cdot Q_1(\alpha_2, \ldots, \alpha_n) + \ldots \\ &\quad + x_1^d \cdot Q_d(\alpha_2, \ldots, \alpha_n) \end{aligned}$$

in the variable x_1.

Now we count the number of roots in the cases (a) and (b) separately.

(a) Since $Q(x_1, \ldots, x_n) \not\equiv 0$, there exists a $k \in \{0, 1, \ldots, d\}$ such that

$$Q_k(x_2, \ldots, x_n) \not\equiv 0.$$

The induction hypothesis implies that the number of roots of Q_k is at most

$$(n-1) \cdot d \cdot p^{n-2}.$$

Hence, there are at most $(n-1) \cdot d \cdot p^{n-2}$ elements $\overline{\alpha} = (\alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^{n-1}$ such that

$$Q_i(\overline{\alpha}) \equiv 0 \bmod p$$

for all $i \in \{0, 1, 2, \ldots, d\}$. Since the value α_1 of x_1 does not have any influence on the condition (a), α_1 can be chosen arbitrarily from $\{0, 1, \ldots, p-1\}$. Thus there are at most

$$p \cdot (n-1) \cdot d \cdot p^{n-2} = (n-1) \cdot d \cdot p^{n-1}$$

elements $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$ that have the property (a).

(b) Since $\overline{Q}(x_1) \not\equiv 0$, the polynomial $\overline{Q}$ has at most d roots[12] (i.e., at most d values $\alpha_1 \in \mathbb{Z}_p$, with $\overline{Q}(\alpha_1) \equiv 0 \bmod p$). Therefore, there are at most

$$d \cdot p^{n-1}$$

values[13] $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in (\mathbb{Z}_p)^n$, that satisfy the property (b).

Combining (a) and (b), $Q(x_1, \ldots, x_n)$ has at most

$$(n-1) \cdot d \cdot p^{n-1} + d \cdot p^{n-1} = n \cdot d \cdot p^{n-1}$$

roots.

□

Corollary 4.5.15. *Let p be a prime, and let n and d be positive integers. For every polynomial $Q(x_1, \ldots, x_n) \not\equiv 0$ over $\mathbb{Z}_p$ of degree at most d, the number of witnesses of "$Q \not\equiv 0$" is at least*

$$\left(1 - \frac{n \cdot d}{p}\right) \cdot p^n.$$

Proof. The number of elements in $(\mathbb{Z}_p)^n$ is exactly p^n and Theorem 4.5.14 implies that at most $n \cdot d \cdot p^{n-1}$ of them are not witnesses. Hence, the number of witnesses is at least

$$p^n - n \cdot d \cdot p^{n-1} = \left(1 - \frac{n \cdot d}{p}\right) \cdot p^n.$$

□

Thus the probability of choosing uniformly a witness of "$Q \not\equiv 0$" at random from p^n elements of $(\mathbb{Z}_p)^n$ is at least

$$\left(1 - \frac{n \cdot d}{p}\right).$$

[12] Theorem 4.5.14

[13] Note that the values $\alpha_1, \alpha_2, \ldots, \alpha_n$ can be chosen arbitrarily.

For $p > 2nd$, the probability of choosing a witness is at least $1/2$. By executing several independent random choices from $(\mathbb{Z}_p)^n$, the probability of finding at least one witness of $Q \not\equiv 0$ (i.e., of $P_1(x_1, \ldots, x_n) \not\equiv P_2(x_1, \ldots, x_n)$) can be brought arbitrarily close to 1.

Exercise 4.5.16.* Extend the algorithm AQP to decide the equivalence of two polynomials over infinite fields, too.

Exercise 4.5.17. Consider the communication protocol R presented in Section 1.2 for the comparison of two binary strings (of the contents of two databases). Let p be a prime and let $a = a_1a_2 \ldots a_n$, $a_i \in \{0, 1\}$ for $i = 1, \ldots, n$, be a binary string. Consider the polynomial P_a of a singe variable x defined over $\mathbb{Z}_p$ as follows.

$$P_a(x) = \sum_{i=1}^{n} a_i x^{i-1}.$$

Apply the idea of algorithm AQP on order to design a 1MC protocol for the comparison of two n bit strings $a = a_1a_2 \ldots a_n$ and $b = b_1b_2 \ldots b_n$. What effect does the choice of p have on the error probability and on the communication complexity? Compare the results with the complexity and the success probability of the protocol R from Section 1.2.

4.6 Summary

The fingerprinting method can be viewed as a special case of the method of the abundance of witnesses. It is a robust technique for designing efficient randomized algorithms for the comparison of representations of two objects (i.e., for solving equivalence problems). It sometimes happens that one is not able to efficiently decide whether or not two object representations (descriptions) represent the same object. The reasons for the hardness of these equivalence tasks often lies in the size of the representations of the objects of the object class considered or in the fact that the representations used are unaffordable for efficient comparison. Deciding the equivalence of two polynomials given in an “arbitrary” representation is an exemplary problem for successful application of fingerprinting. Since the transformation of a polynomial representation into the normal form can exponentially increase the representation size, fingerprinting is the only known efficient way to solve this problem. The basic idea of fingerprinting is to compare fingerprints of the given representations of some objects in order to decide about the equivalence of these representations. A fingerprint of a representation of an object is a short and possibly incomplete representation of this object. To get a short representation (a fingerprint), and so make an efficient comparison possible, we have to pay with the incompleteness of the representation, and so with the possibility of making a wrong decision. This strongly reminds us of hashing, where the fingerprints of the keys determine the distribution of data from an actual set S of data records in a discrete direct access memory T. The art of applying the method of fingerprinting lies in the choice of a set M of fingerprinting methods, such that

for every pair of different objects of the set of objects considered, a random choice of a fingerprinting method from M provides a witness of the difference between these two objects with a reasonable probability.[14]

The power of this method was impressively demonstrated by the protocol R for the comparison of two strings in Section 1.2. In Section 4.2, we have extended this concept in order to design randomized protocols for decision problems such as the membership of a string in a set of strings and the disjointness problem for two sets of strings. We have seen that fingerprinting can be successfully applied to small input sets. Along the way we have learned a new[15] way of reducing the error probability to arbitrarily close to 0. The main idea is to allow larger fingerprints, and in this way to decrease the number of witnesses among the witness candidates (the fingerprinting methods) for any input.

In Section 4.3 we presented another application of randomized string comparison that resulted in the design of an efficient[16] randomized algorithm for the substring problem.

In Section 4.4 we saw that one can compare representations of matrices given in the form of a product of several matrices without executing matrix multiplications. For the matrix representation $A \cdot B$ (as the product of two matrices A and B), we determine the fingerprint $A \cdot (B \cdot \alpha)$ by choosing a vector α at random. The gain in complexity corresponds to the difference between matrix multiplication and the multiplication of a matrix by a vector. We can view this approach as regarding matrices (for instance, given as the product $A \cdot B$, or even as the product of several matrices) as mappings from a set of vectors to a set of vectors, and considering fingerprints determined by a vector α as the values of these mappings for the argument α. It may be surprising that a simple computation of the function value for a given argument can result in a successful application of the fingerprinting method. In Section 4.5 we saw that this simple and elegant approach of Freivalds works for the equivalence problem of two polynomials, too. The reason for success in this case is that the number of roots of polynomials of several variables is not too large when compared with the number of all arguments, and so one can apply fingerprinting to efficiently test whether or not a given polynomial

[14]In other words, a large part of the fingerprinting ways of M maps the two given object representations to different fingerprints, and so proves the difference between these two objects.

[15]different from executing several independent runs on the same input

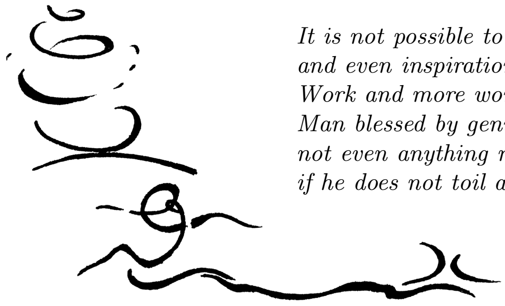
[16]even in the design of an optimal algorithm, because a running time cannot lay asymptotically below the input length

is the zero polynomial. Then, it is sufficient to observe that two polynomials are identical if and only if their difference is the zero polynomial.

The most transparent examples of the application of fingerprinting can be found in the theory of communication complexity, overviews of which are given in the monographs of Kushilevitz and Nisan [KN97] and Hromkovič [Hro97]. Mehlhorn and Schmidt [MS82] applied this method to fix the maximal possible difference between deterministic communication and Las Vegas communication. The concept of comparing two mappings by simply evaluating them on a random argument is due to Freivalds [Fre77]. But one cannot successfully apply this concept for any equivalence problem. For instance, given two representations of two Boolean functions that differ in one of the 2^n possible arguments, one cannot straightforwardly apply the Freivalds technique. But it is possible to efficiently transform some Boolean function representations to polynomials over an appropriate field, and so to reduce the problem of Boolean function comparison to the presented comparison of two polynomials. A detailed and transparent presentation of this involved application of fingerprinting is given in the excellent textbook by Sipser [Sip97] and also in [Hro03]. Further nontrivial applications of fingerprinting are presented in the monograph by Motwani and Raghavan [MR95].

5

Success Amplification and Random Sampling



It is not possible to wait for inspiration,
and even inspiration alone is not sufficient.
Work and more work is necessary.
Man blessed by genius can create nothing really great,
not even anything mediocre,
if he does not toil as hard as a slave.

Piotr Ilyich Tschaikovsky

5.1 Objectives

This chapter is devoted to two paradigms of the design of randomized algorithms, namely the amplification of success probability by repeating runs on the same input and random sampling. The reasons for presenting both these methods in one chapter are their similarity and their equally balanced combination in several applications. Thus, for some randomized algorithms, it is not possible to determine which of these two methods is primarily responsible for success.

In Chapter 2 we called attention to the fact that amplification is a method for reducing the error probability below an arbitrarily given small constant $\epsilon > 0$. We underlined the importance of this observation by classifying randomized algorithms with respect to the speed of error probability reduction with the number of computation repetitions on the same input. In this chapter we aim to present algorithms for which amplification does not only increase the success probability, but directly stamps the process of the algorithm design. Moreover, we do not want only to follow the naive approach of repeating the whole computation on the same input, but also to introduce a more advanced technique that prefers to repeat only some computation parts or to repeat different parts differently many times. The idea is to pay more attention to computation parts in which the probability of making mistakes is greater than in other ones.

With random sampling we aim to document the power of this method by designing efficient randomized algorithms solving problems for which no deterministic polynomial-time algorithm has up to now been discovered.[1]

This chapter is organized as follows. Section 5.2 introduces the above mentioned generalized version of the amplification method. This method is used

[1]and maybe for which no efficient deterministic algorithms exist at all

to design randomized algorithms solving the minimum cut problem for multigraphs. In Section 5.3 we combine amplification with random sampling in order to design a practicable one-sided-error Monte Carlo algorithm for the well known, NP-hard 3-satisfiability (3SAT) problem. Though this algorithm runs in exponential time, it is much faster than algorithms running in $O(2^n)$ time and it can be successfully applied for relatively large instances of 3SAT. In Section 5.3 we present an application of random sampling that results in a Las Vegas polynomial-time algorithm for a number-theoretic problem, which is not known to be in P. Hence, this randomized algorithm is the only efficient way known for solving this problem. Altogether this chapter presents impressive examples documenting the superiority of randomized algorithms over their best known deterministic counterparts. As usual, we finish the chapter by summarizing the most important ideas and results presented.

5.2 Efficient Amplification by Repeating Critical Computation Parts

The aim of this section is to introduce the method of amplification of the success probability as a method for the design of randomized algorithms, and not only (as considered until now) as a technique for error probability reduction of algorithms already designed. For this purpose we consider the following minimization problem MIN-CUT.

MIN-CUT

Input: A multigraph $G = (V, E, c)$, where $c : E \to \mathbb{N} - \{0\}$ determines the multiplicity of the edges of G.

Constraints: The set of all feasible solutions for G is the set

$$\mathcal{M}(G) = \{(V_1, V_2) \mid V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset\}$$

of all cuts of G.

Costs: For every cut $(V_1, V_2) \in \mathcal{M}(G)$,

$$\text{cost}\,((V_1, V_2), G) = \sum_{e \in S(V_1,V_2)} c(e),$$

where $S(V_1, V_2) = \{\{x, y\} \in E \mid x \in V_1 \text{ and } y \in V_2\}$

$\{$i.e., $\text{cost}\,((V_1, V_2), G)$ is equal to the number of edges between V_1 and $V_2\}$

Goal: minimum

The best known deterministic algorithm for MIN-CUT runs in time

$$O\left(|V| \cdot |E| \cdot \log\left(\frac{|V|^2}{|E|}\right)\right),$$

which, in the worst case[2], is in $O(n^3)$ for $n = |V|$. Our goal is to design an efficient randomized algorithm for MIN-CUT. This algorithm is based on the graph operation **Contract** $(\boldsymbol{G}, \boldsymbol{e})$ that, for a given multigraph $G = (V, E)$ and an edge $e = \{x, y\} \in E$, contracts the edge e. The contraction of $e = \{x, y\} \in E$ means that

- the vertices x and y are replaced by a new vertex $\text{ver}(x, y)$,
- the multi-edge $e = \{x, y\}$ is removed (contracted) in this way (we do not allow any self loop),
- each edge $\{r, s\}$ with an $r \in \{x, y\}$ and $s \notin \{x, y\}$ is replaced by a new edge $\{\text{ver}(x, y), s\}$, and
- all remaining parts of G remain unchanged.

Visualizing $\text{Contract}(G, e)$, one simply collapses the vertices x and y into one new vertex. We denote the resulting graph by $G/\{e\}$.

Figure 5.1 shows three contraction operations consecutively executed on the multigraph G depicted in Figure 5.1(a). First, one executes the operation $\text{Contract}(G, \{x, y\})$. The resulting multigraph $G/\{x, y\}$ is depicted in Figure 5.1(b). The next step is the contraction of the edge $\{u, z\}$, and the resulting multigraph $(G/\{x, y\})/\{u, z\}$ is depicted in Figure 5.1(c). Finally. one contracts the edge $\{\text{ver}(x, y), v\}$ and obtains the multigraph of two vertices[3] $\text{ver}(x, y, v)$ and $\text{ver}(u, z)$ depicted in Figure 5.1(d).

Observe that, given a collection of edges $F \subseteq E$, the effect of contracting the edges in F does not depend on the order of contractions. For instance, the multigraph $(G/\{u, z\})/\{x, y\}$ is the same as $(G/\{x, y\})/\{u, z\}$ in Figure 5.1. Therefore we simplify the notation and use G/F for the resulting multigraph. This way, $G/\{\{x, y\}, \{u, z\}, \{x, v\}\}$ is a short representation of the multigraph in Figure 5.1(d).

The design idea for the following naive randomized algorithm for MIN-CUT is very simple. One contracts randomly chosen edges until one gets a multigraph with exactly two vertices $\text{ver}(V_1)$ and $\text{ver}(V_2)$. Obviously, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. Hence, (V_1, V_2) is a cut of G, and the number of edges between the two vertices $\text{ver}(V_1)$ and $\text{ver}(V_2)$ corresponds to the cost of the cut (V_1, V_2).

In our example, the multigraph in Figure 5.1(d) corresponds to the cut $(\{x, y, v\}, \{u, z\})$ with a cost of 4. In other words, one can view the contraction of an edge $\{x, y\}$ as saying that the vertices x and y must be on the same side of the cut approached and so as restricting the set of all cuts to the cuts with x and y on one side. In this way the set of possible cuts is monotonically reduced until it contains only one cut.

[2]if $|E| \in \Omega(|V|^2)$

[3]To be precise, one has to write $\text{ver}(\text{ver}(x, y), v)$ instead of $\text{ver}(x, y, v)$. But we prefer the shorter representation because, in what follows, the only important matter is that the vertices x, y, and v are joined to one vertex. To simplify the notation, we even use $\textbf{ver}(\boldsymbol{V'})$ for the vertex created by the union of the vertices in V'.

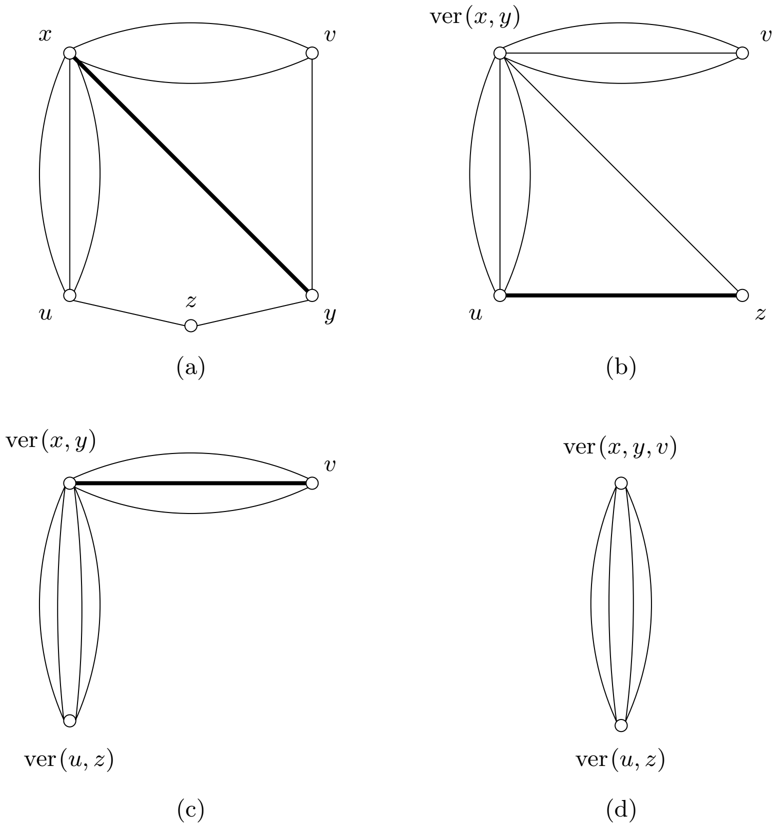


Fig. 5.1.

One can formally describe this approach for solving MIN-CUT as follows. Let $E(G)$ denote the set of edges of a multigraph G.

Algorithm CONTRACTION

```
Input: A connected[4] multigraph $G = (V, E, c)$
Step 1: Set label $(v) := \{v\}$ for every vertex $v \in V$.
Step 2:
    while $G$ has more than two vertices do
        begin
            choose an edge $e = \{x, y\} \in E(G)$;
            $G :=$ Contract $(G, e)$;
            Set label $(z) :=$ label $(x) \cup$ label $(y)$
```

[4]One can determine in time $O(|E|)$ whether or not a multigraph is connected, and so find a minimal cut of cost 0 for any disconnected multigraph.

```
            for the new vertex $z = \text{ver}(x, y)$;
        end
Step 3:
    if $G = (\{u, v\}, E(G))$ for a multiset $E(G)$ then
        output "(label(u), label(v))" and " cost = $|E(G)|$"
```

Theorem 5.2.1. *The algorithm* CONTRACTION *is a randomized polynomial-time algorithm that computes a minimal cut of a given multigraph G of n vertices with probability at least*

$$\frac{2}{n \cdot (n-1)}.$$

Proof. Clearly, the algorithm CONTRACTION computes a cut of G, and so a feasible solution for MIN-CUT. The algorithm executes exactly $n-1$ edge contractions for a multigraph of n vertices, and each contraction can be executed in $O(n)$ steps. Hence, the time complexity of CONTRACTION is in $O(n^2)$.

In what follows, we aim to show that the algorithm CONTRACTION finds a minimal cut with probability at least $\frac{2}{n\cdot(n-1)}$. Let $G = (V, E, c)$ be a multigraph, and let $C_{\min} = (V_1, V_2)$ be a minimal cut of G with $\text{cost}(C_{\min}) = k$ for a natural number k. Let $E(C_{\min})$ denote the set of edges in $C_{\min}$.

In order to prove the lower bound $\frac{2}{n\cdot(n-1)}$ on the success probability, we show that the probability of computing exactly the cut $C_{\min}$ is at least $\frac{2}{n\cdot(n-1)}$.

First, we observe that

G has at least $\frac{n\cdot k}{2}$ edges, i.e., $|E(G)| \geq \frac{n\cdot k}{2}$,

because the minimality of $C_{\min}$ with $\text{cost}(C_{\min}) = k$ implies that every vertex of G has a degree of at least k.

The second important observation is that

the algorithm CONTRACTION *computes $C_{\min}$ if and only if no edge from $E(C_{\min})$ has been contracted.*

Our aim is to study the probability of this event.

The algorithm consists of $n-2$ contractions. Let $S_{\text{Con},G}$ be the set of all possible computations of the algorithm CONTRACTION on G. In the probability space $(S_{\text{Con},G}, \text{Prob})$ we investigate the events

$$\begin{aligned}\text{Event}_i = \{&\text{all computations from } S_{\text{Con},G} \text{ in which no edge} \\ &\text{of } E(C_{\min}) \text{ is contracted in the } i\text{-th contraction step}\}\end{aligned}$$

for $i = 1, 2, \ldots, n-2$. The event that $C_{\min}$ is the output of the algorithm is exactly the event

$$\bigcap_{i=1}^{n-2} \text{Event}_i.$$

Now, we apply the concept of conditional probabilities (Exercise 2.2.16) in order to calculate the probability of this event.

$$\begin{aligned}\mathrm{Prob}\left(\bigcap_{i=1}^{n-2}\mathrm{Event}_i\right) &= \mathrm{Prob}(\mathrm{Event}_1)\cdot\mathrm{Prob}(\mathrm{Event}_2 \mid \mathrm{Event}_1)\\ &\quad\cdot\mathrm{Prob}(\mathrm{Event}_3 \mid \mathrm{Event}_1 \cap \mathrm{Event}_2)\cdot\ldots\\ &\quad\cdot\mathrm{Prob}\left(\mathrm{Event}_{n-2} \,\middle|\, \bigcap_{j=1}^{n-3}\mathrm{Event}_j\right) \qquad (5.1)\end{aligned}$$

To prove Theorem 5.2.1, we have to estimate lower bounds on

$$\mathrm{Prob}\left(\mathrm{Event}_i \,\middle|\, \bigcap_{j=1}^{i-1}\mathrm{Event}_j\right)$$

for $i = 1, \ldots, n-2$.

Since G has at least $\frac{n\cdot k}{2}$ edges and the algorithm makes a random choice for edge contraction,

$$\begin{aligned}\mathrm{Prob}(\mathrm{Event}_1) &= \frac{|E| - |E(C_{\min})|}{|E|}\\ &= 1 - \frac{k}{|E|}\\ &\geq 1 - \frac{k}{\frac{k\cdot n}{2}} = 1 - \frac{2}{n}. \qquad (5.2)\end{aligned}$$

In general, the multigraph G/F_i created after $i-1$ random contractions has exactly $n-i+1$ vertices. If

$$F_i \cap E(C_{\min}) = \emptyset \text{ (i.e., } \bigcap_{j=1}^{i-1}\mathrm{Event}_j \text{ happens)},$$

then $C_{\min}$ is also a minimal cut of G/F_i. Consequently, every vertex in G/F_i has still to have a degree of at least k, and so G/F_i has at least

$$\frac{k\cdot(n-i+1)}{2}$$

edges. Therefore,

$$\begin{aligned}\mathrm{Prob}\left(\mathrm{Event}_i \,\middle|\, \bigcap_{j=1}^{i-1}\mathrm{Event}_j\right) &\geq \frac{|E(G/F_i) - E(C_{\min})|}{|E(G/F_i)|}\\ &\geq 1 - \frac{k}{\frac{k\cdot(n-i+1)}{2}} \qquad (5.3)\\ &= 1 - \frac{2}{(n-i+1)}\end{aligned}$$

for $i = 2, \ldots, n-1$. Inserting (5.3) in (5.1), one obtains

$$\begin{aligned}
\operatorname{Prob}\left(\bigcap_{j=1}^{n-2} \text{Event}_j\right) &\geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) \\
&= \prod_{l=n}^{3}\left(\frac{l-2}{l}\right) \\
&= \frac{2}{n\cdot(n-1)} = \frac{1}{\binom{n}{2}}.
\end{aligned}$$

□

Exercise 5.2.2. Modify the algorithm CONTRACTION in the following way. Instead of choosing an edge at random, choose two vertices x and y randomly and join them into one vertex. Construct multigraphs of n vertices, for which the probability that the modified algorithm finds a minimal cut is exponentially small in n.

Theorem 5.2.1 assures that the probability of discovering a particular minimal cut in one run is at least

$$\frac{2}{n\cdot(n-1)} > \frac{2}{n^2}.$$

Executing $n^2/2$ independent runs of the algorithm and taking the best output from all computed outputs, one does not obtain a minimal cut with a probability of at most

$$\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{e}.$$

Hence, the complementary probability of computing a minimal cut by $n^2/2$ runs of the algorithm is at least

$$1 - \frac{1}{e}.$$

The complexity of the algorithm CONTRACTION$_{n^2/2}$ is in

$$O(n^4).$$

Thus, we need $O(n^4)$ time to compute an optimal solution with a constant probability, and the best known deterministic algorithm computes an optimal solution in time $O(n^3)$ with certainty. The gain of our effort is negative, and one can ask whether it is reasonable to use randomization in this case. But it is not so bad as it seems to be at first glance. The too high complexity is caused by the naive application of the amplification method, in which one increases the success probability by repeating entire runs of the algorithm.

In this case, this standard application of amplification is not clever because the probability of contracting an edge from $C_{\min}$ grows with the number of contractions executed. For the first contractions, this probability[5] is only

$$\frac{2}{n}, \frac{2}{n-1}, \frac{2}{n-2}, \frac{2}{n-3}, \frac{2}{n-4}, \ldots,$$

but for the last edge contractions, it is even as much as 2/3. A very natural idea is to simply give up the last[6] random contractions because the created multigraph G/F, is small enough to be searched for a minimal cut in a deterministic way. In what follows, we describe the algorithm based on this idea. The size of G/F will still remain a free parameter of the algorithm. Let $l : \mathbb{N} \to \mathbb{N}$ be a monotonic function such that $1 \leq l(n) < n$ for every $n \in \mathbb{N}$.

Algorithm DETRAN(l)

```
Input: A multigraph $G = (V, E, c)$ of $n$ edges, $n \in \mathbb{N}$, $n \geq 3$.
Step 1: Perform the algorithm CONTRACTION on $G$ in order to get a multi-
    graph $G/F$ of $l(n)$ vertices.
Step 2: Apply the best known deterministic algorithm on $G/F$ to compute
    an optimal cut $D$ of $G/F$.
Output: $D$
```

First, we analyze the influence of the exchange of CONTRACTION for DETRAN(l) on

(i) the amplification of the success probability, and
(ii) the increase of the complexity[7].

In this analysis we consider l as a free parameter, i.e., the result of the analysis depends on l.

Lemma 5.2.3. *Let $l : \mathbb{N} \to \mathbb{N}$ be a monotonic growing function with $1 \leq l(n) < n$. The algorithm* DETRAN(l) *works in time*

$$O\big(n^2 + (l(n))^3\big),$$

and it finds an optimal solution with probability at least

$$\frac{\binom{l(n)}{2}}{\binom{n}{2}}.$$

[5]How many contractions have to be considered as the last ones will be analyzed in what follows.

[6]of the events $E_1, E_2, E_3, \ldots$

[7]The increase of time complexity is the cost we pay for the increase of the success probability.

Proof. First we analyze the complexity of DETRAN(l). In step 1, $n - l(n)$ contractions are performed, and each contraction can be executed in time $O(n)$. Hence, step 1 can be executed in

$$O((n - l(n)) \cdot n) = O\big(n^2\big)$$

time. The above mentioned deterministic algorithm can compute an optimal cut of G/F in time $O\big((l(n))^3\big)$. Altogether, the complexity of DETRAN(l) is in

$$O\big(n^2 + (l(n))^3\big)\,.$$

Next we analyze the success probability of DETRAN(l). As in the proof of Theorem 5.2.1, let $C_{\min}$ be a minimal cut of G. We prove a lower bound on the success probability of DETRAN(l) by proving a lower bound on the probability of having $C_{\min}$ in the multigraph G/F after executing step 1. The corresponding event is

$$\bigcap_{i=1}^{n-l(n)} \text{Event}_i.$$

Applying (5.1), (5.2), and (5.3), one obtains the following lower bound on the probability of this event, and so also on the probability of computing a minimal:

$$\begin{aligned}
\text{Prob}\left(\bigcap_{i=1}^{n-l(n)} \text{Event}_i\right) &\geq \prod_{i=1}^{n-l(n)} \left(1 - \frac{2}{n-i+1}\right) \\
&= \frac{\prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right)}{\prod_{j=n-l(n)+1}^{n-2}\left(1 - \frac{2}{n-j+1}\right)} \\
&= \frac{\frac{1}{\binom{n}{2}}}{\frac{1}{\binom{l(n)}{2}}} = \frac{\binom{l(n)}{2}}{\binom{n}{2}}.
\end{aligned}$$

This completes the proof. □

Since

$$\frac{n^2}{(l(n))^2} \geq \frac{\binom{n}{2}}{\binom{l(n)}{2}},$$

$\frac{n^2}{(l(n))^2}$ independent runs of DETRAN(l) provide a randomized algorithm that works in time

$$O\left((n^2 + (l(n))^3) \cdot \frac{n^2}{(l(n))^2}\right) = O\left(\frac{n^4}{(l(n))^2} + n^2 \cdot l(n)\right) \tag{5.4}$$

and computes a minimal cut of G with probability at least

$$1 - \frac{1}{e}.$$

The best[8] possible choice of l with respect to the time complexity is

$$l(n) = \left\lfloor n^{2/3} \right\rfloor,$$

and making this choice one obtains the following result.

Theorem 5.2.4. *The algorithm* DETRAN$\left(\lfloor n^{2/3} \rfloor\right)_{n^2/\lfloor n^{2/3} \rfloor}$ *works in time*

$$O\left(n^{8/3}\right)$$

and computes a minimal cut with probability at least

$$1 - e^{-1}.$$

Now, we have a randomized algorithm that is asymptotically faster than any known deterministic algorithm, and whose error probability can be pushed arbitrarily[9] low. Though we are not satisfied with the designed randomized algorithm. The design strategy is too simple and rough for really capturing the growing error probability of an edge contraction with the number of contractions executed. The algorithm DETRAN(l) prohibits the execution of the last random contractions with high error probabilities by computing deterministically in its final part, and so increases the success probability. Hence, fewer runs of DETRAN(l) suffice for getting a constant probability of computing a minimal cut.

If one observes the increase

$$\frac{2}{n}, \frac{2}{n-1}, \frac{2}{n-2}, \frac{2}{n-3}, \ldots, \frac{2}{3}$$

of the error probabilities of the sequence of random contractions, the idea of using fewer run repetitions at the beginning and more at the end of the algorithm CONTRACTION may appear appropriate. We visualize this idea in what follows. The algorithm CONTRACTION needs n^2 runs of $O(n^2)$ complexity to assure a minimal cut with a constant probability. This corresponds to Figure 5.2, where one lets n^2 computations of CONTRACTION run in parallel. If the complexity of each computation is in $O(n^2)$, then the complexity of the entire work is the area of the rectangle of the size $n^2 \times O(n^2)$, and so $O(n^4)$.

Next, we consider the following implementation of our new idea (Figure 5.3). We start by executing only two independent runs in parallel. After some random edge contractions, when the error probability of choosing a wrong edge[10] has grown substantially, we split each of the two computations

[8]To minimize the expression (5.4), one has to take an l such that $\frac{n^4}{(l(n))^2} = n^2 \cdot l(n)$.

[9]The failure probability tends to 0 with exponential speed with respect to the number of repetitions of DETRAN$\left(\lfloor n^{2/3} \rfloor\right)_{n^2/\lfloor n^{2/3} \rfloor}$.

[10]an edge of $C_{\min}$

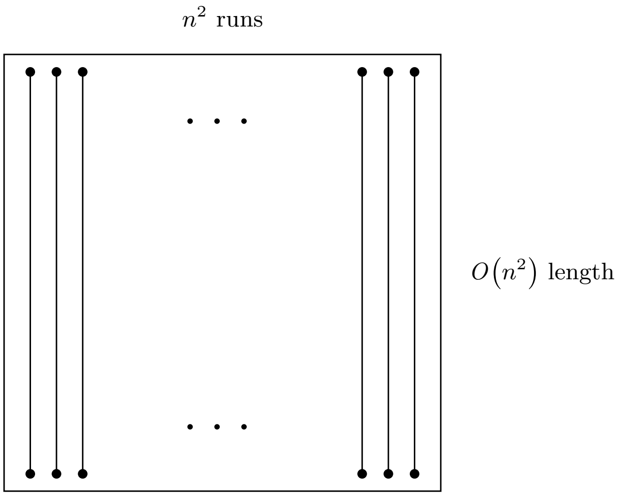


Fig. 5.2.

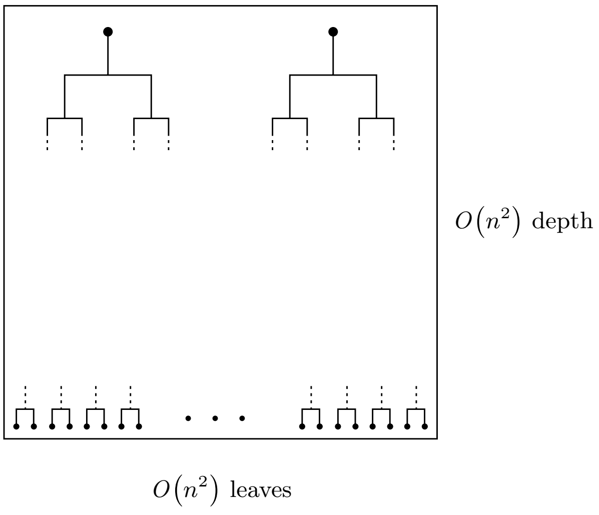


Fig. 5.3.

into two independent runs. After some further contractions we again double the number of runs, and so on. At the very end we also have $O(n^2)$ runs (as in Figure 5.2, too) and we take the best cut of the computed cuts. Hence, the strategy is to use at the end as many runs as in the naive approach of the algorithm CONTRACTION, but to use only a few at the beginning (when the probability of choosing an edge from $C_{\min}$ is small). The crucial point for the complexity analysis of this strategy is estimating the number of random contractions between doubling the number of runs executed in parallel. We double the number of runs always after the number of vertices has been reduced by a factor of $1/\sqrt{2}$ after the last doubling. Since the number of all random contractions is $n-2$, the number of runs (leaves of the tree in Figure 5.3) is really $2^{\log_{\sqrt{2}}(n-2)} \in O(n^2)$. Observing Figure 5.3 we see that the complexity of this strategy is the sum of the lengths of all tree edges of the two trees in Figure 5.3. In contrast to the area of the figure rectangle[11], the trees look light, and so one can expect a substantially smaller complexity. Moreover, if one realizes that the number of edges in the complete binary tree of $O(n^2)$ leaves is in $O(n^2 \cdot \log_2 n^2)$ and that the length of the edges decreases with the number of parallel runs, then one can hope for a complexity in $O(n^2 \log n)$. To recognize that the success probability of this strategy is large enough is a little bit more complicated. Before doing this, we give a formal presentation of the algorithm.

Algorithm REPTREE(*G*)

```
Input: A multigraph $G = (V, E, c)$, $|V| = n$, $n \in \mathbb{N}$, $n \geq 3$.
Procedure:
    if $n \leq 6$ then
        compute a minimal cut deterministically
    else
        begin
            $h := \lceil 1 + \frac{n}{\sqrt{2}} \rceil$;
            Perform two independent runs of CONTRACTION in order to
            get two multigraphs $G/F_1$ and $G/F_2$ of size $h$;
            REPTREE$(G/F_1)$;
            REPTREE$(G/F_2)$
        end
    output the smaller of the two cuts computed by
            REPTREE$(G/F_1)$ and REPTREE$(G/F_2)$
```

Theorem 5.2.5.* *The algorithm* REPTREE *works in time*

$$O(n^2 \cdot \log n)$$

and finds a minimal cut with a probability of at least

$$\frac{1}{\Omega(\log_2 n)}.$$

Proof. First we analyze the time complexity of REPTREE. The depths of the binary trees in Figure 5.3 correspond to the number of recursion calls of

[11] which is in $O(n^4)$

REPTREE. Since the size of a given multigraph is reduced by the multiplicative factor $1/\sqrt{2}$ in each stage of the algorithm, the number of recursion calls is at most

$$\log_{\sqrt{2}} n \in O(\log_2 n)\,.$$

Since the original algorithm CONTRACTION works in time $O\big(n^2\big)$ on any multigraph of n vertices, we can roughly bound the time of reducing a multigraph G of size m to a multigraph G/F of size $\lceil 1 + \frac{m}{\sqrt{2}} \rceil$ by $O\big(m^2\big)$. Hence, we obtain the following recurrence for $\mathrm{Time}_{\mathrm{REPTREE}}$:

$$\begin{aligned} &\mathrm{Time}_{\mathrm{REPTREE}}(n) \in O(1) \text{ for } n \le 6, \text{ and} \\ &\mathrm{Time}_{\mathrm{REPTREE}}(n) = 2 \cdot \mathrm{Time}_{\mathrm{REPTREE}}\left(\left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil\right) + O\big(n^2\big)\,. \end{aligned} \tag{5.5}$$

One can easily check that

$$\mathrm{Time}_{\mathrm{REPTREE}}(n) = \Theta(n^2 \cdot \log_2 n).$$

In what follows, we estimate a lower bound on the success probability of the algorithm REPTREE. Once again, we do this by analyzing the probability of computing a specific minimal cut $C_{\min}$ with $|E(C_{\min})| = k$ for a positive integer k.

First, we pose the following question. Let p_l be the probability that the multigraph G/F_i $(i = 1, 2)$ of size $\lceil 1 + (l/\sqrt{2}) \rceil$ still contains $C_{\min}$, assuming that the multigraph G/F of l vertices has contained $C_{\min}$ before the computation split into two runs leading to G/F_1 and G/F_2. How large is p_l? Following the analysis of the algorithm DETRAN, we have

$$p_l \ge \frac{\binom{\lceil 1+\frac{l}{\sqrt{2}} \rceil}{2}}{\binom{l}{2}} = \frac{\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil \cdot \left(\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil - 1\right)}{l \cdot (l-1)} \ge \frac{1}{2}.$$

Let $\mathrm{Prob}(n)$ be the probability that REPTREE finds a minimal set of a multigraph of n vertices. Then, for $i = 1, 2$,

$$p_l \cdot \mathrm{Prob}\left(\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil\right)$$

is a lower bound on the conditional probability that REPTREE computes $C_{\min}$ by the reduction from G/F to G/F_i and then by the recursive call $\mathrm{REPTREE}(G/F_i)$, assuming G/F has contained $C_{\min}$. Since REPTREE starting from G/F executes two runs of CONTRACTION with outputs G/F_1 and G/F_2,

$$\left(1 - p_l \cdot \mathrm{Prob}\left(\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil\right)\right)^2$$

is an upper bound on the conditional probability that REPTREE does not find $C_{\min}$ assuming G/F has contained $C_{\min}$. Hence, we obtain the following recurrence for $\mathrm{Prob}(n)$:

$$\begin{aligned}
\text{Prob}(2) &= 1, \text{ and} \\
\text{Prob}(l) &\geq 1 - \left(1 - p_l \cdot \text{Prob}\left(\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil\right)\right)^2 \\
&\geq 1 - \left(1 - \frac{1}{2} \cdot \text{Prob}\left(\left\lceil 1 + \frac{l}{\sqrt{2}} \right\rceil\right)\right)^2 \qquad (5.6)
\end{aligned}$$

One can show that each function Prob satisfying the recurrence (5.6) is in $\Theta(\frac{1}{\log_2 n})$. □

Exercise 5.2.6. Prove that the solution of the recurrence (5.5) is a function in $\Theta(n^2 \cdot \log n)$.

Exercise 5.2.7. Prove that each function Prob satisfying the recurrence (5.6) is a function that fulfills $\text{Prob}(n) \geq \frac{1}{\Omega(\log_2 n)}$.

Exercise 5.2.8.* Analyze the success probability and the time complexity of the versions of the algorithm REPTREE, for which one takes the following size reduction between two splits of the computation:

(i) from l to $\left\lfloor \frac{l}{2} \right\rfloor$
(ii) from l to $\sqrt{l}$
(iii) from l to $\frac{l}{\log_2 l}$
(iv) from l to $l - \sqrt{l}$.

A consequence of Theorem 5.2.5 is that $O(\log_2 n)$ repetitions of the algorithm REPTREE are sufficient in order to compute a minimal cut with a constant probability. Already $O\big((\log_2 n)^2\big)$ repetitions suffice to reduce the non-success probability to a function tending to 0 with growing n and one can consider this algorithm applicable. The complexity of $\text{REPTREE}_{(\log_2 n)^2}$ is in

$$O\big(n^2 \cdot (\log_2 n)^3\big)$$

which is substantially better than the complexity $O\big(n^3\big)$ of the best deterministic algorithm and the complexity $O\big(n^{8/3}\big)$ of the randomized algorithm $\text{DETRAN}\left(\lfloor n^{2/3} \rfloor\right)_{n^2/\lfloor n^{2/3} \rfloor}$.

Hence, the idea of repeating different parts differently many times can be very fruitful.

5.3 Repeated Random Sampling and Satisfiability

Here we combine amplification and random sampling with local search in order to design a randomized algorithm that can solve the 3-satisfiability problem[12]

[12]Remember that the instances of 3SAT are formulas in 3CNF, and the task is to decide whether or not a given formula is satisfiable.

(3SAT) for instances of nontrivial size in acceptable time. 3SAT is a known NP-hard problem, and we call attention to the fact that until now no randomized polynomial-time algorithm with a bounded error has been discovered for any NP-hard problem. Our experience with general[13] polynomial-time computations point to the commonly accepted conjecture that NP-hard problems are also hard for randomized polynomial time and so one does not hope for exponential gaps in the time complexity between determinism and randomization[14]. Therefore in the case of the NP-hard 3SAT problem, we aim to design a practicable exponential randomized algorithm. Table 5.1 shows that the design of exponential algorithms can be a reasonable and worthy objective. The values 10, 50, 100, and 300 in the first row represent the input sizes, and the first column contains the running times considered. The particular items in the table give the number of corresponding computer operations. If a number is too large, we provide its number of digits only.

Table 5.1.

$f(n)$ \ n	10	50	100	300
$n!$	$\approx 3.6 \cdot 10^6$	(65 digits)	(158 digits)	(625 digits)
2^n	1024	(16 digits)	(31 digits)	(91 digits)
$2^{n/2}$	32	$\approx 33 \cdot 10^6$	(16 digits)	(46 digits)
$(1.2)^n$	≈ 6.19	9100	$\approx 8.2 \cdot 10^7$	(24 digits)
$10 \cdot 2^{\sqrt{n}}$	≈ 30	≈ 1345	10240	$\approx 1.64 \cdot 10^6$
$n^2 \cdot 2^{\sqrt{n}}$	895	≈ 336158	$1.024 \cdot 10^7$	$\approx 1.48 \cdot 10^{11}$
n^6	10^6	$1.54 \cdot 10^{10}$	10^{12}	$\approx 7.29 \cdot 10^{14}$

If one considers approximately 10^{16} operations as the boundary of practically doable, then one observes that algorithms with a running time like $n!$ or 2^n are already not applicable for small input sizes. But algorithms with $2^{n/2}$ time complexity are useful for problem instances of size 100 and, for a time complexity of $(1.2)^n$, one can go still further. Algorithms with an exponential time complexity such as $10 \cdot 2^{\sqrt{n}}$ or $n^2 \cdot 2^{\sqrt{n}}$ can be successfully applied for relatively large problem instances (Table 5.1), and are even faster than polynomial-time algorithms with a time complexity of n^6 for $n \leq 300$. Moreover, these exponential algorithms run on inputs of size less than 300 in a few

[13]General is meant in the sense of unrestricted computing models that represent algorithms.

[14]The most impressive presentation of the computational power of randomization relative to determinism can be found by restricted frameworks of simple models of computation such as finite automata, pushdown automata, communication protocols, etc.

seconds, something that cannot be said about an n^6-algorithm on inputs of size 100.

The aim of this section is to design a 1MC algorithm for 3CNF that works in time

$$O\left(|F| \cdot n^{\frac{3}{2}} \cdot \left(\frac{4}{3}\right)^n\right)$$

for each formula in 3CNF over n variables.[15] This running time is not achievable by pure random sampling. If F is satisfiable, but there are only a few assignments satisfying F, then one would need in average $\Omega(2^n)$ random samples to find one of these satisfying assignments.

Exercise 5.3.9. Let F be a formula of n variables that is satisfied by exactly k assignments to its variables. How many random samples from the set $\{0,1\}^n$ are necessary in order to find an assignment satisfying F with a probability of at least $1/2$?

The idea of the design of the algorithm is simple. We repeat at most

$$O\left(\sqrt{n} \cdot \left(\frac{4}{3}\right)^n\right) \text{ times}$$

the following procedure. Choose one of the 2^n assignments from $\{0,1\}^n$ at random and perform at most $3 \cdot n$ steps of local search in order to find an assignment satisfying F. One step of local search consists of flipping one bit of the current assignment. Which particular bit is flipped is partially decided at random. A detailed description of the algorithm follows.

Algorithm SCHÖNING

```
Input: A formula $F$ in 3CNF over $n$ Boolean variables.
Step 1:
    NUMBER := 0;
    ATMOST := $\lceil 20 \cdot \sqrt{3\pi n} \cdot (\frac{4}{3})^n \rceil$;
    FOUND := FALSE;
    {The variable NUMBER counts the number of random samples
      executed. ATMOST gives the upper bound on the number of ran-
      dom samples we are willing to perform. FOUND indicates whether
      or not an assignment satisfying $F$ was already found.}
Step 2:
    while NUMBER < ATMOST and FOUND = FALSE do
        begin
            NUMBER := NUMBER + 1;
            Generate at random an assignment $\alpha \in \{0,1\}^n$;
            if $F$ is satisfied by $\alpha$ then FOUND := TRUE;
```

[15] $|F|$ denotes the length of the formula F, i.e., the input length. Clearly, $n \leq |F|$.

```
            $M := 0$;
            while $M < 3 \cdot n$ and FOUND = FALSE do
               begin
                  $M := M + 1$;
                  Find a clause $C$ that is not satisfied by $\alpha$.
                  {Since $\alpha$ does not satisfy $F$, such a clause must exist. If there
                   are several such clauses, it does not matter which one is
                   chosen.}
                  Pick one of the literals of $C$ at random, and flip the value of
                  its variable in order to get a new assignment $\alpha$.
                  {Observe that $\alpha$ now satisfies the clause $C$.}
                  if $\alpha$ satisfies $F$ then FOUND := TRUE;
               end
         end
Step 3:
    if FOUND = TRUE then
         output "$F$ is satisfiable"
    else
         output "$F$ is not satisfiable".
```

The main difficulty is to show that random sampling with an enclosed short, randomized local search has s substantially higher probability of success (of finding an assignment satisfying F, assuming F is satisfiable) than the pure random sampling.

Theorem 5.3.10.* *The algorithm* SCHÖNING *is a* 1MC *algorithm for the* 3SAT*-problem that runs in time*

$$O\left(|F| \cdot n^{3/2} \cdot \left(\frac{4}{3}\right)^n\right)$$

for any instance F of n variables.

Proof. First, we analyze the time complexity in the worst case. Step 1 can be performed in time $O\big(n^2 \cdot \log_2 n\big)$ by repeated squaring[16], and step 3 requires only $O(1)$ operations. The random sampling accompanied with the local search described above is executed at most

$$\left\lceil 20 \cdot \sqrt{3\pi n} \cdot \left(\frac{4}{3}\right)^n \right\rceil$$

times. Each local search consists of at most $3 \cdot n$ steps, and each step can be performed in time[17] $O(|F|)$. Hence, the time complexity of the second step (and so the time complexity of the whole algorithm) is in

[16]see Appendix, Section A.2

[17]A formula F can be evaluated in time $O(|F|)$ for every given assignment. During the evaluation procedure, one fixes which clauses are satisfied and which ones are not.

$$O\left(|F| \cdot n^{\frac{3}{2}} \cdot \left(\frac{4}{3}\right)^n\right).$$

In order to show that SCHÖNING is a 1MC algorithm for 3SAT, we analyze its failure probability. As usual, we distinguish two cases with respect to the satisfiability of F.

Let F be not satisfiable. Then, the algorithm SCHÖNING does not find any assignment that satisfies F and outputs the correct answer "F is not satisfiable" with certainty.

Let F be satisfiable. To prove a lower bound on the success probability of SCHÖNING, we proceed in a way similar to that in Section 5.2; namely we analyze the probability of finding a certain assignment α^* that satisfies F. Let p be the probability that the algorithm SCHÖNING finds α^* by a random sample followed by at most $3 \cdot n$ steps of the local search described above. Our first aim is to show that

$$p \geq \frac{1}{2 \cdot \sqrt{3\pi n}} \cdot \left(\frac{3}{4}\right)^n. \tag{5.7}$$

We start by partitioning the set $\{0,1\}^n$ of all assignments of F into $n+1$ classes with respect to their distances to α^*. Let α and β be two assignments. We define the distance **Dist $(\boldsymbol{\alpha}, \boldsymbol{\beta})$** between α and β as the number of bits in which they differ (i.e., as the number of flip operations necessary to get from α to β, or vice versa). For every $j = 0, 1, 2, \ldots, n$ we define the j-th class as

$$\text{Class}\,(j) = \{\beta \in \{0,1\}^n \mid \text{Dist}\,(\alpha^*, \beta) = j\}.$$

These $n+1$ classes are pairwise disjoint and so they build a partitioning of $\{0,1\}^n$. Clearly,

$$\text{Class}\,(0) = \{\alpha^*\},$$

and

$$|\text{Class}\,(j)| = \binom{n}{j}$$

for $j = 0, 1, \ldots, n$. The kernel of our analysis is to investigate the execution of the local search as a run (movement) between these classes, i.e., to mimic the local search by a sequence of corresponding classes. To visualize this we use the graph in Figure 5.4. The vertex i of this graph corresponds to the class $Class(i)$ for $i = 0, 1, \ldots, n$, and we say that the algorithm is situated in the vertex i of the current assignment, α is in $Class(i)$. Our goal is to achieve the vertex 0. Every local step[18] changes exactly one bit of the assignment, and so corresponds to the movement to one of the two neighboring vertices. Hence, executing a local step one can approach the vertex 0 by decreasing the distance by exactly 1, or one increases the distance to the vertex 0 by 1. The only exceptions are vertices 0 and n. One can move from vertex n only in the

[18]performed by flipping a bit of the current assignment

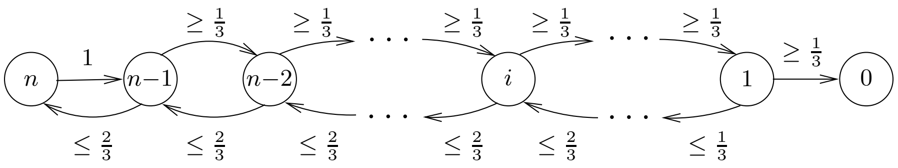


Fig. 5.4.

direction of vertex 0 and we do not want to leave vertex 0 anymore.[19] Now, we are claiming that, for $j = 1, 2, \ldots, n-1$,

the probability of moving from vertex j to vertex $j-1$ in one local search step is at least $\frac{1}{3}$. (5.8)

This can be shown as follows. Let α be the current assignment from Class$\,(j)$ and let C be a clause that is not satisfied by α. The clause C contains at most 3 literals and so C depends on at most 3 variables. The assignment α does not satisfy any of these literals. Since α^* satisfies F, and so C, too, α^* must differ from α in at least one of these variables of C. The algorithm SCHÖNING chooses one of these variables for flipping at random, and so one has probability of at least 1/3 to approach α^* (i.e., to decrease the distance to α^* by 1). Hence, the complementary probability[20] of moving away from α^* (of increasing the distance from α^* by 1) is at most 2/3.

In what follows, we analyze for all i, j, with $i \leq j \leq n$, the probability $q_{j,i}$ that the algorithm SCHÖNING starting in the class Class$\,(j)$ (in the vertex j) reaches α^* (the vertex 0) in exactly $j + 2 \cdot i$ local steps, or more precisely in $j + i$ steps toward α^* and i steps in the opposite direction (i.e., toward the vertex n). Since

$$j + 2 \cdot i \leq 3 \cdot n,$$

such a run of the algorithm is possible. To establish a lower bound on $q_{j,i}$, we describe the movement of the algorithm during the $j + 2 \cdot i$ steps on the graph in Figure 5.4 by a word (string) in $\{+,-\}^{j+2\cdot i}$. A move toward α^* (from the left to the right in Figure 5.4) is represented by $+$, and a move in the opposite direction (from the right to the left in Figure 5.4) is represented by $-$. The number of words over $\{+,-\}$ of length $j + 2 \cdot i$ with exactly i $-$ symbols is

$$\binom{j + 2 \cdot i}{i}.$$

But not every one of these strings corresponds to a possible computation from vertex j to vertex 0 in exactly $j + 2 \cdot i$ steps. Only those words correspond to possible runs for which every suffix contains at least as many $+$ symbols as

[19]If the algorithm achieves vertex 0, then it halts.

[20]As already observed, there is no possibility of remaining in the class Class$\,(j)$.

$-$ symbols. The number of such words is at least one third[21] of the number of strings from $\{+,-\}^{j+2\cdot i}$ with exactly $i+j$ symbols $+$ and so there are at least

$$\frac{1}{j+2i}\cdot\binom{j+2\cdot i}{i} \tag{5.9}$$

such words. Let w be an arbitrary word of this set. The word w can be viewed as the representation of the event $\mathrm{Event}(w)$ containing all computations[22] whose movement in the graph is described by w. Since the symbol $+$ occurs with a probability of at least $1/3$ and the $-$ symbol occurs with a probability of at most $2/3$,

$$\mathrm{Prob}(\mathrm{Event}(w)) \geq \left(\frac{1}{3}\right)^{j+i}\cdot\left(\frac{2}{3}\right)^{i} \tag{5.10}$$

for each w of the string set considered. Combining (5.9) and (5.10), we obtain

$$q_{j,i} \geq \frac{1}{j+2i}\cdot\binom{j+2\cdot i}{i}\cdot\left(\frac{1}{3}\right)^{j+i}\cdot\left(\frac{2}{3}\right)^{i}. \tag{5.11}$$

For $j=0,1,2,\ldots,n$, let q_j be the probability of reaching α^* from an $\alpha \in \mathrm{Class}(j)$ in at most $3\cdot n$ local steps. Since $j+2\cdot i \leq 3\cdot n$ for all $0\leq i\leq j\leq n$, one obtains

$$q_j \geq \sum_{i=0}^{j} q_{j,i} \tag{5.12}$$

for each $j \in \{1,2,\ldots,n\}$. Note that $q_0 = 1$. Inserting (5.11) in (5.12), we obtain

$$\begin{aligned} q_j &\geq \sum_{i=0}^{j}\left[\frac{1}{j+2i}\cdot\binom{j+2\cdot i}{i}\cdot\left(\frac{1}{3}\right)^{j+i}\cdot\left(\frac{2}{3}\right)^{i}\right] \\ &> \frac{1}{3}\cdot\binom{3\cdot j}{j}\cdot\left(\frac{1}{3}\right)^{2\cdot j}\cdot\left(\frac{2}{3}\right)^{j} \end{aligned}$$

{We took the element with $i=j$ of the sum as a lower bound for the whole sum.}

Inserting the Stirling formula (Section A.3)

$$r! = \sqrt{2\pi r}\left(\frac{r}{e}\right)^{r}\cdot\left(1+\frac{1}{12r}+O\left(\frac{1}{r^2}\right)\right) \sim \sqrt{2\pi r}\cdot\left(\frac{r}{e}\right)^{r},$$

[21] We show in Lemma A.3.69 that the number of such words is exactly

$$\binom{j+2i-1}{i} - \binom{j+2i-1}{i-1} = \binom{j+2i}{i}\cdot\frac{1}{j+2i}.$$

[22] as elementary events

we obtain

$$\begin{aligned}
q_j &\geq \frac{1}{3} \cdot \frac{(3 \cdot j)!}{(2 \cdot j)! \cdot j!} \cdot \left(\frac{1}{3}\right)^{2 \cdot j} \cdot \left(\frac{2}{3}\right)^{j} \\
&\geq \frac{1}{3} \cdot \frac{\sqrt{2\pi \cdot 3j} \cdot \left(\frac{3 \cdot j}{e}\right)^{3 \cdot j}}{\sqrt{2\pi \cdot 2j} \cdot \left(\frac{2 \cdot j}{e}\right)^{2 \cdot j} \cdot \sqrt{2\pi j} \cdot \left(\frac{j}{e}\right)^{j}} \cdot \left(\frac{1}{3}\right)^{2 \cdot j} \cdot \left(\frac{2}{3}\right)^{j} \\
&= \frac{1}{3} \cdot \frac{\sqrt{3}}{2 \cdot \sqrt{\pi j}} \cdot \frac{3^{3 \cdot j}}{2^{2 \cdot j}} \cdot \left(\frac{1}{3}\right)^{2 \cdot j} \cdot \left(\frac{2}{3}\right)^{j} \\
&= \frac{1}{2 \cdot \sqrt{3\pi j}} \cdot \left(\frac{1}{2}\right)^{j}. \qquad (5.13)
\end{aligned}$$

Since the random samples (initial assignments) are chosen uniformly, the cardinalities of the classes Class (j) determine the probability p_j, that a random sample is in Class (j), i.e.,

$$p_j = \binom{n}{j} / 2^n. \qquad (5.14)$$

Thus, we obtain

$$\begin{aligned}
p &= \mathrm{Prob}\left(\begin{array}{l}\text{SCHÖNING finds } \alpha^* \text{ by a random sample fol-}\\ \text{lowed by a local search of at most } 3n \text{ steps}\end{array}\right) \\
&\geq \sum_{j=0}^{n} p_j \cdot q_j. \qquad (5.15)
\end{aligned}$$

Inserting (5.13) and (5.14) into (5.15), we get

$$\begin{aligned}
p &> \sum_{j=0}^{n} \left[\left(\frac{1}{2}\right)^{n} \cdot \binom{n}{j} \cdot \left(\frac{1}{2 \cdot \sqrt{3\pi j}} \cdot \left(\frac{1}{2}\right)^{j} \right) \right] \\
&\geq \frac{1}{2 \cdot \sqrt{3\pi n}} \cdot \left(\frac{1}{2}\right)^{n} \cdot \sum_{j=0}^{n} \left[\binom{n}{j} \cdot \left(\frac{1}{2}\right)^{j} \right] \\
&= \frac{1}{2 \cdot \sqrt{3\pi n}} \cdot \left(\frac{1}{2}\right)^{n} \cdot \left(1 + \frac{1}{2}\right)^{n}
\end{aligned}$$

{One can write $\binom{n}{j} \cdot \left(\frac{1}{2}\right)^{j}$ as $\binom{n}{j} \cdot \left(\frac{1}{2}\right)^{j} \cdot 1^{n-j}$ and then apply the binomial formula.}

$$= \frac{1}{2 \cdot \sqrt{3\pi n}} \cdot \left(\frac{3}{4}\right)^{n} = \widetilde{p}. \qquad (5.16)$$

Thus, the probability that the algorithm does not find any assignment satisfying F by one random sample followed by the considered local search is at most

$$1 - \widetilde{p}.$$

Therefore, the error probability after t independent attempts of the algorithm is at most[23]

$$(1-\widetilde{p})^t \leq e^{-\widetilde{p}\cdot t}. \tag{5.17}$$

Taking $t = \text{ATMOST} = 20 \cdot \sqrt{3\pi n} \cdot \left(\frac{4}{3}\right)^n$ and together with (5.16) inserting it into (5.17), one obtains

$$\text{Error}_{\text{SCHÖNING}}(F) \leq (1-\widetilde{p})^t \leq e^{-10} < 5 \cdot 10^{-5}.$$

Thus, we have proved that the algorithm SCHÖNING is a one-sided-error Monte Carlo algorithm for 3SAT. □

We have seen that a clever execution of random sampling can essentially help reduce the number of random samples. The idea presented here can also be extended for the kSAT problem for an arbitrary, fixed positive integer k.

Exercise 5.3.11. Extend the algorithm SCHÖNING for 4SAT. Observe that the lower bound on the probability of moving toward α^* in a local step decreases to 1/4 in this case. How many repetitions of random sampling followed by a local search are necessary to get a constant success probability?

Exercise 5.3.12.* Prove, for every positive integer $k \geq 5$, that the concept of Schöning provides a 1MC-algorithm for kSAT that works in time

$$O\left(|F| \cdot P(n) \cdot \left(2 - \frac{2}{k}\right)^n\right)$$

for a polynomial P.

5.4 Random Sampling and Generating Quadratic Nonresidues

In Section 5.3 we indicated that it is not realistic to try to design bounded-error randomized polynomial-time algorithms for NP-hard problems. But this is far from the claim that there do not exist any randomized polynomial-time algorithms for problems that cannot be solved in deterministic polynomial time. The world between the class P of decision problems solvable in deterministic polynomial time and the class of NP-hard problems can be very rich of problems of distinct difficulty. There are several problems for which

(i) one does not know any deterministic polynomial-time algorithm, and
(ii) there are no proofs presenting their NP-hardness.

[23] Lemma A.3.59

A problem of this kind and of enormous theoretical as well as practical importance is the factorization of natural numbers. Though the best known deterministic algorithms for the factorization run in exponential time, most researchers believe that factorization is not NP-hard. Since we consider the class of problems solvable by bounded-error randomized polynomial-time algorithms instead of the class P as the class of practically solvable problems, the research focuses on the design of efficient randomized algorithms for problems outside of P, but not NP-hard. One does not know any randomized polynomial-time algorithm for factorization,[24] but there are other problems with properties (i) and (ii) for which efficient randomized algorithms were discovered. An example already presented is the 1MC algorithm for the comparison of two polynomials that is based mainly on the fingerprinting method. Here, we go even further and design an efficient Las Vegas algorithm for a problem with properties (i) and (ii).

In what follows, we consider the problem of generating a quadratic nonresidue modulo p for a given prime p. A **quadratic residue** in the field[25] $\mathbb{Z}_p$ is any element $a \in \mathbb{Z}_p$ such that

$$a = x^2 \bmod p$$

for an $x \in \mathbb{Z}_p$.

A **quadratic nonresidue** is any number $b \in \mathbb{Z}_p$ such that

$$d^2 \not\equiv b \pmod{p}$$

for all $d \in \mathbb{Z}_p$, i.e., every element of $\mathbb{Z}_p$ that is not a quadratic residue. The problem of our current interest is the following one:

Generation of a quadratic nonresidue

Input: A prime $p > 2$.
Output: A quadratic nonresidue modulo p.

Usually one considers this problem for primes that are several hundreds of digits long. Hence, it is not possible to check all[26] elements of $\mathbb{Z}_p$, whether they are quadratic residues or not. Note that the size of the input p is the length $\lceil \log_2(p+1) \rceil$ of its binary representation and so $|\mathbb{Z}_p| = p$ is exponential in input size.

[24]Note that we do know of a polynomial-time quantum algorithm for factorization. Currently we are not able to build a quantum computer working with more than a few bits, but quantum computing can be viewed as a generalization of randomized computations, and this viewpoint opens a lot of interesting questions about the limits of physically based computers.

[25]Note that $\mathbb{Z}_p$ is a field iff p is a prime (a direct consequence of Theorem A.2.27).

[26]their number is larger than the number of protons in the known universe.

It is also remarkable that the "dual" problem of generating a quadratic residue is trivial. It takes an arbitrary element x of Z_p and computes the quadratic residue $x^2 \bmod p$. Moreover, 1 is a quadratic residue for every prime p.

Exercise 5.4.13. Determine all quadratic nonresidues for

(i) $p = 5$,
(ii) $p = 11$,
(iii) $p = 17$.

We aim to show that by applying random sampling one can solve this problem efficiently. One takes a few samples from $\{1, 2, \ldots, p-1\}$ at random, and there is a quadratic nonresidue among these samples with a reasonably high probability. To convince the reader that this very simple idea really works, we have to prove the following two facts:

(A) For every prime p and every $a \in \mathbb{Z}_p$, one can efficiently decide (in a deterministic way) whether a is a quadratic residue or a quadratic nonresidue modulo p.
(B) For every prime p exactly half of the elements of $\mathbb{Z}_p - \{0\}$ are quadratic nonresidues, i.e., a random[27] sample from $\{1, 2, \ldots, p-1\}$ provides a quadratic nonresidue with probability $1/2$.

First, we present Euler's criterion in order to prove (A). In what follows we also use the symbol -1 to denote $p-1$ as the inverse element to 1 with respect to $\oplus_{\bmod p}$ in Z_p. All notions and results of the number theory used in what follows are presented in detail in Section A.2.

Theorem 5.4.14. Euler's Criterion
Let p, with $p > 2$, be a prime. For every $a \in \{1, 2, \ldots, p-1\}$,

(i) if a is a quadratic residue modulo p, then

$$a^{(p-1)/2} \equiv 1 \pmod{p},$$

and
(ii) if a is a quadratic nonresidue modulo p, then

$$a^{(p-1)/2} \equiv p-1 \pmod{p}.$$

Proof. Following Fermat's Little Theorem (Theorem A.2.28)

$$a^{p-1} \equiv 1 \pmod{p},$$

i.e.,

$$a^{p-1} - 1 \equiv 0 \pmod{p} \tag{5.18}$$

[27]with respect to the uniform distribution

for all $a \in \{1, 2, \ldots, p-1\}$.

Since $p > 2$ and p is odd, there is a[28] $p' \geq 1$ such that

$$p = 2 \cdot p' + 1. \tag{5.19}$$

Inserting (5.19) into (5.18), one obtains

$$a^{p-1} - 1 = a^{2 \cdot p'} - 1 = (a^{p'} - 1) \cdot (a^{p'} + 1) \equiv 0 \pmod{p}. \tag{5.20}$$

If a product of two integers is divisible by a prime, then one of the factors must be divisible[29] by p. Therefore (5.20) implies

$$a^{(p-1)/2} - 1 \equiv 0 \pmod{p} \text{ or } a^{(p-1)/2} + 1 \equiv 0 \pmod{p},$$

and so

$$a^{(p-1)/2} \equiv 1 \pmod{p} \text{ or } a^{(p-1)/2} \equiv -1 \pmod{p}.$$

In this way we have established that

$$a^{(p-1)/2} \bmod p \in \{1, p-1\} \tag{5.21}$$

for every $a \in \{1, 2, \ldots, p-1\}$. Now, we are ready to prove (i) and (ii).

(i) Let a be a quadratic residue modulo p.
Then there exists an $x \in \mathbb{Z}_p$ such that

$$a \equiv x^2 \pmod{p}.$$

Since Fermat's Little Theorem implies[30] $x^{p-1} \equiv 1 \pmod{p}$, we obtain

$$a^{(p-1)/2} \equiv \left(x^2\right)^{(p-1)/2} \equiv x^{p-1} \equiv 1 \pmod{p}.$$

(ii) Let a be a quadratic nonresidue modulo p.
Following (5.21) it is sufficient to show that

$$a^{(p-1)/2} \bmod p \neq 1.$$

Since $(\mathbb{Z}_p^*, \odot_{\bmod p})$ is a cyclic group,[31] there exists a generator g of $\mathbb{Z}_p^*$. Since a is a quadratic nonresidue, a must be an even power of g, i.e.,

$$a = g^{2 \cdot l + 1} \bmod p$$

for an integer $l \geq 0$. Hence,

---

[28] $p' = (p-1)/2$

[29] This is a direct consequence of the Fundamental Theorem of Arithmetics about the unambiguousity of factorization (Theorem A.2.3).

[30] Theorem A.2.28

[31] Recall that $\mathbb{Z}_p - \{0\} = \mathbb{Z}_p^*$ for every prime p.

$$a^{(p-1)/2} \equiv \left(g^{2 \cdot l+1}\right)^{(p-1)/2} \equiv g^{l \cdot (p-1)} \cdot g^{(p-1)/2} \pmod p. \tag{5.22}$$

The Fermat's Little Theorem implies $g^{p-1} \bmod p = 1$, and so

$$g^{l \cdot (p-1)} \equiv \left(g^{p-1}\right)^l \equiv 1^l \equiv 1 \pmod p. \tag{5.23}$$

Inserting (5.23) into (5.22), we obtain

$$a^{(p-1)/2} \equiv g^{(p-1)/2} \pmod p.$$

Since g is a generator of $(\mathbb{Z}_p^*, \odot_{\bmod p})$, the order of g is p, and so

$$g^{(p-1)/2} \bmod p \neq 1.$$

Thus, $a^{(p-1)/2} \bmod p \neq 1$, too and, following (5.21), we obtain

$$a^{(p-1)/2} \bmod p = -1 = p - 1.$$

□

Euler's Criterion provides a simple way for testing whether an element $a \in \{1, 2, \ldots, p-1\}$ is a quadratic residue or not. It is sufficient to compute the number

$$a^{(p-1)/2}.$$

We know that this number can be computed in $O(\log_2 p)$ operations[32] over $\mathbb{Z}_p$ by the method of repeated squaring (Section A.2). In this way, the proof of (A) is completed.

The following theorem proves assertion (B).

Theorem 5.4.15. *For every odd prime*[33] *p, exactly half of the nonzero elements of $\mathbb{Z}_p$ are quadratic residues modulo p.*

Proof. Let

$$\mathrm{Quad}\,(p) = \{1^2 \bmod p,\ 2^2 \bmod p, \ldots, (p-1)^2 \bmod p\}$$

be the set of all quadratic nonresidues in $\mathbb{Z}_p^*$. Since every element from $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ is either a quadratic residue or a quadratic nonresidue, it is sufficient to show that

$$|\mathrm{Quad}\,(p)| = \frac{(p-1)}{2}. \tag{5.24}$$

We prove the equality (5.24) by separately proving the inequalities[34]

$$|\mathrm{Quad}\,(p)| \leq \frac{(p-1)}{2} \quad \text{and} \quad |\mathrm{Quad}\,(p)| \geq \frac{(p-1)}{2}.$$

[32]in $O\big((\log_2 p)^3\big)$ binary operations

[33]i.e., for every prime $o \geq 3$

[34]Note that for solving our problem by random sampling, it is sufficient to prove the inequality $|\mathrm{Quad}\,(p)| \leq \frac{(p-1)}{2}$.

(i) For every $x \in \{1, 2, \ldots, p-1\}$,

$$(p-x)^2 = p^2 - 2 \cdot p \cdot x + x^2 = p \cdot (p - 2 \cdot x) + x^2 \equiv x^2 \pmod{p}.$$

Therefore,

$$|\text{Quad}\,(p)| \leq \frac{(p-1)}{2}.$$

(ii) For the opposite inequality, it is sufficient to show that the congruence

$$x^2 \equiv y^2 \pmod{p} \tag{5.25}$$

has at most one solution $y \in \{1, 2, \ldots, p-1\}$ different from x. Without loss of generality, we assume $y > x$, i.e., $y = x + i$ for an $i \in \{1, 2, \ldots, p-2\}$. Since

$$y^2 = (x+i)^2 = x^2 + 2 \cdot i \cdot x + i^2,$$

the congruence (5.25) implies

$$2 \cdot i \cdot x + i^2 = i \cdot (2 \cdot x + i) \equiv 0 \pmod{p}.$$

Since $\mathbb{Z}_p$ is a field,[35] and $i \not\equiv 0 \pmod{p}$,

$$2 \cdot x + i \equiv 0 \pmod{p}. \tag{5.26}$$

Since $i = -(2x)$ is the only solution[36] of the congruence (5.26),

$$|\text{Quad}\,(p)| \geq \frac{(p-1)}{2}.$$

□

Now, we are ready to present our algorithm for generating a quadratic nonresidue in Z_p for any given prime p.

Algorithm NQUAD

Input: A prime p.
Step 1: Choose uniformly an $a \in \{1, 2, \ldots, p-1\}$ at random.
Step 2: Compute

$$A := a^{(p-1)/2} \bmod p$$

by the method of repeated squaring.

[35]More precisely, if the product of two elements in a field is 0, then one of the elements must be 0, too. But here one can argue in a simplier way without using algebra. If a product of two integers is divisible by a prime p, then one of the factors must be divisible by p (Corollary A.2.4).

[36]Every element i of the group $(\mathbb{Z}_p, \oplus_{\text{ mod } p})$ has exactly one inverse element with respect to $\oplus_{\text{ mod } p}$. If $2x < p$, then $i = p - (2 \cdot x)$. If $2 \cdot xp$, then $i = -(2 \cdot x - p)$.

Step 3:

```
if $A = p - 1$ then
    output "$a$"
else
    output "?"
```

The above proved claims (A) and (B) imply that

(i) NQUAD does not make any error[37], and
(ii) NQUAD finds[38] a quadratic nonresidue with the probability 1/2.

Hence, NQUAD is a Las Vegas algorithm for generating a quadratic nonresidue. Its running time is in $O(\log_2 p)$ when taking one time unit for executing an operation over $\mathbb{Z}_p$ and in $O\big((\log_2 p)^3\big)$ when counting the number of binary operations.

Exercise 5.4.16. Modify the algorithm NQUAD in such a way that it always halts[39] with a quadratic nonresidue (i.e., the answer "?" never appears). Analyze the expected running time of your modified NQUAD and prove that the probability of executing an infinite computation is 0.

Exercise 5.4.17. Let p be a prime and let $k \leq \log_2 p$ be a positive integer. Design an efficient Las Vegas algorithm that computes k pairwise distinct quadratic nonresidues modulo p.

We have shown above that the problem of generating quadratic nonresidues is not hard because it can be solved efficiently by a randomized algorithm. The main reason for this success is that half the elements of $\{1, 2, \ldots, p-1\}$ are quadratic nonresidues. Hence, one could think that it is possible to find a quadratic nonresidue in $\mathbb{Z}_p - \{0\}$ efficiently by a suitable deterministic search in $\{1, 2, \ldots, p-1\}$. Unfortunately, until now no one has been able to discover a deterministic polynomial-time strategy for finding a quadratic nonresidue modulo p in $\{1, 2, \ldots, p-1\}$. If there does not exist any efficient deterministic algorithm for this task, one could say that the distribution of quadratic nonresidues in $\{1, 2, \ldots, p-1\}$ can be considered random[40] (chaotic). On the other hand, we have to mention that there is an indication for the existence of some structure (order) in Z_p. It is well known that if the famous *extended Riemann's Hypothesis* holds, then Z_p must contain a quadratic nonresidue among its $O\big((\log_2 p)^2\big)$ smallest elements, i.e., a quadratic nonresidue can be found deterministically by simply checking all these smallest elements of Z_p.

[37]This is a direct consequence of the Euler's criterion (Theorem 5.4.14).

[38]This is a consequence of claim (B) (Theorem 5.4.15).

[39]if it halts at all

[40]Recall the discussion about the random distribution of witnesses in the set of witness candidates.

5.5 Summary

In this chapter we have presented amplification not only as a method for reducing the error probability of already designed randomized algorithms, but also as a method that can be the core of algorithm design. The design of randomized algorithms for the MIN-CUT problem exemplifies this point of view on amplification. We have seen that it can be promising to exchange some parts of randomized computations having high error probability with deterministic ones. Repeating different parts of a randomized computation differently many times with respect to their success probability may also save a lot of computer work, and so be very helpful. Both these approaches lead to the design of randomized algorithms for the MIN-CUT problem that are more efficient than any naive application of amplification based on repeating the entire computations on a given input.

Some exponential algorithms are not only successfully applicable for realistic input sizes, but, for some applications, even more efficient than polynomial-time algorithms whose running time is bounded by a polynomial of a higher degree. Therefore it is meaningful to try to design algorithms with running time in $O(p(n) \cdot c^n)$ for a polynomial p and a constant $c < 2$. Combining random sampling with local search we have designed an efficient randomized algorithm for 3SAT that, for any satisfiable formula, finds a satisfying assignment with probability at least $\Omega\left(\left((n^{1/2}) \cdot (4/3)^n\right)^{-1}\right)$. Executing $O(n \cdot (4/3)^n)$ independent runs of this algorithm, one gets a Monte Carlo algorithm for 3SAT. The essential point of this algorithm design is that pure random sampling cannot assure success probability greater than 2^{-n}. Extending random sampling by a suitable local search increases the success probability to $n^{-1/2} \cdot (3/4)^{-n}$.

There are computing tasks for which on the one hand no deterministic polynomial-time algorithms are known, and, on the other hand, no one is able to prove that these problems are NP-hard. This world between P and NP-hardness is the central area for applying randomization. Famous examples of problems in this world are factorization of integers, equivalence of two polynomials, and the generation of quadratic nonresidues. An efficient one-sided-error Monte Carlo algorithm for the equivalence of two polynomials has been presented in Chapter 4. Here, applying random sampling, we have even designed an efficient Las Vegas algorithm for generating a quadratic nonresidue. The method of random sampling is especially successful in situations where one searches for an object with some special properties in a set of objects, in which the objects of interest are randomly distributed and there is an abundance of them. It may also happen that the objects of our interest are not randomly distributed in the set of all objects (i.e., the set has some structure), but we are not able to recognize the order of their distribution. Also, in this case of our poor knowledge, applying random sampling is the best we can do.

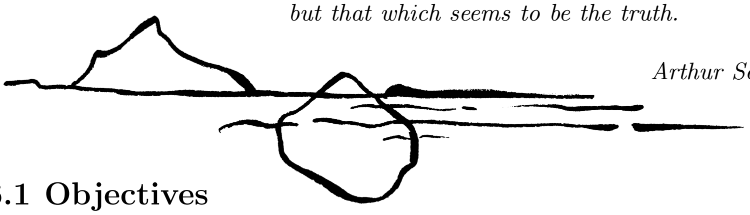
Further successful applications of the methods of random sampling and amplification are presented in the comprehensive textbook of Motwani and

Raghavan [MR95]. Karger [Kar93] designed the algorithm CONTRACTION for MIN-CUT, and the most efficient version was developed by Karger and Stein [KS93]. The exponential randomized algorithm for 3SAT is due to Schöning [Sch99]. The Las Vegas algorithm for generating quadratic non-residues was used by Adleman, Manders, and Miller [AMM77] as a subroutine for computing roots of quadratic residues.

6 Abundance of Witnesses

The greatest obstacle in finding the truth
is not the lie itself,
but that which seems to be the truth.

Arthur Schopenhauer



6.1 Objectives

The method of abundance of witnesses is a real jewel of the design of randomized algorithms. Not only because its applications typically require involved considerations related to nontrivial discoveries of computer science and mathematics, but also, and especially, because of its relation to the fundamental question of whether or not randomization can be more efficient than the deterministic computation mode.

A witness for an input x is additional information, and with its knowledge computing the output for x is substantially more efficient than without this information. To apply the paradigm of abundance of witnesses, one needs, for every problem instance, a set of candidates for a witness, in which there is an abundance[1] of witnesses. Then, it suffices simply to choose an element from the set of candidates at random and one obtains a witness with a reasonable probability. The question of whether, for a given computing task, randomized algorithms can work faster than any deterministic algorithm is strongly related to the question of whether the witnesses are randomly distributed in the set of candidates. If there is an order in the set of candidates, and one can efficiently compute this order from the given problem instance, then one can find a witness efficiently in a deterministic way, too. Only if this is impossible for any kind of witnesses, can the randomized algorithms be essentially more efficient than their deterministic counterparts. Whether a distribution of witnesses in a set of candidates is truly random or not is very hard to decide. It may also happen that there is a structure in the distribution of witnesses, but nobody is able to recognize it. Even if one believes in the existence of an order on the set of candidates but is unable to find it, it is profitable to apply the method of abundance of witnesses. In this way one can efficiently solve problems even though she or he was not successful in discovering the nature of the problem

[1] This usually means that the ratio between the cardinality of the set of candidates and the number of witnesses in the set is a constant.

considered. A good example of this is primality testing. Initially, for hundreds of years no one knew a method[2] better than trying to divide a given integer n by all integers less than or equal to $\sqrt{n}$. In the middle of the 1970s the first efficient randomized algorithms for primality testing were developed, but we had to wait another 25 years to discover a deterministic polynomial-time algorithm.

The aim of this chapter is to explain some basic ideas of and approaches to efficient randomized primality testing. All efficient randomized algorithms for primality testing are based on the method of abundance of witnesses, and so for a long time the central open question in this area was whether, for all kinds of witnesses, the witnesses are indeed randomly distributed in the candidate set or whether one is only unable to find an existing order in their distribution. In 2002 a definition of witnesses was discovered with the property that if p is composite, then there must always be a witness of p's compositeness among the smallest candidates. This discovery resulted in a deterministic algorithm for primality testing running in time $O\big((\log_2 n)^{11}\big)$. Though this fascinating result is of enormous theoretical importance, this deterministic algorithm is no competitor for the fast randomized algorithms in current applications.

In contrast with the natural and easily discovered kinds of witnesses used in the presentation of applications of fingerprinting[3] in Chapter 4, the search for witnesses suitable for primality testing is a little bit more complicated. To provide an accessible start to this search, we begin in Section 6.2 with a few simple ideas that are not sufficient for getting an efficient primality test for all odd integers, but at least for getting most of them. Based on these ideas, we develop in Section 6.3 a randomized algorithm for primality testing that already works for all odd integers. In Section 6.4 we show how one can generate large random primes.

The fundamentals of algebra and number theory presented in the appendix are of key importance for understanding of the topic of this chapter. Especially useful are Fermat's Little Theorem (Theorem A.2.28), the Chinese Remainder Theorem (Theorem A.2.34), and Lagrange's Theorem (Theorem A.2.48).

6.2 Searching for Witnesses for Primality Testing

Here we are trying to solve primality testing, which is the following decision problem. For a given positive integer n, decide whether n is a prime of a composite number. Our aim is to design an efficient, randomized algorithm for primality testing. Since the input length of n is the length $\lceil \log_2(n+1) \rceil$ of its binary representation, we aim to design an algorithm running in time

[2]Observe that the input size of a positive integer n is $\lceil \log_2(n+1) \rceil$, and so $\sqrt{n}$ is exponential in input size.

[3]Remember that the method of fingerprinting can be viewed as a special case of the method of abundance of witnesses and so when applying fingerprinting one may speak about witnesses.

$O((\log_2 n)^c)$ for a small constant c. A small polynomial degree c is a real necessity here, because we have to work for security reasons with primes of several hundreds of digits in practice.[4]

The usual way of defining primes is the following one:

> *A positive integer n is a prime if and only of it does not have any factor (any nontrivial divisor), i.e., if and only if it is not divisible by any number from* $\{2, 3, \ldots, n-1\}$.

This definition provides us the following simple method for testing primality.

Algorithm NAIV

```
Input: A number $n \in \mathbb{N} - \{0, 1, 2\}$.
    $I := 2$
    PRIME := TRUE
    while $I < n$ and PRIME = TRUE do
        begin
            if $n \bmod I = 0$ then PRIME := FALSE;
            $I := I + 1$
        end
    if PRIME = TRUE then
        output "$n$ is a prime
    else
        output "$n$ is composite"
```

We immediately observe that the algorithm NAIV can be improved as follows. Instead of testing all integers from $\{2, 3, \ldots, n-1\}$ for divisibility of n, it suffices to consider the integers from $\{2, 3, \ldots, \lfloor\sqrt{n}\rfloor\}$. This is a direct consequence of the fact that $n = p \cdot q$ implies that the smallest integer of p and q must be smaller or equal to $\sqrt{n}$. But also after this improvement the running time of the algorithm is at least $\sqrt{n} = 2^{(\log_2 n)/2}$, and so exponential in input size. For realistic inputs n of values about 10^{500}, the complexity is incomparably larger than the number of protons in the known universe.

Therefore, we aim to design a randomized algorithm, instead of a deterministic one for primality testing. Our design strategy is based on the paradigm of abundance of witnesses, and so

> *we are searching for a kind of witness that is suitable for an efficient proof of the assertion that "n is composite".*

Recall the following requirements for a suitable definition of witnesses:

[4] In the area of secret communication, especially in cryptographic applications such as online banking or e-commerce.

(i) A witness of the fact "n is composite" has to offer a possibility of efficiently proving this fact.
(ii) Every candidate for a witness must be efficiently checkable, whether or not it is a witness.
(iii) The set of candidates must be specified in such a way that there is an abundance of witnesses in the set of candidates.

First, let us observe that every kind of witness satisfying (i), (ii), and (iii) assures the design of an efficient randomized algorithm. We do not force that the distribution of witnesses in the set of candidates has to be random. The opposite is true; we are happy to discover an order on the set of candidates that could be helpful for efficiently creating a witness deterministically. In such a case, the method of abundance of witnesses would even lead to an efficient deterministic algorithm.

Let PRIM denote the set of all primes. The simplest idea would be to say:

"number $a \in \{2, \ldots, n-1\}$ is a witness of the fact "$n \notin$ PRIM" if and only if a divides n."

Clearly, this definition of a witness fulfills the constraints (i) and (ii). For many integers n, the constraint (iii) on the abundance of witnesses is fulfilled, too. But, for numbers $n = p \cdot q$ for two primes p and q, there are only two such witnesses of the fact "$n \notin$ PRIM". The probability of choosing one of them from the set $\{2, 3, \ldots, n-1\}$ at random is only $2/(n-2)$ and the expected number of random samples for getting a witness is in $\Omega(n)$. Therefore, this definition of a witness is not suitable.

The next attempt to find a good kind of witness is based on Fermat's Little Theorem (Theorem A.2.28).

For every prime p and every $a \in \{1, 2, \ldots, p-1\}$,

$$a^{p-1} \bmod p = 1.$$

Observe that this is no definition of primes, but only an implication, that says what has to be fulfilled by each prime. Though Fermat's Little Theorem has been used for the design of randomized primality testing. The idea was to define a kind of witness as follows:

A number $a \in \{1, 2, \ldots, n-1\}$ is a witness of the fact "$n \notin$ PRIM" if and only if

$$a^{n-1} \bmod n \neq 1.$$

Since one can efficiently compute the value $a^{n-1} \bmod n$ by repeated squaring (Section A.2), this kind of witness satisfies the constraints (i) and (ii). Most composite numbers have a lot of witnesses of this kind, and so this concept of witnesses was used even for generating large primes.[5] Unfortunately this

[5]Currently, one calls the randomized primality testing based on the little Fermat's Theorem a pseudotesting of primality.

simple primality testing does not work for all positive integers. There even exist numbers n such that n is not prime, but

$$a^{n-1} \bmod n = 1 \textit{ for all } a \in \{1, \ldots, n-1\}. \tag{6.1}$$

For such composite numbers there is no witness of "$n \notin$ PRIM". The composite numbers satisfying (6.1) are called Carmichael numbers. The smallest Carmichael numbers are $561 = 3 \cdot 11 \cdot 17$, $1105 = 5 \cdot 13 \cdot 17$, and $1729 = 7 \cdot 13 \cdot 19$. Other bad news is that there are infinitely many Carmichael numbers.

Thus, there is no way out other than to continue in our search for a suitable kind of witness. Obviously, it is sufficient to deal with odd integers $n > 2$, because the even integers greater than 2 are composite and divisibility by 2 can be checked easily. To perform the first successful step in the search for good kinds of witnesses, we consider some different possibilities for defining primes. It is well known[6] that

$$n \textit{ is a prime } \Leftrightarrow (\mathbb{Z}_n - \{0\}, \odot_{\bmod p}) \textit{ is a group}^7 \tag{6.2}$$

or equivalently

$$n \textit{ is a prime } \Leftrightarrow (\mathbb{Z}_n, \oplus_{\bmod p}, \odot_{\bmod p}) \textit{ is a field.}$$

Unfortunately, a naive testing of the properties of these algebraic structures, and so of determining whether an algebra is a group or a field, requires the same amount of work as the naive primality test. The usefulness of these characterizations of primes is in their leading us to the following new, useful definition of primes.

Theorem 6.2.1. *Let $p > 2$ be an odd integer. Then*

$$p \textit{ is a prime } \Leftrightarrow a^{\frac{(p-1)}{2}} \bmod p \in \{1, p-1\} \textit{ for all } a \in \mathbb{Z}_p - \{0\}.$$

Proof. We prove the particular implications separately.

(i) Let p be a prime.
Since $p > 2$ and p is odd, one can write p as

$$p = 2 \cdot p' + 1$$

for an $p' = \frac{(p-1)}{2} \geq 1$. Following the little Fermat's Theorem we have

$$a^{p-1} \equiv 1 \pmod{p} \tag{6.3}$$

for all $a \in \mathbb{Z}_p - \{0\}$. Since

$$a^{p-1} = a^{2 \cdot p'} = \left(a^{p'} - 1\right) \cdot \left(a^{p'} + 1\right) + 1,$$

[6]Theorem A.2.27
[7]Theorem A.2.27

(6.3) implies

$$\left(a^{p'} - 1\right) \cdot \left(a^{p'} + 1\right) \equiv 0 \pmod{p}.$$

Since p is a prime, p must divide $\left(a^{p'} - 1\right)$ or $\left(a^{p'} + 1\right)$, and so

$$a^{p'} - 1 \equiv 0 \pmod{p} \qquad \text{or} \qquad a^{p'} + 1 \equiv 0 \pmod{p}. \tag{6.4}$$

Inserting $p' = \frac{(p-1)}{2}$ in (6.4), we obtain

$$a^{\frac{(p-1)}{2}} \equiv 1 \pmod{p} \qquad \text{or} \qquad a^{\frac{(p-1)}{2}} \equiv -1 \equiv p - 1 \pmod{p}.$$

(ii) Let $p > 2$ be an odd integer such that

$$c^{\frac{(p-1)}{2}} \bmod p \in \{1, p-1\} \text{ for all } c \in \mathbb{Z}_p - \{0\}.$$

We prove by contradiction that p is a prime. Let $p = a \cdot b$ be a composite number. From our assumption,

$$a^{\frac{(p-1)}{2}} \bmod p \in \{1, -1\} \text{ and } b^{\frac{(p-1)}{2}} \bmod p \in \{1, -1\}.$$

Since $\odot_{\bmod p}$ is commutative,

$$(a \cdot b)^{\frac{(p-1)}{2}} \bmod p = a^{\frac{(p-1)}{2}} \cdot b^{\frac{(p-1)}{2}} \bmod p \in \{1, -1\}. \tag{6.5}$$

Since $a \cdot b = p$, we have

$$0 = p \bmod p = p^{\frac{(p-1)}{2}} \bmod p = (a \cdot b)^{\frac{(p-1)}{2}},$$

which contradicts (6.5).

□

This new definition of primes leads to the following definition of witnesses.

Let n be an odd integer, $n \geq 3$. A number $a \in \{1, 2, \ldots, n-1\}$ is a witness of the fact "$n \notin$ PRIM", if and only if (6.6)

$$a^{(n-1)/2} \bmod n \notin \{1, n-1\}.$$

Clearly, this kind of witness satisfies the conditions (i) and (ii). The following theorem shows that this definition of witnesses assures the abundance of witnesses for at least every second odd integer greater than 2.

Theorem 6.2.2. *For every positive integer n with an odd $(n-1)/2$ (i.e., for $n \equiv 3 \pmod{4}$),* *(i) if n is a prime, then*

$$a^{(n-1)/2} \bmod n \in \{1, n-1\}$$

for all $a \in \{1, \ldots, n-1\}$, and
(ii) if n is composite, then

$$a^{(n-1)/2} \bmod n \notin \{1, n-1\}$$

for at least half of the elements a from $\{1, 2, \ldots, n-1\}$.

Proof. The assertion (i) has already been proved in Theorem 6.2.1. Hence, it remains to show (ii).

Let n be a composite number such that $n \equiv 3 \pmod 4$. Our task is to show that at least half of the elements from $\{1, 2, \ldots, n-1\}$ are witnesses of the fact "$n \notin$ PRIM".

Let

$$\text{WITNESS} = \{a \in \{1, \ldots, n-1\} \mid a^{(n-1)/2} \bmod n \notin \{1, n-1\}\}$$

be the set of all witnesses of "$n \notin$ PRIM", and let

$$\text{EULER} = \{a \in \{1, \ldots, n-1\} \mid a^{(n-1)/2} \bmod n \in \{1, n-1\}\}$$

be the complementary set of non-witnesses. The elements in EULER are called Eulerian numbers and we prefer this term to the term non-witnesses in what follows. Our proof strategy is to show that there exists an injective (one-to-one) mapping h_b from EULER to WITNESS that directly implies

$$|\text{EULER}| \leq |\text{WITNESS}|.$$

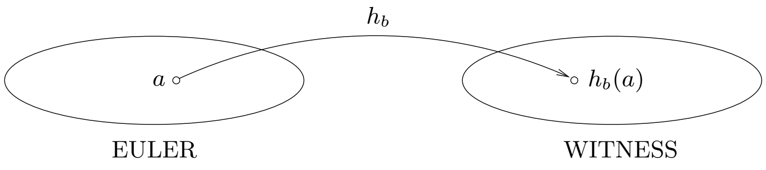


Fig. 6.1.

Assume that b is an element from the set WITNESS, for which there exists a multiplicative inverse b^{-1} in the group $(\mathbb{Z}_n^*, \odot_{\bmod n})$. If such a b exists, one could define h_b by

$$h_b(a) = a \cdot b \bmod n.$$

Next, we will show that h_b is an injective mapping from EULER to WITNESS.

For every $a \in$ EULER, $h_b(a) = a \cdot b$ is not in EULER, and so is in WITNESS, because

$$\begin{aligned}(a \cdot b)^{\frac{(n-1)}{2}} \bmod n &= \left(a^{\frac{(n-1)}{2}} \bmod n\right) \cdot \left(b^{\frac{(n-1)}{2}} \bmod n\right)\\ &= \pm\, b^{\frac{(n-1)}{2}} \bmod n \notin \{1, n-1\}\\ &\quad \{\text{Since } a^{\frac{(n-1)}{2}} \bmod n \in \{1, n-1\} \text{ and}\\ &\quad\ b \in \text{WITNESS}\}\end{aligned}$$

Thus, we have proved that h_b is a mapping from EULER to WITNESS. To show that h_b is injective, one has to prove

$$\text{for all } a_1, a_2 \in \text{EULER},\ a_1 \neq a_2 \text{ implies } h_b(a_1) \neq h_b(a_2).$$

We prove this in an indirect way. Assume $h_b(a_1) = h_b(a_2)$, i.e., assume

$$a_1 \cdot b \equiv a_2 \cdot b \pmod{n}. \tag{6.7}$$

Multiplying the congruence (6.7) from the right by b^{-1}, we obtain

$$a_1 = a_1 \cdot b \cdot b^{-1} \bmod n = a_2 \cdot b \cdot b^{-1} \bmod n = a_2.$$

To complete the proof we have still to show that there exists[8] an element $b \in \text{WITNESS} \cap \mathbb{Z}_n^*$. We do this for

$$n = p \cdot q$$

for two nontrivial factors p and q with GCD $(p, q) = 1$. The case $n = p^d$ for a prime p and a positive integer $d \geq 2$ is left as an exercise to the reader.[9]

Since it is clearer to search for b in $\mathbb{Z}_p \times \mathbb{Z}_q$ instead of searching in $\mathbb{Z}_n$, we apply the Chinese Remainder Theorem. Remember that, for every $a \in \mathbb{Z}_n$, the pair

$$(a \bmod p,\ a \bmod q)$$

is the representation of a in $\mathbb{Z}_p \times \mathbb{Z}_q$. If a is a Eulerian number, we know that

$$a^{(n-1)/2} \bmod p \cdot q \in \{1, n-1\},$$

which implies that

$$\text{either } a^{(n-1)/2} = k \cdot p \cdot q + 1 \text{ or } a^{(n-1)/2} = k \cdot p \cdot q + n - 1 \tag{6.8}$$

for a $k \in \mathbb{N}$. A direct consequence of (6.8) is either

[8]The fact $b \in \mathbb{Z}_n^*$ is equivalent to saying that b has a multiplicative inverse with respect to $\odot_{\bmod p}$. The proof of this observation is given in Theorem A.2.30.

[9]Note that one can efficiently check whether or not n can be expressed as p^d for a prime p, and so one can solve primality testing for such an n without searching for witnesses.

$$a^{\frac{(n-1)}{2}} \bmod p = a^{\frac{(n-1)}{2}} \bmod q = 1$$

or

$$\begin{aligned} a^{\frac{(n-1)}{2}} \bmod p &= (n-1) \bmod p = (p \cdot q - 1) \bmod p = p - 1 \text{ and} \\ a^{\frac{(n-1)}{2}} \bmod q &= (n-1) \bmod q = (p \cdot q - 1) \bmod q = q - 1. \end{aligned}$$

Hence,

$$\text{either } (1,1) \text{ or } (p-1, q-1) = (-1,-1)$$

is the representation of $a^{\frac{(n-1)}{2}} \bmod n$ in $\mathbb{Z}_p \times \mathbb{Z}_q$ for every $a \in$ EULER. Therefore, we choose

$$(1, q-1) = (1, -1)$$

as the representation of b in $\mathbb{Z}_p \times \mathbb{Z}_q$. We show that b has the required properties. The representation of $b^{(n-1)/2} \bmod n$ in $\mathbb{Z}_p \times \mathbb{Z}_q$ is

$$(b^{\frac{(n-1)}{2}} \bmod p, b^{\frac{(n-1)}{2}} \bmod q) = (1^{\frac{(n-1)}{2}} \bmod p, (-1)^{\frac{(n-1)}{2}} \bmod q) = (1, -1),$$

because $(n-1)/2$ is odd.[10] Hence, b is not a Eulerian number, and so $b \in$ WITNESS.

To complete the proof, we show that

$$b^{-1} = b.$$

Since $(1,1)$ is the neutral element with respect to the multiplication in $\mathbb{Z}_p \times \mathbb{Z}_q$,

$$(1, q-1) \odot_{p,q} (1, q-1) = (1 \cdot 1 \bmod p, (q-1) \cdot (q-1) \bmod q) = (1,1)$$

implies that b is inverse to itself. This completes the proof of Theorem 6.2.2. □

Exercise 6.2.3. Find a suitable b in the proof of Theorem 6.2.2 if n is a power of a prime.

Exercise 6.2.4. Let n be a positive integer. Design a deterministic algorithm that efficiently decides whether or not n is a power of a prime.

Exercise 6.2.5. Execute a search for a non-Eulerian number b in the proof of Theorem 6.2.2 without applying the Chinese Remainder Theorem.

The assertion of Theorem 6.2.2 suggests the following algorithm for primality testing.

[10]This is the only place in the proof where we apply this assumption of Theorem 6.2.2.

SSSA (Simplified Solovay-Strassen Algorithm)

```
Input: An odd integer $n$ with $n \equiv 3 \pmod 4$.
Step 1: Choose uniformly an $a \in \{1, 2, \ldots, n-1\}$ at random.
Step 2: Compute $A := a^{(n-1)/2} \bmod n$.
Step 3:
    if $A \in \{1, -1\}$ then
        output "$n \in$ PRIM" {reject}
    else
        output "$n \notin$ PRIM" {accept}
```

Theorem 6.2.6. SSSA *is a polynomial-time* 1MC *algorithm for the recognition of composite numbers n with $n \bmod 4 = 3$.*

Proof. The value of A can be efficiently computed by repeated squaring.[11] The fact that SSSA is a 1MC algorithm is a direct consequence of Theorem 6.2.2. If p is a prime, then (i) of Theorem 6.2.2 assures that there is no witness of $p \notin$ PRIM, and so the algorithm SSSA answers "$n \in$ PRIM" with certainty. If p is composite, then (ii) of Theorem 6.2.2 assures that

$$\text{Prob}\big(\text{SSSA outputs "}n \notin \text{PRIM}''\big) \geq \frac{1}{2}.$$

□

Observe, that SSSA is not a 1MC algorithm for recognizing primes because it can be wrong by indicating prime for a composite number.

Corollary 6.2.7. *For any positive integer n with $n \equiv 3 \pmod 4$,* SSSA *is a polynomial-time bounded-error randomized algorithm for the recognition of primes (for accepting the set* PRIM*).*

6.3 Solovay-Strassen Algorithm for Primality Testing

In Section 6.2 we have a kind of witness, that provides an efficient randomized algorithm for primality testing for all positive integers n with $n \equiv 3 \pmod 4$. This section aims to extend this kind of witness in a way that results in a randomized primality testing for all odd integers.

First, we observe that an $a \in \{1, 2, \ldots, n-1\}$ with $\text{GCD}\,(a, n) \neq 1$ is also a witness of the fact "$n \notin$ PRIM", and that $\text{GCD}\,(a, n)$ can be efficiently computed by the Euclidean algorithm (Section A.2). This could suggest the following extension of the definition (6.6) of witnesses:

A number $a \in \{1, 2, \ldots, n-1\}$ is a witness of the fact "$n \notin$ PRIM" for an odd positive integer n if

(i) $\text{GCD}\,(a, n) > 1$, *or*
(ii) $\text{GCD}\,(a, n) = 1$ *and* $a^{(n-1)/2} \bmod n \notin \{1, n-1\}$. (6.9)

[11] see Section A.2

Exercise 6.3.8. Let n be a composite number, $n \equiv 3 \pmod 4$. Prove that at least half the elements of $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n - \{0\} \mid \mathrm{GCD}(a,n) = 1\}$ are witnesses (with respect to (6.9)) of "$n \notin$ PRIM".
{Hint: A possible way of proving this is by showing that all non-witnesses in $\mathbb{Z}_n^*$ are in a proper subgroup of $(\mathbb{Z}_n^*, \odot_{\bmod n})$.}

Clearly, the kind (6.9) of witnesses is at least as suitable as the kind (6.6) for any n with $n \equiv 3 \pmod 4$. Unfortunately, (6.9) does not guarantee the abundance of witnesses for Carmichael numbers, and so we cannot use this kind of witness for the design of a randomized algorithm for primality testing for all odd, positive integers.

Exercise 6.3.9.* Let n be a composite number, that is not a Carmichael number. Prove that at least half the elements of $\mathbb{Z}_n - \{0\}$ are witnesses (with respect to (6.9)) of the fact "$n \notin$ PRIM".

To improve the witness kind (6.9) we introduce the following two terms.

Definition 6.3.10. Legendre Symbol
For any prime $p > 2$ and any positive integer a with $\mathrm{GCD}(a,p) = 1$ *the* **Legendre symbol for a and p** *is*

$$\mathbf{Leg}\left[\frac{a}{p}\right] = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p,\\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p. \end{cases}$$

The following assertion is a direct consequence of the Euclidean Criterion (Theorem 5.4.14).

Lemma 6.3.11. *For every prime $p > 2$ and every positive integer a with* $\mathrm{GCD}(a,p) = 1$,

$$\mathrm{Leg}\left[\frac{a}{p}\right] = a^{\frac{(p-1)}{2}} \bmod p.$$

Observe that due to Lemma 6.3.11 the Legendre symbols are efficiently computable.

The following definition extends the Legendre symbol from primes to composite numbers.

Definition 6.3.12. Jacobi Symbol
Let

$$n = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_l^{k_l}$$

be the factorization of an odd integer $n \geq 3$, where $p_1 < p_2 < \ldots < p_l$ are primes and $k_1, k_2, \ldots, k_l$ are positive integers for a positive integer l.

For all positive integers a with $\mathrm{GCD}(a,n) = 1$, *the* **Jacobi symbol of a and n** *is*

$$\mathbf{Jac}\left[\frac{a}{n}\right] = \prod_{i=1}^{l} \left(\mathrm{Leg}\left[\frac{a}{p_i}\right]\right)^{k_i} = \prod_{i=1}^{l} \left(a^{\frac{p_i-1}{2}} \bmod p_i\right)^{k_i}.$$

Observation 6.3.13. For all positive integers a and n satisfying the assumptions of Definition 6.3.10,

$$\text{Jac}\left[\frac{a}{n}\right] \in \{1, -1\}.$$

Proof. The Jacobi symbols are products of Legendre symbols and all Legendre symbols are from $\{1, -1\}$. □

Note that working with Jacobi symbols, we really consider -1 a negative integer, and not $n-1$ in $\mathbb{Z}_n$.

Knowing the factorization of n, one can efficiently compute the Jacobi symbol $\text{Jac}\left[\frac{a}{n}\right]$ for any $a \in \mathbb{Z}_n^*$. But for using Jacobi symbols in a definition of witnesses of compositeness, one needs to design an efficient algorithm for computing Jacobi symbols without knowing the factorization[12] of n. One such possibility is provided by the following lemma that formulates the basic rules of working with Jacobi symbols.

Lemma 6.3.14. *Let n be an odd integer greater than 3, and let a, b be natural numbers with* $\text{GCD}(a,n) = \text{GCD}(b,n) = 1$. *Then,*

(i) $\text{Jac}\left[\frac{a \cdot b}{n}\right] = \text{Jac}\left[\frac{a}{n}\right] \cdot \text{Jac}\left[\frac{b}{n}\right]$,
(ii) $\text{Jac}\left[\frac{a}{n}\right] = \text{Jac}\left[\frac{b}{n}\right]$ *for all a, b with* $a \equiv b \pmod{n}$,
(iii) $\text{Jac}\left[\frac{a}{n}\right] = (-1)^{\frac{a-1}{2} \cdot \frac{n-1}{2}} \cdot \text{Jac}\left[\frac{n}{a}\right]$ *for all odd a,*
(iv) $\text{Jac}\left[\frac{1}{n}\right] = 1$, *and* $\text{Jac}\left[\frac{n-1}{n}\right] = (-1)^{\frac{(n-1)}{2}}$,
(v) $\text{Jac}\left[\frac{2}{n}\right] = -1$ *for all n with* $n \bmod 8 \in \{3, 5\}$, *and*
$\text{Jac}\left[\frac{2}{n}\right] = 1$ *for all n with* $n \bmod 8 \in \{1, 7\}$.

Proof. Here, we prove the properties (i) and (ii). The proofs of the other properties are left as exercises.

(i) Let $n = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_l^{k_l}$ for pairwise different, odd primes $p_1, p_2, \ldots, p_k$. Because of the commutativity of $\oplus_{\bmod p_i}$, we have

$$\begin{aligned}
\text{Jac}\left[\frac{a \cdot b}{n}\right] &\underset{def.}{=} \prod_{i=1}^{l} \left((a \cdot b)^{\frac{(p_i-1)}{2}} \bmod p_i\right)^{k_i} \\
&= \prod_{i=1}^{l} \left(\left(a^{\frac{(p_i-1)}{2}} \bmod p_i\right) \cdot \left(b^{\frac{(p_i-1)}{2}} \bmod p_i\right)\right)^{k_i} \\
&= \prod_{i=1}^{l} \left(a^{\frac{(p_i-1)}{2}} \bmod p_i\right)^{k_i} \cdot \prod_{i=1}^{l} \left(b^{\frac{(p_i-1)}{2}} \bmod p_i\right)^{k_i} \\
&\underset{def.}{=} \text{Jac}\left[\frac{a}{n}\right] \cdot \text{Jac}\left[\frac{b}{n}\right].
\end{aligned}$$

This completes the proof of (i).

[12]More precisely, without knowing whether n is a prime or not.

(ii) Following the definition of Jacobi symbols, we see that it is sufficient to show

$$\text{Leg}\left[\frac{a}{p}\right] = \text{Leg}\left[\frac{b}{p}\right]$$

for every prime p and all a, b with $\text{GCD}(a,p) = \text{GCD}(b,p) = 1$ and $a \equiv b \pmod{p}$.
Since $a \equiv b \pmod{p}$, one can express a and b as follows:

$$a = p \cdot r + z \text{ and } b = p \cdot s + z \qquad (6.10)$$

for appropriate $r, s, z \in \mathbb{N}$, $z < p$. Then, one can write

$$\begin{aligned}
\text{Jac}\left[\frac{a}{p}\right] &= a^{\frac{(p-1)}{2}} \bmod p \\
&\underset{(6.10)}{=} (p \cdot r + z)^{\frac{(p-1)}{2}} \bmod p \\
&= \sum_{i=0}^{(p-1)/2} \binom{(p-1)/2}{i} \cdot (p \cdot r)^{\frac{p-1}{2}-i} \cdot z^i \bmod p \\
&= z^{(p-1)/2} \bmod p \\
&\quad \{\text{Because all other members of the sum are divisible by } p \cdot r.\}
\end{aligned}$$

Analogously, applying (6.10) one obtains

$$\text{Jac}\left[\frac{b}{p}\right] = z^{(p-1)/2} \bmod p$$

and so

$$\text{Leg}\left[\frac{a}{p}\right] = \text{Leg}\left[\frac{b}{p}\right].$$

□

Exercise 6.3.15. Prove the properties (iii), (iv), and (v) of Jacobi symbols presented in Lemma 6.3.14.

Lemma 6.3.14 provides us an efficient method for computing of Jacobi symbols without knowing the factorization of n. The following recursive procedure is one of the provided possibilities.

Algorithm JACOBI$[a, n]$

Input: An odd integer $n \geq 3$, and a positive integer a with GCD $(a, n) = 1$.
Procedure: JACOBI$[a, n]$

```
begin
    if $a = 1$ then
        JACOBI$[a, n] := 1$;
    if $a = 2$ and $n$ mod $8 \in \{3, 5\}$ then
        JACOBI$[a, n] := -1$;
    if $a = 2$ and $n$ mod $8 \in \{1, 7\}$ then
        JACOBI$[a, n] := 1$;
    if $a$ is odd then
        JACOBI$[a, n] :=$ JACOBI$[2, n] \cdot$ JACOBI$[a/2, n]$;
    if $a > n$ then
        JACOBI$[a, n] :=$ JACOBI$[a \bmod n, n]$
    else
        JACOBI$[a, n] := (-1)^{\frac{a-1}{2} \cdot \frac{n-1}{2}} \cdot$ JACOBI$[n \bmod a, a]$
end
```

We observe that every recursive call of JACOBI[ ] reduces one of the arguments at least by one half[13], and so $O(\log_2 n)$ recursive calls are sufficient to compute Jac$\left[\frac{a}{n}\right]$. The computation related to a recursive call can be executed by $O(1)$ arithmetical operations or by $O\big((\log_2(a+n))^2\big)$ binary operations. Hence, the algorithm JACOBI$[a, n]$ can be performed by at most $O\big((\log_2(a+n))^3\big)$ binary operations.

Now we are ready to introduce a new definition of witnesses of compositeness. The idea is to say that an a with

$$\text{Jac}\left[\frac{a}{n}\right] \neq a^{\frac{(n-1)}{2}} \bmod n$$

witnesses the fact "$n \notin$ PRIM".

Definition 6.3.16. *Let n be an odd integer, $n \geq 3$. A number $a \in \{1, 2, \ldots, n-1\}$ is called a* **Jac-witness** *of the fact*[14] *"$n \notin$ PRIM" if*

(i) GCD $(a, n) \neq 1$, *or*
(ii) Jac$\left[\frac{a}{n}\right] \neq a^{\frac{(n-1)}{2}} \bmod n$.

The following assertion says that, for every odd, positive, composite integer n, one can guarantee an abundance of Jac-witnesses of n's compositeness.

Theorem 6.3.17. *For every odd integer n, $n \geq 3$, the following holds:*

(a) If n is a prime, then

$$\text{Jac}\left[\frac{a}{n}\right] = \text{Leg}\left[\frac{a}{n}\right] = a^{\frac{(n-1)}{2}} \bmod n$$

for all $a \in \{1, 2, \ldots, n-1\}$.

[13]similarly as with the Euclidean algorithm
[14]of the compositness of n

(b) If n is composite, then

$$\text{Jac}\left[\frac{a}{n}\right] \neq a^{\frac{(n-1)}{2}} \bmod n$$

for at least half the elements $a \in \{1, 2 \ldots, n-1\}$ with the property $\text{GCD}\,(a, n) = 1$ *(i.e., for at least half the elements $a \in \mathbb{Z}_n^*$).*

Proof. The claim (a) is a direct consequence of the definition of Jacobi symbols and the Eulerian Criterion.

It remains to prove the claim (b).

Let n be an odd, composite integer, $n \geq 3$. Our set of witness candidates is $\{1, 2, \ldots, n-1\} = \mathbb{Z}_n - \{0\}$. The Jac-witnesses of "$n \notin$ PRIM" according to Definition 6.3.16(i) are all elements from $\{1, 2, \ldots, n-1\} - \mathbb{Z}_n^*$. If one denotes the set of all non-Jac-witnesses by

$$\overline{\text{Wit}}_n = \{a \in \mathbb{Z}_n^* \mid \text{Jac}\left[\frac{a}{n}\right] = a^{\frac{(n-1)}{2}} \bmod n\},$$

then

$$\mathbb{Z}_n^* - \overline{\text{Wit}}_n$$

is the set of Jac-witnesses of "$n \notin$ PRIM" with respect to Definition 6.3.16(ii). Our aim is to show that

$$|\overline{\text{Wit}}_n| \leq |\mathbb{Z}_n^*|/2, \tag{6.11}$$

and so that

$$|\{1, 2, \ldots, n-1\} - \overline{\text{Wit}}_n| \geq |\overline{\text{Wit}}_n|.$$

In order to prove (6.11), we use an algebraic technique based on Lagrange's Theorem (Theorem A.2.48). We aim to show that

$$(\overline{\text{Wit}}_n, \odot_{\bmod n}) \text{ is a proper subgroup of } (\mathbb{Z}_n^*, \odot_{\bmod n}). \tag{6.12}$$

Lagrange's Theorem says that the cardinality of any subgroup of a group divides the cardinality of the group. Hence, (6.11) is a direct consequence of (6.12).

First, we show that $(\overline{\text{Wit}}_n, \odot_{\bmod n})$ is a group. Following Theorem A.2.40 it is sufficient to show that $\overline{\text{Wit}}_n$ is closed according to $\odot_{\bmod n}$. Let a and b be two elements from $\overline{\text{Wit}}_n$. From Lemma 6.3.14(i) we have

$$\begin{aligned}
\text{Jac}\left[\frac{a \cdot b}{n}\right] &= \text{Jac}\left[\frac{a}{n}\right] \cdot \text{Jac}\left[\frac{b}{n}\right] \\
&= \left(a^{\frac{(n-1)}{2}} \bmod n\right) \cdot \left(b^{\frac{(n-1)}{2}} \bmod n\right) \\
&\quad \{\text{Since } a, b \in \overline{\text{Wit}}_n\} \\
&= (a \cdot b)^{\frac{(n-1)}{2}} \bmod n,
\end{aligned}$$

and so $a \cdot b \in \overline{\text{Wit}}_n$.

The hardest part of the proof is to show that $\overline{\text{Wit}_n}$ is a *proper* subset of$\mathbb{Z}_n^*$. Hence, we are searching for an element $a \in \mathbb{Z}_n^* - \overline{\text{Wit}_n}$. Let

$$n = p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k}$$

be the factorization of n, where i_j are positive integers for $j = 1, \ldots, k$ and $2 < p_1 < p_2 < \ldots < p_k$ are primes. We set

$$q = p_1^{i_1} \text{ and } m = p_2^{i_2} \cdot p_3^{i_3} \cdot \ldots \cdot p_k^{i_k}$$

in order to search for an $a \in \mathbb{Z}_n^* - \overline{\text{Wit}_n}$ in $\mathbb{Z}_q \times \mathbb{Z}_m$ instead of searching directly in $\mathbb{Z}_n$. Let g be the generator of the cyclic group $(\mathbb{Z}_q^*, \odot_{\bmod q})$. We make the choice of a by the following recurrences:

$$a \equiv g \pmod{q}$$

$$a \equiv 1 \pmod{m}.$$

Hence, we choose a as

$$(g, 1) \text{ in } \mathbb{Z}_q \times \mathbb{Z}_m.$$

If $m = 1$, then we simply take $a = g$.

First, we show that $a \in \mathbb{Z}_n^*$, i.e., that GCD $(a, n) = 1$. Therefore, we have to show that

none of the primes $p_1, p_2, \ldots, p_k$ divides the number a. (6.13)

We prove (6.13) in an indirect way. If p_1 divides a, then the equality[15]

$$g = a \bmod p_1^{i_1}$$

contradicts the assumption that g is a generator of $\mathbb{Z}_q^*$.

If, for an $r \in \{2, \ldots, k\}$, p_r divides a, then $a = p_r \cdot b$ for a positive integer b. The congruence $a \equiv 1 \pmod{m}$ implies

$$a = m \cdot x + 1$$

for a natural number x. Hence,

$$a = p_r \cdot b = m \cdot x + 1 = p_r \cdot (m/p_r) \cdot x + 1,$$

which implies[16]

$$p_r \text{ divides } 1.$$

Since $p_r > 1$, the prime p_r cannot divide 1 and so p_r does not divide a.

Thus, we have proved $a \in \mathbb{Z}_n^*$.

Finally, we have to prove that

$$a \notin \overline{\text{Wit}_n}.$$

Now, we distinguish two possibilities, namely $i_1 = 1$ and $i_1 \geq 2$.

[15]This equality follows from the congruence $a \equiv g \pmod{q}$ for $q = p_1^{i_1}$.

[16]Exercise A.2.6 (if p_r divides x and y, and $x = y + z$, then p_r must divide z, too)

(1) Let $i_1 = 1$.
We compute the Jacobi symbol for a and n and the number $a^{\frac{(n-1)}{2}} \bmod n$ in order to show that they are different.
For the following calculation it is important to remember that

$$n = p_1 \cdot m,\ m > 1 \text{ and } \mathrm{GCD}\,(p_1, m) = 1.$$

$$\begin{aligned}
\mathrm{Jac}\left[\frac{a}{n}\right] \underset{def.}{=} & \prod_{j=1}^{k} \left(\mathrm{Jac}\left[\frac{a}{p_i}\right]\right)^{i_j} \\
= & \ \mathrm{Jac}\left[\frac{a}{p_1}\right] \cdot \prod_{j=2}^{k} \left(\mathrm{Jac}\left[\frac{a}{p_j}\right]\right)^{i_j} \\
& \{\text{since } i_1 = 1\} \\
= & \ \mathrm{Jac}\left[\frac{a}{p_1}\right] \cdot \prod_{j=2}^{k} \left(\mathrm{Jac}\left[\frac{1}{p_j}\right]\right)^{i_j} \\
& \{\text{because of } a \equiv 1 \pmod{m} \text{ and Lemma 6.3.14(ii)}\} \\
= & \ \mathrm{Jac}\left[\frac{a}{p_1}\right] \\
& \{\text{Lemma 6.3.14(iv)}\} \\
= & \ \mathrm{Jac}\left[\frac{g}{p_1}\right] \\
& \{\text{becuase of } a \equiv g \pmod{p_1} \text{ and Lemma 6.3.14(ii)}\} \\
= & \ \mathrm{Leg}\left[\frac{g}{p_1}\right] \\
& \{\text{since } p_1 \text{ is a prime}\} \\
= & \ -1 \\
& \{\text{because a generator } g \text{ of } \mathbb{Z}_{p_1}^{*} \text{ cannot be a quadratic residue modulo } p_1.\}
\end{aligned}$$

Hence, we proved that

$$\mathrm{Jac}\left[\frac{a}{n}\right] = -1.$$

Since $a \equiv 1 \pmod{m}$, we obtain

$$\begin{aligned}
a^{\frac{(n-1)}{2}} \bmod m &= (a \bmod m)^{\frac{(n-1)}{2}} \bmod m \\
&= 1^{\frac{(n-1)}{2}} \bmod m \\
&= 1. \qquad (6.14)
\end{aligned}$$

Now, the equality $a^{\frac{(n-1)}{2}} \bmod n = -1$ for $n = q \cdot m$ cannot hold because $a^{(n-1)/2} \bmod n = -1$ implies[17]

[17] If $n = q \cdot m$ and $d \bmod n = -1$ for a $d \in \mathbb{N}$, then $d = l \cdot n - 1 = k \cdot q \cdot m - 1$ for a $k \in \mathbb{N}$. Then, it is obvious that $d \equiv -1 \pmod{m}$.

$$a^{\frac{(n-1)}{2}} \bmod m = -1 \; (= m-1 \text{ in } \mathbb{Z}_m^*),$$

which contradicts (6.14).
Hence, we have proved

$$-1 = \text{Jac}\left[\frac{a}{n}\right] \neq a^{\frac{(n-1)}{2}} \bmod n$$

and so

$$a \in \mathbb{Z}_n^* - \overline{\text{Wit}}_n.$$

(2) Let $i_1 \geq 2$.
We prove $a \notin \overline{\text{Wit}}_n$ in an indirect way.
Assume $a \in \overline{\text{Wit}}_n$. This assumption implies

$$a^{\frac{(n-1)}{2}} \bmod n = \text{Jac}\left[\frac{a}{n}\right] \in \{1, -1\},$$

and so

$$a^{n-1} \bmod n = 1.$$

Since $n = q \cdot m$, we also have

$$a^{n-1} \bmod q = 1.$$

Since $g = a \bmod q$, we obtain

$$1 = a^{n-1} \bmod q = (a \bmod q)^{n-1} \bmod q = g^{n-1} \bmod q. \tag{6.15}$$

Since g is a generator of the cyclic group $(\mathbb{Z}_q^*, \odot_{\bmod q})$, the order of g is $|\mathbb{Z}_q^*|$. From (6.15) we have that the order of g must divide $n-1$, i.e.,

$$|\mathbb{Z}_q^*| \text{ divides } n-1. \tag{6.16}$$

Since $q = p_1^{i_1}$ for an $i_1 \geq 2$, and

$$\mathbb{Z}_q^* = \{x \in \mathbb{Z}_q \mid \text{GCD}\,(x, q) = 1\} = \{x \in \mathbb{Z}_q \mid p_1 \text{ does not divide } x\}$$

and the number of elements of $\mathbb{Z}_q$ that are a multiple of p_1 is exactly $|\mathbb{Z}_q|/p_1$, one obtains

$$|\mathbb{Z}_q^*| = |\mathbb{Z}_q| - |\mathbb{Z}_q|/p_1 = p_1^{i_1} - p_1^{i_1-1} = p_1 \cdot \left(p_1^{i_1-1} - p_1^{i_1-2}\right).$$

Hence,

$$p_1 \text{ divides } |\mathbb{Z}_q^*|. \tag{6.17}$$

But (6.16) and (6.17) together imply that

$$p_1 \text{ divides } n-1. \tag{6.18}$$

Since $n = p_1^{i_1} \cdot m$, we have obtained

$$p_1 \text{ divides } n \text{ and } p_1 \text{ divides } n-1.$$

Since no prime can divide both[18] $n-1$ and n, our assumption $a \in \overline{\mathrm{Wit}}_n$ cannot hold, and we obtain

$$a \notin \overline{\mathrm{Wit}}_n.$$

□

Theorem 6.3.17 shows that the Jac-witnesses are a suitable kind of witnesses of randomized primality testing. The following algorithm is a straightforward application of the method of abundance of witnesses for the Jac-witnesses.

Algorithm Solovay-Strassen

```
Input: An odd integer $n$, $n \geq 3$.
Step 1: Choose uniformly an $a$ from $\{1, 2, \ldots, n-1\}$ at random.
Step 2: Compute GCD $(a, n)$.
Step 3:
    if GCD $(a, n) \neq 1$ then
        output ("$n \notin$ PRIM") {accept}
Step 4: Compute Jac$\left[\frac{a}{n}\right]$ and $a^{\frac{(n-1)}{2}}$ mod $n$
Step 5:
    if Jac$\left[\frac{a}{n}\right] = a^{\frac{(n-1)}{2}}$ mod $n$ then
        output ("$n \in$ PRIM") {reject}
    else
        output ("$n \notin$ PRIM") {accept}.
```

Theorem 6.3.18. *The Solovay-Strassen algorithm is a polynomial-time one-sided-error Monte Carlo algorithm for the recognition of composite numbers.*

Proof. First, we analyze the time complexity of the algorithm. Since $a < n$, one can measure the complexity of all operations over a and n with respect to the representation length $\lceil \log_2(n+1) \rceil$ of the input n. Computing GCD (a, n) in step 2 costs at most $O\big((\log_2 n)^3\big)$ binary operations. The value $a^{(n-1)/2}$ mod n can be computed in $O\big((\log_2 n)^3\big)$ binary operations,[19] too. The algorithm JACOBI computes Jac$\left[\frac{a}{n}\right]$ in $O\big((\log_2 n)^3\big)$ binary operations. The cost of the comparison in step 5 is linear in $\log_2 n$, and so the overall complexity of the algorithm is in $O\big((\log_2 n)^3\big)$.

Next, we analyze the success probability of the Solovay-Strassen algorithm.

[18]If a prime p divides both $n-1$ and n, then the equality $n = (n-1) + 1$ would imply that p divides 1.

[19]If one applies the best known algorithm for multiplication of large integers, the the value $a^{(n-1)/2}$ mod n can even be computed in time $O\big((\log_2 n)^2 \cdot \log\log\log n\big)$.

If n is a prime, there does not exist any witness of "$n \notin$ PRIM" (Theorem 6.3.17(a)), and so the algorithm outputs the answer "$n \in$ PRIM" ("reject") with certainty.

If n is composite, Theorem 6.3.17(b) assures that at least half the elements of $\{1, 2, \ldots, n-1\}$ are Jac-witnesses of "$n \notin$ PRIM". Therefore, the Solovay-Strassen algorithm gives the right answer "$n \notin$ PRIM" with probability at least 1/2. $\square$

Since the Solovay-Strassen algorithm is a 1MC algorithm, a constant[20] number of repetitions of its work on the same input is sufficient in order to reduce the error probability[21] to an arbitrarily small chosen $\epsilon > 0$. Hence, the algorithm is practical and used in many applications.

6.4 Generation of Random Primes

One of the most frequent tasks in the modern cryptography is the following:

For a given positive integer l, generate a random prime of the binary length l.

Typically, the given binary length l is of the order of hundreds. Hence, the number of primes of the length l is larger that the number of protons in the known universe. Clearly, one cannot solve this task by generating all primes of length l and than choosing one of them at random. The strategy used simply generates a random integer of length l and then applies a randomized primality test in order to check whether or not the generated number is a prime. This approach works due to the Primality Theorem (Theorem A.2.9), that assures an abundance[22] of primes among natural numbers. If one uniformly generates an integer of length l at random and repeats this procedure until a prime is generated, then the prime generated can be considered random with respect to the uniform probability distribution over all primes of length l.

In what follows, we present an algorithm for the generation of random primes. This algorithm has two inputs, l and k. The number l is the length of primes we are looking for and the number k is the number of independent runs of the Solovay-Strassen algorithm performed on every generated number in order to check whether or not it is a prime.

PRIMEGEN (l, k)

Input: Positive integers l and k, $l \geq 3$.

[20]with respect to the input size

[21]Recall, that the error probability of 1MC algorithms tends to 0 with an exponential speed in the number of runs executed.

[22]For a randomly chosen number n, the probability that n is a prime is approximately $1/\ln n$.

```
Step 1:
    $X$ := "still not found";
    $I := 0$;
Step 2:
    while $X$ := "still not found" and $I < 2 \cdot l^2$ do
    begin
        generate a bit sequence $a_1, a_2, \ldots, a_{l-2}$ at random
        and compute
            $$n := 2^{l-1} + \sum_{i=1}^{l-2} a_i \cdot 2^i + 1$$
        {Hence, $n$ is a random integer of length $l$}
        Perform $k$ independent runs of the Solovay-Strassen algorithm on
        $n$;
        if at least one output is "$n \notin$ PRIM" then
            $I := I + 1$
        else
            begin
                $X$ := already found;
                output "$n$"
            end;
    end;
Step 3:
    if $I = 2 \cdot l^2$ then
        output "I was unable to find a prime."
```

Theorem 6.4.19. *The algorithm* PRIMEGEN (l, l) *is a bounded-error algorithm for generating primes that works in time polynomial in l.*

Proof. First, we analyze the time complexity of the randomized algorithm PRIMEGEN (l, k) for $l = k$. Though the input length is $2 \cdot \lceil \log_2(l+1) \rceil$, we measure the complexity in l, because the output length is l.

The algorithm performs at most $2 \cdot l^2$ attempts to generate a number and to test it. The random generation itself runs in linear time in l, and one run of the Solovay-Strassen algorithm can require $O(l^3)$ binary operations. Hence, the worst-case running time of PRIMEGEN (l, l) is in $O(l^5)$.

We begin the analysis of the success probability of PRIMEGEN (l, l) by estimating the probability that PRIMEGEN (l, l) answers "I was unable to find a prime". This unwanted event can occur only if none of the $2 \cdot l^2$ randomly generated numbers is a prime[23], and for every one of these generated numbers, the Solovay-Strassen primality test proves in l runs that the given number is

[23]Recall, that the Solovay-Strassen algorithm must output "$n \in$ PRIM" if n is a prime.

composite. Since the probability, that a random number of length l is a prime, is at least[24]

$$\frac{1}{\ln n} > \frac{1}{2 \cdot l},$$

the probability of generating no prime in one attempt is at most

$$1 - \frac{1}{2 \cdot l}. \tag{6.19}$$

Let

$$w_l \geq 1 - \frac{1}{2^l}$$

be the probability, that l runs of the Solovay-Strassen primality test succeed in proving "$n \notin$ PRIM" for a given, composite n of length l. Hence, we obtain

$$\begin{aligned}
\text{Prob}(\text{PRIMEGEN}\,(l,l) = \text{“I was unable to find a prime”}) \\
&\underset{(6.19)}{<} \left(\left(1 - \frac{1}{2 \cdot l}\right) \cdot w_l\right)^{2 \cdot l^2} \\
&< \left(1 - \frac{1}{2 \cdot l}\right)^{2 \cdot l^2} \\
&= \left[\left(1 - \frac{1}{2 \cdot l}\right)^{2 \cdot l}\right]^l \\
&< \left(\frac{1}{e}\right)^l = e^{-l}.
\end{aligned}$$

Clearly, e^{-l} tends to 0 with growing l, and $e^{-l} < \frac{1}{4}$ for all $l \geq 2$. For $l \geq 100$, e^{-l} is substantially smaller than 10^{-40}.

Next, we analyze the probability of the second unwanted event, that PRIMEGEN (l,l) outputs a composite number n as a prime.[25] The algorithm PRIMEGEN (l,l) produces a composite number n only if

(i) all numbers generated before[26] n were composite, and this fact was proved in PRIMEGEN (l,l) for each of these numbers by at most l runs of the Solovay-Strassen algorithm, and
(ii) n is composite, but PRIMEGEN (l,l) does not succeed in proving n's compositeness in l runs of the Solovay-Strassen algorithm.

Since n can be the i-th randomly generated number for $i = 1, 2, \ldots, 2 \cdot l^2$, denote by p_i the probability that the wrong answer n is the i-th generated number. The lower bound (6.19) implies

[24]due to the Prime Number Theorem (Theorem A.2.9)
[25]i.e., the probability of a wrong output (error probability).
[26]If n is the i-th generated number, then we mean the first $i-1$ numbers generated.

$$p_1 \leq \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l},$$

since 2^{-l} is a lower bound on the probability that "$n \notin$ PRIM" was not proved in l runs of the Solovay-Strassen primality test.

For all $i = 2, 3, \ldots, 2 \cdot l^2$,

$$p_i \leq \left[\left(1 - \frac{1}{2 \cdot l}\right) \cdot w_l\right]^{i-1} \cdot \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l},$$

where $\left[\left(1 - \frac{1}{2 \cdot l}\right) \cdot w_l\right]^{i-1}$ is an upper bound on the probability that the first $i-1$ generated numbers are composite and that this fact was successfully recognized. Thus, we obtain

$$\begin{aligned}
\mathrm{Error}_{\mathrm{PRIMEGEN}(l,l)}(l) &\leq p_1 + \sum_{j=2}^{2 \cdot l^2} p_j \\
&\leq \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l} + \\
&\quad + \sum_{i=1}^{2 \cdot l^2 - 1} \left[\left(1 - \frac{1}{2 \cdot l}\right) \cdot w_l\right]^i \cdot \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l} \\
&\leq \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l} \cdot \left(\sum_{i=1}^{2 \cdot l^2 - 1} \left(1 - \frac{1}{2 \cdot l}\right)^i + 1\right) \\
&\leq \left(1 - \frac{1}{2 \cdot l}\right) \cdot \frac{1}{2^l} \cdot 2 \cdot l^2 \\
&\leq \frac{l^2}{2^{l-1}}.
\end{aligned}$$

Clearly, the value $l^2 \cdot 2^{-(l-1)}$ tends to 0 with growing l, and $\mathrm{Error}_{\mathrm{PRIMEGEN}(5,5)}(5) \leq 1/5$. For $l \geq 100$,

$$\mathrm{Error}_{\mathrm{PRIMEGEN}(l,l)}(l) \leq l^2 \cdot 2^{-(l-1)} \leq 1.58 \cdot 10^{-26}.$$

□

In order to increase the success probability of PRIMEGEN (l, k), we have probably chosen a too large k, that essentially increases the time complexity. The following exercises provide to the reader an opportunity of thinking about an appropriate trade-off between the amount of work and the success probability.

Exercise 6.4.20. Analyze the time complexity and the success probability of PRIMEGEN (l, k) for the following values of k:

(i) $k = 2 \cdot \lceil \log_2 l \rceil$,
(ii) $k = 2 \cdot (\lceil \log_2 l \rceil)^2$, and
(iii) $k = \lceil \sqrt{l} \rceil$.

Exercise 6.4.21. The number $2 \cdot l^2$ of attempts to generate a random prime in PRIMEGEN (l, k) may be too large. Which choice of the number of attempts would you prefer in order to reduce the worst-case complexity of PRIMEGEN (l, k) on the one hand, but still assure a high success probability on the other hand?

Exercise 6.4.22. Modify PRIMEGEN (l, k) in such a way that it must run until it outputs a number n. This means that one forbids the output "I was unable to find a prime," and so there exist infinite runs of PRIMEGEN (l, k). Analyze the expected running time and the error probability of such a modification of PRIMEGEN (l, k).

6.5 Summary

Abundance of witnesses as a method for the design of efficient randomized algorithms uncovers the deepest roots of the nature of the power of randomization. If for a hard computing problem there are only sets of candidates for a witness in which the witnesses are randomly distributed, then randomized algorithms can solve the problem efficiently despite the fact that the problem cannot be efficiently solved by any deterministic algorithm. The art of applying the method of abundance of witnesses lies in the search for a suitable kind of witness. Simple kinds of witnesses were considered in Chapter 4, where we applied fingerprinting as a special case of the method of abundance of witnesses for several tasks. In this chapter we presented a part of the story of searching for kinds of witnesses suitable for randomized primality testing. We call attention to the fact that one can test primality in deterministic polynomial time, recently shown by Agrawal, Kayal and Saxena [AKS02], and that the design of this deterministic algorithm can also be viewed as a search for a kind of witness. But here one does not look for an abundance of witnesses. One looks for a set of candidates in which one can determine a "small" subset[27] that contains at least one witness with certainty. Since the designed deterministic primality test runs in $O\big((\log_2 n)^{10}\big)$, it is not considered a real competitor for the known efficient randomized algorithms in current applications.

Primality testing is one of the fundamental problems of mathematics and computer science. For 2000 years, whether or not one could test primality of n faster than trying to divide n by all numbers smaller or equal to $\sqrt{n}$ was an open question. In the 17th century, Fermat, with his Fermat's Little Theorem, first brought to attention the possibility of an efficient primality test. This primality test does not work for all numbers, especially not for Carmichael

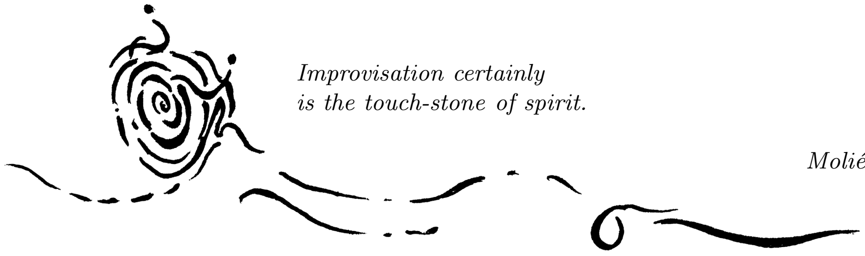
[27]The cardinality of the subset should be polynomial in input size.

numbers. The Carmichael numbers were discovered by Carmichael [Car12], and the proof that there exist infinitely many Carmichael numbers was given by Alford, Granville, and Pomerance [AGP92]. At first, the development of algorithmics and complexity theory brought an essential progress in attacking this fundamental problem. Pratt [Pra75] has proved that primality testing is in NP. The Solovay-Strassen algorithm presented here, designed by Solovay and Strassen [SS77], is one of the first efficient randomized and practicable algorithms for primality testing. A transparent explanation of the development of ideas leading to its design is given by Strassen [Str96]. In 1976, Miller showed that the validity of the Extended Riemann Hypothesis implies the existence of an efficient deterministic algorithm for primality testing. Rabin [Rab76, Rab80] modified[28] Miller in order to design an efficient randomized algorithm. Both the above mentioned algorithms are one-sided-error Monte Carlo algorithms. To improve on them, Adleman and Huang [AH87] designed a polynomial-time Las Vegas algorithm for primality testing. In 1983 Adleman, Pomerance, and Rumely even discovered a deterministic primality test that runs in super-polynomial time $(\log_2 n)^{O(\log \log \log n)}$. The biggest break was achieved in 2002 when Agrawal, Kayal, and Saxena [AKS02] discovered a deterministic primality test running in time $O\big((\log_2 n)^{12}\big)$. The design of this algorithm is considered to be one of the most important achievements of algorithmics.

For a more involved study of primality testing we recommend the textbooks of Motwani and Raghavan [MR95], Cormen, Leiserson, Rivest, and Stein [CLRS01], and Hromkovič [Hro03]. The fascinating story of the development of algorithms for primality testing is presented in a correspondingly fascinating way by Dietzfelbinger [Die04].

[28]Rabin exchanged the assumption of the validity of the Extended Riemann Hypothesis in Miller's algorithm for randomization.

7 Optimization and Random Rounding



Improvisation certainly
is the touch-stone of spirit.

Moliére

7.1 Objectives

Integer linear programming (ILP) and 0/1-linear programming (0/1-LP) are known NP-hard optimization problems. On the other hand, the linear programming problem (LP) is solvable in polynomial time. Interestingly, all three problems, ILP, 0/1-LP, and LP, are specified by the same kind of linear constraints and have the same kind of linear objectives. The only difference is in requiring integer solutions for ILP and Boolean solutions for 0/1-LP, while the basic problem of linear programming is considered over real[1] numbers.

How is it possible that there are such large differences in the computational complexities of these problems, while they look so similar? The reason is that the constraints requiring integer or Boolean solutions cannot be formulated as linear equations, and so ILP and 0/1-LP leave the area of linear constraints in this sense. The linearity of LP is really substantial, because the set of feasible solutions of a system of linear inequalities (of an instance of LP) builds a polytype.[2] To search for an optimum with respect to a linear function in a polytype is not so hard, because one can reach an optimum by applying a local search starting from an arbitrary vertex of the polytype and moving along the edges of the polytype.[3]

This strong similarity between the efficiently solvable LP and the NP-hard ILP and 0/1-LP is used in order to design efficient approximation algorithms. First, one expresses an instance of a discrete optimization problem as an instance of ILP or 0/1-LP. Usually, this can be done very easily, because the constraints of many hard optimization problems are naturally expressible in

[1]i.e., without any restriction on the type of input values

[2]a convex, multidimensional object

[3]This is exactly what the famous Simplex algorithm does. The Simplex algorithm runs very fast on almost all instances of LP. But there exist artificial LP instances requiring exponentially many local improvements during the execution of the local search to reach an optimum.

the form of linear equations (or inequalities).[4] The second step is called the relaxation to linear programming. Here, one does not take the constraints requiring integer or Boolean solutions into account, and efficiently solves the given problem instance as an instance of LP. The third and final step is devoted to the conversion of the computed optimal solution[5] for LP to a reasonably good, feasible solution to the original problem instance. One of the possible conversion strategies is random rounding, which we aim to present in this chapter.

This chapter is organized as follows. In Section 7.2 we introduce the method of relaxation to linear programming and show how one can get approximation algorithms by applying this method.

In Section 7.3 we combine the method of relaxation to LP with random rounding in order to design a randomized approximation algorithm for MAX-SAT. We shall see that this algorithm in incomparable with the random sampling algorithm for MAX-SAT presented in Section 2.5 (Exercise 2.5.72), where the incomparability is considered with respect to the quality (approximation ratio) of computed solutions. In Section 7.4 we merge these two randomized approximation algorithms for MAX-SAT and obtain a randomized algorithm whose expected approximation ratio is at most 4/3 for any given formula in CNF.

Finally, Section 7.5 provides a short summary of the most important ideas of the chapter and a survey of related results and recommended sources for more involved study of this topic.

7.2 Relaxation to Linear Programming

The relaxation to linear programming is one of the most frequently applied methods for designing algorithms for NP-hard, discrete optimization problems. The basic schema of this method can be described as follows.

Schema of the Relaxation to Linear Programming

Input: An instance I of an optimization problem U

(1) Reduction:
Express I as an instance ILP(I) of ILP (or 0/1-LP).

(2) Relaxation:
Consider ILP(I) as an instance Rel-LP(I) of LP (i.e., do not take the constraints requiring integer or Boolean solutions into account), and compute an optimal solution α for Rel-LP(I) by a method of linear programming.

[4]Due to this linear programming problems became the paradigmatic problems of combinatorial optimization and operations research.

[5]Which is not necessarily a feasible (integer or Boolean) solution to the original problem instance.

{It is important to observe, that the cost of α is a bound on the achievable optimal cost of the original problem instance I, i.e., that

$$\mathrm{cost}(\alpha) = \mathrm{Opt}_{\mathrm{LP}}(\text{Rel-LP}(I)) \leq \mathrm{Opt}_U(I)$$

if U is a minimization problem, and

$$\mathrm{cost}(\alpha) = \mathrm{Opt}_{\mathrm{LP}}(\text{Rel-LP}(I)) \geq \mathrm{Opt}_U(I)$$

if U is a maximization problem).

The reason for this is that the constraints of Rel-LP (I) are a proper subset of the constraints of ILP(I) (and so of I). Therefore the set $\mathcal{M}(\mathrm{ILP}(I))$ of feasible solutions for I is a subset of the set $\mathcal{M}(\text{Rel-LP}(I))$ of feasible solutions for Rel-LP (I).}

(3) Solving the original problem:

Use α in order to compute a feasible solution β for I that is of a sufficiently high quality (i.e., optimal or a good approximation of an optimal solution). {Though one is often unable to efficiently compute an optimal solution for I, and so to estimate the cost $\mathrm{Opt}_U(I)$, the approximation ration of β can be upper bounded by comparing $\mathrm{cost}(\alpha) = \mathrm{Opt}_{\mathrm{LP}}(\text{Rel-LP}(I))$ with $\mathrm{cost}(\beta)$.}

Parts (1) and (2) of the schema are executable in polynomial time. Therefore, part (3) corresponds to an NP-hard problem if one forces to compute an optimal solution.[6] If one aims to design an approximation algorithm only, one can search for a strategy by executing step (3), which runs in polynomial time and guarantees a reasonable approximation ratio for every problem instance.

To illustrate the method of relaxation to linear programming, we give a few examples of reductions to ILP and then present the design of an approximation algorithm.

The problem of linear programming is one of the fundamental optimization problems in mathematics. Its importance lies especially in the fact that many real situations and frameworks can be well modeled by LP and in the fact that many different optimization problems can be expressed in terms of LP.

A general version of LP that accepts equations as well as inequalities is as follows:

For every problem instance $A = [a_{ji}]_{j=1,\ldots,m,\ i=1,\ldots,n}$, $b = (b_1, \ldots, b_m)^T$, $c = (c_1, \ldots, c_n)$, $M_1, M_2 \subseteq \{1, \ldots, m\}$, $M_1 \cap M_2 = \emptyset$, $n, m \in \mathbb{N} - \{0\}$,

$$\text{minimize the linear function } f_c(x_1, \ldots, x_n) = \sum_{i=1}^{n} c_i \cdot x_i$$

[6]This does not exclude the possibility of efficiently computing optimal solutions for several specific instances.

under the linear constraints[7]

$$\sum_{i=1}^{n} a_{ji} \cdot x_i = b_j \text{ for } j \in M_1,$$
$$\sum_{i=1}^{n} a_{si} \cdot x_i \geq b_s \text{ for } s \in M_2, \text{ and}$$
$$\sum_{i=1}^{n} a_{ri} \cdot x_i \leq b_r \text{ for } r \in \{1, 2, \ldots, m\} - (M_1 \cup M_2).$$

If $x = (x_1, x_2, \ldots, x_n)^T$ is considered over real numbers, then one can solve this problem in polynomial time.[8] If one adds the additional nonlinear constraints $x_i \in \{0, 1\}$ or $x_i \in \mathbb{Z}$, one obtains the NP-hard problems 0/1-LP and ILP.

In what follows we present a few examples of the reduction and the relaxation of a few discrete optimization problems.

Example 7.2.1. **The minimum vertex cover problem (MIN-VC)**

Remember that an instance of MIN-VC is a graph $G = (V, E)$. A feasible solution is any vertex set $U \subseteq V$ such that each edge from E has at least one end point in U. The objective is to minimize the cardinality of U.

Let $V = \{v_1, v_2, \ldots, v_n\}$. One can represent a feasible solution U by a Boolean vector $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$, where

$$x_i = 1 \Leftrightarrow v_i \in U.$$

This representation of feasible solutions enables us to express an instance $G = (V, E)$ of MIN-VC as the following instance ILP(G) of 0/1-LP:

$$\text{Minimize } \sum_{i=1}^{n} x_i \tag{7.1}$$

under the $|E|$ linear constraints

$$x_i + x_j \geq 1 \text{ for every edge } \{v_i, v_j\} \in E \tag{7.2}$$

and the n nonlinear constraints

$$x_i \in \{0, 1\} \text{ for } i = 1, 2, \ldots, n. \tag{7.3}$$

[7]We know that each LP instance can be reduced to normal forms that allow either only equations or only inequalities. This can be done efficiently by introducing new variables, but it is a topic of operations research, and we omit the presentation of such details here.

[8]It was an open question for a long time whether or not LP is solvable in polynomial time (see Section 7.5).

If one relaxes (7.3) to the $2 \cdot n$ linear constraints

$$x_i \geq 0 \text{ and } x_i \leq 1 \text{ for } i = 1, 2, \ldots, n, \tag{7.4}$$

one obtains the instance Rel-LP (G) of LP. □

Exercise 7.2.2. Consider the weighted MIN-VC, where every vertex has been assigned a positive integer weight, and the task is to minimize the overall weight of the vertex cover. Express this optimization problem as 0/1-LP.

Example 7.2.3. **The knapsack problem (MAX-KP)**

An instance I of MAX-KP is given by a sequence of $2 \cdot n + 1$ positive integers $I = w_1, w_2, \ldots, w_n, c_1, c_2, \ldots, c_n, b$ for a positive integer n. The idea is to consider n objects, where for $i = 1, 2, \ldots, n$, w_i is the weight of the i-th object and c_i is the cost of the i-th object. One has a knapsack whose weight capacity is bounded by b and the aim is to pack some objects in the knapsack in such a way that the weight of the knapsack content is not above b and the overall cost of objects in the knapsack is maximized. Again, one can describe any feasible solution by a vector $(x_1, x_2, \ldots, x_n) \in \{0, 1\}^n$, where

$$x_i = 1 \Leftrightarrow \text{ the } i\text{-th object is in the knapsack.}$$

Then, an instance I of MAX-KP can be expressed as the following instance ILP (I) of 0/1-LP:

$$\text{Maximize } \sum_{i=1}^{n} c_i \cdot x_i$$

under the linear constraint

$$\sum_{i=1}^{n} w_i \cdot x_i \leq b,$$

and the n nonlinear constraints

$$x_i \in \{0, 1\} \text{ for } i = 1, \ldots, n.$$

If one exchanges the constraints $x_i \in \{0, 1\}$ by the following $2 \cdot n$ linear constraints

$$x_i \leq 1 \text{ and } x_i \geq 0 \text{ for } i = 1, 2, \ldots, n,$$

then one obtains the corresponding relaxed instance Rel-LP (I) of LP. □

Exercise 7.2.4. The maximum matching problem is to find a matching of maximum cardinality in a given graph G. Express any instance of this problem as an instance of 0/1-LP.

Example 7.2.5. **The set cover problem (MIN-SC)**

An instance of MIN-SC is a pair $(X, \mathcal{F})$, where $X = \{a_1, \ldots, a_n\}$ and $\mathcal{F} = \{S_1, S_2, \ldots, S_m\}$, $S_i \subseteq X$ for $i = 1, \ldots, m$. A feasible solution is any set $\mathcal{S} \subseteq \mathcal{F}$, such that $X = \bigcup_{S \in \mathcal{S}} S$. This task is to minimize the cardinality of $\mathcal{S}$. Similarly as in Exercise 7.2.1 and Exercise 7.2.3 one can represent a feasible solution $\mathcal{S}$ by a Boolean vector $(x_1, x_2, \ldots, x_m) \in \{0,1\}^m$, such that

$$x_i = 1 \Leftrightarrow S_i \in \mathcal{S}.$$

We introduce the notation

$$\text{Index}(k) = \{d \in \{1, \ldots, m\} \mid a_k \in S_d\}$$

for $k = 1, 2, \ldots, n$. Then, one can express $(X, \mathcal{F})$ as the following instance of 0/1-LP:

$$\text{Minimize } \sum_{i=1}^{m} x_i$$

under the n linear constraints

$$\sum_{j \in \text{Index}(k)} x_j \geq 1 \text{ for } k = 1, 2, \ldots, n$$

and under the m nonlinear constraints

$$x_i \in \{0, 1\} \text{ for } i = 1, 2, \ldots, m.$$

□

Exercise 7.2.6. What do MIN-VC and MIN-SC have in common? Can one consider one of these problems as a special case of the other?

We have seen that some optimization problems can be expressed as instances of ILP in a very natural way. Hence, part (1) of the schema of the relaxation to LP is usually the simplest one. As already observed, part (2) can be efficiently performed.[9] The development of efficient algorithms for LP is a part of operations research. Since the details of their design are not directly related to our considerations and aims, we omit the details of the execution of part (2). From our point of view, the most creative part of the design of approximation algorithms by the method of the relaxation to LP is part (3), for which one does not have any universal, or at least robust, approach working for a large class of optimization problems. We finish this section by presenting an example that shows that sometimes even simple rounding can help.

Example 7.2.7. Consider the MIN-VC problem that we already expressed as 0/1-LP in Exercise 7.2.1.

[9]Though the corresponding algorithms and their analyses are highly nontrivial.

Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in [0,1]^n$ be an optimal solution for the relaxed instance Rel-LP (G), which is determined by (7.1), (7.2), and (7.4). Clearly, an $\alpha_i \notin \{0,1\}$ (from $[0,1]$) does not have any interpretation for MIN-VC. Hence, we have to create a $\beta = (\beta_1, \beta_2, \ldots, \beta_n) \in \{0,1\}^n$ from α. Let us do this by simply rounding the α_i's in the following way:

$$\beta_i = 1 \Leftrightarrow \alpha_i \geq \frac{1}{2}.$$

Next, we show that this algorithm following the schema of the relaxation to LP is a 2-approximation algorithm.

From the constraints (7.2) we see that the optimal solution α for Rel-LP (G) must satisfy the inequality

$$\alpha_i + \alpha_j \geq 1$$

for every edge $\{v_i, v_j\} \in E$. Hence,

$$\alpha_i \geq \frac{1}{2} \text{ or } \alpha_j \geq \frac{1}{2},$$

and so rounding α_i and α_j one obtains

$$\beta_i = 1 \text{ or } \beta_j = 1.$$

Therefore, at least one of the vertices v_i and v_j is in the resulting[10] vertex cover, and so the edge $\{v_i, v_j\}$ is covered. Thus, we have shown that β is a feasible solution for G.

Since one rounds to the closest value from $\{0,1\}$,

$$\beta_i \leq 2 \cdot \alpha_i,$$

and so

$$\text{cost}(\beta) = \sum_{i=1}^{n} \beta_i \leq 2 \cdot \sum_{i=1}^{n} \alpha_i = 2 \cdot \text{cost}(\alpha). \tag{7.5}$$

Since the set of feasible solutions for the instance ILP(G) of 0/1-LP (and so for G as an instance of MIN-VC) is a subset of the set of feasible solutions for Rel-LP (G) as an instance of LP,

$$\text{cost}(\alpha) = \text{Opt}_{\text{LP}}(\text{Rel-LP}(G)) \leq \text{Opt}_{\text{MIN-VC}}(G). \tag{7.6}$$

In this way one finally obtains

$$\text{Ratio}(G) \underset{def.}{=} \frac{\text{cost}(\beta)}{\text{Opt}_{\text{MIN-VC}}(G)} \underset{\substack{(7.5)\\(7.6)}}{\leq} \frac{2 \cdot \text{cost}(\alpha)}{\text{cost}(\alpha)} = 2.$$

□

[10] described by β.

Exercise 7.2.8. Let k be a positive integer. Consider MIN-SC(k) as the following restricted version of MIN-SC. The instances of MIN-SC(k) are usual instances $(X, \mathcal{F})$ of MIN-SC with the additional restriction that each element $x \in X$ is contained in at most k sets from $\mathcal{F}$. Apply the method of the relaxation to LP in order to design a k-approximation algorithm for MIN-SC(k). {Hint: Observe that MIN-SC(2) and MIN-VC are the same optimization problems.}

7.3 Random Rounding and MAX-SAT

The aim of this section is to combine the method of the relaxation to linear programming with random rounding in order to design a randomized approximation algorithm for MAX-SAT. At least for formulas with short clauses, the expected number of satisfied clauses should be larger than the number of clauses satisfied by the solutions produced by the naive RSAM algorithm from Exercise 2.3.35 that simply generates a random[11] assignment.

Before presenting the new algorithm, we show how to express an instance of MAX-SAT as an instance of 0/1-LP. Let

$$\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$$

be a formula over the set $\{x_1, x_2, \ldots, x_n\}$ of Boolean variables, where F_i is a clause for $i = 1, 2, \ldots, m$, $n, m \in \mathbb{N} - \{0\}$. Let $\mathrm{Set}(F_i)$ be the set of all literals[12] in F_i. We denote by $\mathrm{Set}^+(F_i)$ the set of all variables that occur in F_i in the positive setting (without negation), and by $\mathrm{Set}^-(F_i)$ the set[13] of all variables whose negations occur in F_i. We assume $\mathrm{Set}^+(F_i) \cap \mathrm{Set}^-(F_i) = \emptyset$, because in the opposite case F_i is always satisfied, and we do not need to consider such clauses in the instances of MAX-SAT.

We denote by $\mathrm{In}^+(F_i)$ and $\mathrm{In}^-(F_i)$ the set[14] of indices of the variables in $\mathrm{Set}^+(F_i)$ and $\mathrm{Set}^-(F_i)$, respectively. Using this notation one can express the instance Φ of MAX-SAT as the following instance $\mathrm{LP}(\Phi)$ of 0/1-LP:

$$\text{Maximize } \sum_{j=1}^{m} z_j$$

under the m linear constraints

$$\sum_{i \in \mathrm{In}^+(F_j)} y_i + \sum_{i \in \mathrm{In}^-(F_j)} (1 - y_i) \quad \geq \quad z_j \text{ for } j = 1, 2, \ldots, m \qquad (7.7)$$

and the $n + m$ constraints

[11] with respect to the uniform probability distribution

[12] If, for instance, $F_i = x_1 \vee \overline{x_3} \vee \overline{x_7} \vee x_8$, then $\mathrm{Set}(F_i) = \{x_1, \overline{x_3}, \overline{x_7}, x_8\}$.

[13] For $F_i = x_1 \vee \overline{x_3} \vee \overline{x_7} \vee x_8$, $\mathrm{Set}^+(F_i) = \{x_1, x_8\}$ and $\mathrm{Set}^-(F_i) = \{x_3, x_7\}$.

[14] For $F_i = x_1 \vee \overline{x_3} \vee \overline{x_7} \vee x_8$, $\mathrm{In}^+(F_i) = \{1, 8\}$ and $\mathrm{In}^-(F_i) = \{3, 7\}$.

$$\begin{aligned} & y_i \in \{0,1\} \text{ for } i = 1, \ldots, n, \text{ and} \\ & z_j \in \{0,1\} \text{ for } j = 1, \ldots, m. \end{aligned} \tag{7.8}$$

For $i = 1, 2, \ldots, n$, the variable y_i overtakes the role of the Boolean variable x_i in this representation. The idea of this representation LP(Φ) of Φ is that z_j can have the value 1 only if[15] at least one variable from $\mathrm{Set}^+(F_j)$ has been assigned the value 1 or at least one variable from $\mathrm{Set}^-(F_j)$ has been assigned the value 0 (i.e., only if F_j is satisfied). Thus, the objective function $\sum_{j=1}^{m} z_j$ counts the number of satisfied clauses.

The relaxed version Rel-LP(Φ) of LP(Φ) can be obtained from LP(Φ) by exchanging the $n + m$ constraints (7.8) by the following $2 \cdot n + 2 \cdot m$ linear constraints:

$$\begin{aligned} & y_i \geq 0, \; y_i \leq 1 \text{ for } i = 1, \ldots, n, \text{ and} \\ & z_j \geq 0, \; z_j \leq 1 \text{ for } j = 1, \ldots, n. \end{aligned} \tag{7.9}$$

Let $\alpha(u)$ for every $u \in \{y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_m\}$ be the value of u in an optimal solution for the instance Rel-LP(Φ) of LP. Since the set of feasible solutions of 0/1-LP(Φ) is a subset[16] of the feasible solutions of Rel-LP(Φ), the following is true:

$\sum_{j=1}^{m} \alpha(z_j)$ *is an upper bound on the number of clauses that can be satisfied by any assignment to the variables of* Φ. (7.10)

To produce an assignment to the variables $x_1, x_2, \ldots, x_n$, one can round the values $\alpha(y_1), \alpha(y_2), \ldots, \alpha(y_n)$ of the optimal solution

$$(\alpha(y_1), \ldots, \alpha(y_n), \alpha(z_1), \ldots, \alpha(z_m))$$

of Rel-LP(Φ). How to round is explained in the following presentation of the designed algorithm.

Algorithm RRR (Relaxation with Random Rounding)

Input: A formula $\Phi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$ over $X = \{x_1, \ldots, x_n\}$ in CNF, $n, m \in \mathbb{N} - \{0\}$.

Step 1: Reduce the instance Φ of MAX-SAT to the instance 0/1-LP(Φ) of 0/1-LP with the constraints (7.7) and (7.8).

Step 2: Relax 0/1-LP(Φ) to the instance Rel-LP(Φ) with the constraints (7.7) and (7.9), and solve Rel-LP(Φ) efficiently.
Let $(\alpha(y_1), \alpha(y_2), \ldots, \alpha(y_m), \alpha(z_1), \alpha(z_2), \ldots, \alpha(z_m))$ be the computed optimal solution for Rel-LP(Φ).

[15]because of the j-th constraints of (7.7)

[16]The constraints (7.8) strengthen the constraints (7.9).

Step 3:

```
    Choose, uniformly, $n$ values $\gamma_1, \ldots, \gamma_n$ from the interval $[0,1]$ at random.
    for $i = 1$ to $n$ do
        if $\gamma_i \in [0, \alpha(y_i)]$ then
            $\beta_i := 1$
        else
            $\beta_i := 0$.
```

Output: $\mathrm{RRR}(\Phi) = (\beta_1, \beta_2, \ldots, \beta_n)$

Hence, $\alpha(y_i)$ is the probability that x_i takes the value 1. The main difference with the random sampling algorithm RSAM is that RSAM takes its random choice with respect to the uniform probability distribution over $\{0,1\}^n$ while RRR chooses an assignment with respect to the probability distribution determined by $\alpha(y_1), \alpha(y_2), \ldots, \alpha(y_n)$.

We start our analysis of RRR by giving a lower bound on the probability that RRR satisfies a clause of k literals.

Lemma 7.3.9. *Let k be a positive integer and let F_j by a clause of k literals in Φ. Let $\alpha(y_1), \ldots, \alpha(y_n), \alpha(z_1), \ldots, \alpha(z_m)$ be the optimal solution of* Rel-LP (Φ) *computed by* $\mathrm{RRR}(\Phi)$.

Then the probability that the assignment $\mathrm{RRR}(\Phi)$ *satisfies the clause F_j is at least*

$$\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \alpha(z_j).$$

Proof. Since one considers the clause F_j independently of other clauses, one can assume without loss of generality that it contains only uncomplemented variables, and that it can be expressed[17] as

$$F_j = x_1 \vee x_2 \vee \ldots \vee x_k.$$

From the j-th constraint in (7.7) of Rel-LP (Φ), we have

$$y_1 + y_2 + \ldots + y_k \geq z_j. \tag{7.11}$$

The clause F_j remains unsatisfied if and only if all the variables $x_1, x_2, \ldots, x_k$ are set to zero. Since the random rounding in step 3 of RRR runs independently for each variable, this occurs with the probability

$$\prod_{i=1}^{k} (1 - \alpha(y_i)).$$

Complementary, F_j is satisfied with the probability

[17]This way we avoid double indexing and additional notation for negated variables.

$$1 - \prod_{i=1}^{k} (1 - \alpha(y_i)). \tag{7.12}$$

Under the constraints (7.11), the function (7.12) is minimized when

$$\alpha(y_i) = \frac{\alpha(z_j)}{k}$$

for $i = 1, 2, \ldots, k$. In this way we obtain

$$\text{Prob}(F_j \text{ is satisfied}) \geq 1 - \prod_{i=1}^{k} \left(1 - \frac{\alpha(z_j)}{k}\right). \tag{7.13}$$

The lower bound (7.13) on the probability of satisfying F_j can be viewed as a function of one variable $\alpha(z_j) \in [0, 1]$. To complete the proof it suffices to show that, for every positive integer k,

$$f_k(r) = 1 - \left(1 - \frac{r}{k}\right)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot r = g_k(r) \tag{7.14}$$

for all $r \in [0, 1]$ (and so for every $\alpha(z_j)$). Next, we show that the relation between the functions $g(r)$ and $f(r)$ is as shown in Figure 7.1.

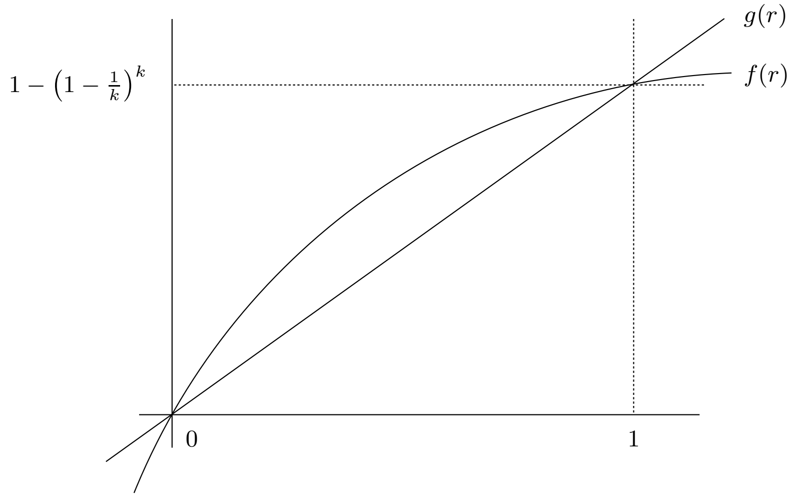


Fig. 7.1.

Since f_k is a concave function and g_k is a linear function, it is sufficient to show (Figure 7.1) that

$$f_k(0) = g_k(0) \text{ and } f_k(1) = g_k(1).$$

Clearly,

$$f_k(0) = 1 - (1-0)^k = 0 = g_r(0) \text{ and } f_k(1) = 1 - \left(1 - \frac{1}{k}\right)^k = g_k(1).$$

Inserting $r = \alpha(z_j)$ into (7.14) and combining (7.13) with (7.14) one obtains the assertion of Lemma 7.3.9. □

Theorem 7.3.10. *The algorithm* RRR *runs in polynomial time and it is*

(i) a randomized $\mathrm{E}\left[\frac{e}{(e-1)}\right]$*-approximation algorithm for* MAX-SAT *and*

(ii) a randomized $\mathrm{E}\left[\frac{k^k}{(k^k-(k-1)^k)}\right]$*-approximation algorithm for* MAX-EkSAT.

Proof. First we analyze the time complexity of the algorithm RRR. The reduction in step 1 can be performed in linear time. The instance Rel-LP (Φ) of LP can be solved in polynomial time. Step 3 can be executed in linear time.

Following (7.10), in order to show that RRR is an $\mathrm{E}[d]$-approximation algorithm it suffices to show that the expected number of satisfied clauses is at least

$$d^{-1} \cdot \sum_{j=1}^{m} \alpha(z_j).$$

Our probability space[18] is $(\{0,1\}^n, \mathrm{Prob})$, where

$$\mathrm{Prob}(\{(\delta_1, \delta_2, \ldots, \delta_n)\}) = \prod_{i=1}^{n} q_j,$$

where

$$\begin{aligned} q_i &= \alpha(y_i) && \text{if} \quad \delta_i = 1 \;\; \text{and} \\ q_i &= 1 - \alpha(y_i) && \text{if} \quad \delta_i = 0. \end{aligned}$$

for $i = 1, 2, \ldots, n$. For $\delta = (\delta_1, \ldots, \delta_n)$ and $j = 1, 2, \ldots, m$ we consider the random variable Z_j defined by

$$Z_j(\delta) = \begin{cases} 1 \text{ if } \delta \text{ satisfies the clause } F_j \\ 0 \text{ if } \delta \text{ does not satisfy the clause } F_j. \end{cases}$$

Since Z_j is an indicator variable, $\mathrm{E}[Z_j]$ is the probability that the output $\mathrm{RRR}(\Phi)$ of RRR satisfies the clause F_j. Therefore, Lemma 7.3.9 provides

$$\mathrm{E}[Z_j] \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \alpha(z_j) \tag{7.15}$$

[18] We identify each computation of RRR with its output.

if F_j consists of k literals. Let us consider the random variable

$$Z = \sum_{j=1}^{m} Z_j$$

that counts the number of satisfied clauses. If all clauses consist of exactly k literals,[19] the linearity of expectation provides the following lower bound on $\mathrm{E}[Z]$:

$$\begin{aligned}
\mathrm{E}[Z] &= \sum_{j=1}^{m} \mathrm{E}[Z_j] \\
&\underset{(7.15)}{\geq} \sum_{j=1}^{m} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \alpha(z_j) \\
&\geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \sum_{j=1}^{m} \alpha(z_j). \qquad (7.16)
\end{aligned}$$

Since $\mathrm{Opt}_{\text{MAX-SAT}}(\Phi) \underset{(7.10)}{\leq} \sum_{j=1}^{m} \alpha(z_j)$, we obtain

$$\begin{aligned}
\mathrm{E}[\mathrm{Ratio}_{\mathrm{RRR}}(\Phi)] &= \frac{\mathrm{Opt}_{\text{MAX-E}k\text{SAT}}(\Phi)}{\mathrm{E}[Z]} \\
&\underset{\substack{(7.9)\\(7.10)}}{\leq} \frac{\sum_{j=1}^{m} \alpha(z_j)}{\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \sum_{j=1}^{m} \alpha(z_j)} \\
&= \left(1 - \left(1 - \frac{1}{k}\right)^k\right)^{-1} \\
&= \frac{k^k}{k^k - (k-1)^k}.
\end{aligned}$$

Hence, we have proved claim (ii).

Since $\left(1 - \frac{1}{k}\right)^k \leq e^{-1}$ for all $k \in \mathbb{N} - \{0\}$, we have

$$1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e} \qquad (7.17)$$

for all positive integers k. Inserting (7.17) into (7.16), we obtain

$$\mathrm{E}[Z] \geq \left(1 - \frac{1}{e}\right) \cdot \sum_{j=1}^{m} \alpha(z_j).$$

Therefore

[19] if Φ is an instance of MAX-EkSAT

$$\begin{aligned}\mathrm{E}[\mathrm{Ratio}_{\mathrm{RRR}}(\varPhi)] &= \frac{\mathrm{Opt}_{\text{MAX-SAT}}(\varPhi)}{\mathrm{E}[Z]} \\ &\leq \left(1-\frac{1}{e}\right)^{-1} = \frac{e}{e-1}.\end{aligned}$$

$\square$

We note that the algorithm RRR has a special, very appreciative property with respect to the success probability amplification by repeated runs. One does not need to repeat the complete runs of RRR. Once the first two steps of RRR are executed, and so the probability space $(\{0,1\}^n, \mathrm{Prob})$ is determined, it suffices to execute several independent random choices in this probability space. Thus, the most expensive step 2 is executed only once, independently of the number of assignments one wants to generate at random.

Exercise 7.3.11. Let $\varPhi$ be a formula consisting of $m = 3 \cdot d$ clauses, where d clauses are of the length 2, the next d clauses are of length 3, and the last d clauses consist of 4 literals. Estimate a lower bound for the expectation $\mathrm{E}[\mathrm{Ratio}_{\mathrm{RRR}}(\varPhi)]$.

7.4 Combining Random Sampling and Random Rounding

Now, we have two different algorithms for MAX-SAT. On the one hand the algorithm RSAM based on random sampling, and on the other hand the algorithm RRR designed in Section 7.3 by the relaxation method and random rounding. Since $2 > \frac{e}{e-1}$, the algorithm RSAM with $\mathrm{E}[\mathrm{Ratio}_{\mathrm{RSAM}}(\varPhi)] \leq 2$ for every formula $\varPhi$ provides in general a weaker guarantee that the algorithm RRR with $\mathrm{E}[\mathrm{Ratio}_{\mathrm{RRR}}(\varPhi)] < \frac{e}{e-1}$.

Surprisingly, if one looks at the behavior of these algorithms more carefully, one sees that the naive algorithm RSAM is better for problem instances with long clauses. Since

$$\frac{2^k}{2^k-1} < \frac{k^k}{k^k-(k-1)^k}$$

for $k \geq 3$, RSAM assures a better upper bound on the expected approximation ratio for MAX-EkSAT instances than RRR. Hence, one can consider RSAM and RRR as incomparable algorithms for MAX-SAT, because for some formulas RSAM can provide better results than RRR, and vice versa.

Exercise 7.4.12. Find an infinite set of input instances of MAX-SAT for which the expected solutions computed by RSAM are better than the expected solutions computed by RRR.

Exercise 7.4.13. Find an infinite set of instances of MAX-SAT such that one can expect better results from RRR than from RSAM.

Exercise 7.4.14. Let k be a positive integer, $k \geq 3$. Find instances $\varPhi$ of MAX-EkSAT such that

$$\mathrm{E}\left[\mathrm{Ratio}_{\mathrm{RRR}}(\varPhi)\right] < \mathrm{E}\left[\mathrm{Ratio}_{\mathrm{RSAM}}(\varPhi)\right].$$

Because of the incomparability of RSAM and RRR, a very natural idea is to combine these algorithms into one algorithm by running them independently in parallel and then taking the better of the two solutions computed. Next, we show that the resulting algorithm has an expected approximation ratio of at most 4/3.

Algorithm COMB

```
Input: A formula $\varPhi = F_1 \wedge F_2 \wedge \ldots \wedge F_m$ in CNF over a set $X$ of Boolean
    variables.
Step 1: Compute an assignment $\beta$ for $X$ by the algorithm RSAM (i.e., $\beta :=$
    RSAM $(\varPhi)$).
Step 2: Compute an assignment $\gamma$ for $X$ by the algorithm RRR (i.e., $\gamma :=$
    RRR $(\varPhi)$).
Step 3:
    if $\beta$ satisfies more clauses of $\varPhi$ than $\gamma$ then
        output ($\beta$)
    else
        output ($\gamma$).
```

Theorem 7.4.15. *The algorithm* COMB *is a polynomial-time, randomized* $\mathrm{E}\left[\frac{4}{3}\right]$*-approximation algorithm for* MAX-SAT*.*

Proof. Since both RSAM and RRR work in polynomial time, COMB is a polynomial-time algorithm, too.

Now, we analyze the expected approximation ratio of COMB. Let $\varPhi = F_1 \wedge \ldots \wedge F_m$ be a formula over $X = \{x_1, \ldots, x_n\}$. Let $(S_{\mathrm{RSAM},\varPhi}, \mathrm{Prob}_{\mathrm{RSAM}})$ be the probability space for the analysis of RSAM, where $S_{\mathrm{RSAM},\varPhi}$ is the set of all 2^n computations of RSAM on $\varPhi$ and $\mathrm{Prob}_{\mathrm{RSAM}}$ is the uniform probability distribution over $S_{\mathrm{RSAM},\varPhi}$. Let $(S_{\mathrm{RRR},\varPhi}, \mathrm{Prob}_{\mathrm{RRR}})$ be the probability space for the analysis of the work of RRR, where $S_{\mathrm{RRR},\varPhi}$ is the set of all 2^n computations of RRR on $\varPhi$ and $\mathrm{Prob}_{\mathrm{RRR}}$ is the probability distribution determined by the optimal solution for Rel-LP$(\varPhi)$ computed in the second step of RRR. Obviously,

$$(S_{\mathrm{RSAM},\varPhi} \times S_{\mathrm{RRR},\varPhi}, \mathrm{Prob}),$$

with

$$\mathrm{Prob}(\{(C, D)\}) = \mathrm{Prob}_{\mathrm{RSAM}}(\{C\}) \cdot \mathrm{Prob}_{\mathrm{RRR}}(\{D\})$$

for all $(C, D) \in S_{\mathrm{RSAM},\varPhi} \times S_{\mathrm{RRR},\varPhi}$ is the probability space[20] for the analysis of the work of COMB on $\varPhi$. Let us consider the following random variables.

[20]For simplicity we view the computations of COMB as pairs (C, D), where C is a run of RSAM and D is a run of RRR.

For all $(C, D) \in S_{\mathrm{RSAM},\Phi} \times S_{\mathrm{RRR},\Phi}$, let

$Y((C, D))$ be the number of clauses satisfied by β as the output of C

and let

$Z((C, D))$ be the number of clauses satisfied by the output γ of D.

Hence, Y counts the number of clauses satisfied by the output of a run of RSAM, and Z counts the number of clauses satisfied by the assignment computed by a run of RRR. We introduce a new random variable $U = \max\{Y, Z\}$, defined by

$$U((C, D)) = \max\{Y((C, D)), Z((C, D))\}$$

for all computations (C, D) of COMB.

Hence, U counts the number of clauses satisfied by the output of a run (C, D) of the algorithm COMB. Since

$$\max\{Y((C, D)), Z((C, D))\} \geq \frac{Y((C, D)) + Z((C, D))}{2}$$

for all $(C, D) \in S_{\mathrm{RSAM},\Phi} \times S_{\mathrm{RRR},\Phi}$, we have

$$\mathrm{E}[U] \geq \frac{\mathrm{E}[Y] + \mathrm{E}[Z]}{2}. \tag{7.18}$$

From the analysis of RRR, we know that no assignment can satisfy more than $\sum_{j=1}^{m} \alpha(z_j)$ clauses.

Following (7.18), it is sufficient to show that

$$\frac{\mathrm{E}[Y] + \mathrm{E}[Z]}{2} \geq \frac{3}{4} \cdot \sum_{j=1}^{m} \alpha(z_j). \tag{7.19}$$

To show this, we investigate the probability of satisfying every particular clause with respect to its length. For each integer $k \geq 1$, let $C(k)$ be the set of clauses from $\{F_1, F_2, \ldots, F_m\}$ that consist of exactly k literals. Lemma 7.3.9 implies that

$$\mathrm{E}[Z] \geq \sum_{k \geq 1} \sum_{F_j \in C(k)} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \alpha(z_j). \tag{7.20}$$

Since $\alpha(z_j) \in [0, 1]$, the analysis of RSAM provides the following lower bound on $\mathrm{E}[Y]$:

$$\mathrm{E}[Y] = \sum_{k \geq 1} \sum_{F_j \in C(k)} \left(1 - \frac{1}{2^k}\right) \geq \sum_{k \geq 1} \sum_{F_j \in C(k)} \left(1 - \frac{1}{2^k}\right) \cdot \alpha(z_j). \tag{7.21}$$

Hence, we obtain

$$\begin{aligned}
\mathrm{E}[U] &\underset{(7.18)}{\geq} \frac{\mathrm{E}[Y]+\mathrm{E}[Z]}{2}\\
&\underset{\substack{(7.20)\\(7.21)}}{\geq} \frac{1}{2}\cdot\sum_{k\geq 1}\sum_{F_j\in C(k)}\left[\left(1-\frac{1}{2^k}\right)+\left(1-\left(1-\frac{1}{k}\right)^k\right)\right]\cdot\alpha(z_j)\\
&\geq \frac{1}{2}\cdot\frac{3}{2}\cdot\sum_{k\geq 1}\sum_{F_j\in C(k)}\alpha(z_j)\\
&\quad \{\text{Since } (1-2^{-k})+(1-(1-k^{-1})^k)\geq \tfrac{3}{2} \text{ for all positvie integers } k.\}\\
&= \frac{3}{4}\cdot\sum_{j=1}^{m}\alpha(z_j).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\mathrm{E}[\mathrm{Ratio}_{\mathrm{COMB}}(\Phi)] &= \frac{\mathrm{Opt}_{\text{MAX-SAT}}(\Phi)}{\mathrm{E}[U]}\\
&\leq \frac{\sum_{j=1}^{m}\alpha(z_j)}{\frac{3}{4}\cdot\sum_{j=1}^{m}\alpha(z_j)}\\
&= \frac{4}{3}.
\end{aligned}$$

□

Exercise 7.4.16. Implement the algorithm COMB and test it for real MAX-SAT instances. Try to estimate the average approximation ratio with respect to your input data.

7.5 Summary

The relaxation to linear programming is a robust method for designing approximation algorithms for hard optimization problems as well as for computing bounds on the costs of optimal solutions. The kernel of this method is that many problems can be naturally expressed as instances of the NP-hard integer linear programming and 0/1-linear programming, and that the basic problem of linear programming is efficiently computable. Based on these facts, one obtains the following schema for applying this method.

1. Reduction

A given instance of an optimization problem is expressed as an instance of ILP or 0/1-LP.

2. Relaxation

The instance of ILP or 0/1-LP is relaxed to an instance of LP[21] by removing the nonlinear constraints of the domain of the output values. The instance of LP is efficiently solved.

3. Solving the original problem instance

The optimal solution of the relaxed LP instance is used to create a feasible solution for the original problem instance. The cost of the computed optimal solution for the relaxed LP instance is a bound on the achievable cost of the feasible solutions for the original problem instance.

While, today, parts 1 and 2 of this schema can be performed by standard, efficient algorithms, part 3 may require that we apply distinct concepts and methods, and it is a matter of investigation for various concrete optimization problems.

In this book we presented two techniques for implementing part 3 of the schema. First, we used simple rounding for the minimum vertex cover problem, and got a polynomial-time 2-approximation algorithm for MIN-VC in this way.

In Section 7.3 we applied random rounding to design a randomized, polynomial-time $\mathrm{E}[e/(e-1)]$-approximation algorithm for MAX-SAT. If one executes this algorithm and the algorithm RSAM[22] on the same MAX-SAT instance in parallel, and then takes the better of the two computed assignments, one obtains a $\mathrm{E}[4/3]$-approximation algorithm for MAX-SAT.

The most involved source for the study of the design of approximation algorithms by the relaxation to linear programming is the textbook by Vazirani [Vaz01]. A detailed introduction to the application of this method is presented in [Hro03], too. There are many good textbooks on linear programming. For computer scientists, we warmly recommend the excellent, classical book by Papadimitriou and Steiglitz [PS82].

The famous Simplex algorithm for solving instances of linear programming was discovered by Dantzig [Dan49] in 1947. Klee and Minty [KM72] were the first researchers who constructed LP instances on which the Simplex algorithm does not work efficiently.[23] The long stated open problem about the existence of a polynomial-time algorithm for LP was solved by Khachian [Kha94] in 1979 in a positive sense.

An excellent source for the study of MAX-SAT is the book [MPS98] edited by Mayr, Prömel, and Steger.

[21] All constraints of the relaxed instance are linear equations or inequalities.

[22] which is based on a simple application of the method of random sampling

[23] i.e., in exponential time

A

Fundamentals of Mathematics

Chance favors only those
whose spirit has been prepared already,
those unprepared cannot see the hand
stretched out to them by fortune.

Louis Pasteur

A.1 Objectives

In computer science, one views mathematics on the one hand as a language for precise text formulations with an unambiguous interpretation, and on the other hand as a collection of instruments for analyzing and solving various situations and tasks. Mathematics is a living language that continuously grows and creates new notions and models that enable us to describe more and more about our world and to argue more and more in depth.

Methods of mathematics are developing intensively in order to master further real-world problems. The history of mathematics is full of impulses coming from other scientific disciplines, engineering and practice. Computer science has a very special relationship with mathematics. It does not only pose requirements and wishes for developing new terms and new formal methods and does not only use mathematics as its basic instrument. Computer science in its own interest also takes a very active part in the development of mathematics by formulating new formal concepts, by creating new fundamental terms, and by discovering essential, deep facts of pure mathematical nature. The famous PCP Theorem and the deterministic polynomial-time primality test are only two of the well-known examples. Also the design of randomized algorithms, and so the content of this book, can be assigned to mathematics as well as to computer science.

Our introduction to randomization is built on elementary notions and instruments of probability theory. Therefore, we started this book with an introduction to probability theory. But, in different parts of this textbook, we also need methods and knowledge of other areas of mathematics, such as combinatorics, graph theory, algebra, and number theory. The objective of this appendix is to present the most frequently used results and concepts of mathematics applied here, and so to make this textbook self-contained.

In this short appendix we do not aim to provide an involved introduction to several areas of mathematics. We give detailed explanations and proof for only those results that can contribute to the understanding of the design of randomized algorithms and are helpful in developing corresponding ideas and approaches. Some assertions, though needed in our arguments, we present without proofs if the proofs do not have any strong relation to the design and analysis of randomized algorithms.

Section A.2 is devoted to number theory and algebra, which are crucial for success in finding efficient randomized algorithms. The content of this section is especially important for the design of algorithms for the number-theoretical problems in Chapters 4, 5, and 6. The Fundamental Theorem of Arithmetics, Fermat's Little Theorem, the Chinese Remainder Theorem, the Euclidean algorithm, and Lagrange's Theorem are the most frequently applied discoveries of algebra and number theory used as powerful instruments in the design of randomized algorithms. Therefore, the derivation of these results is carefully explained and presented in detail. Also, the Prime Number Theorem (one of the most fundamental discoveries of mathematics) is one of the most frequently applied results of mathematics. In spite of its high importance, we omit its proof. We do this not only because of the high complexity of its proof, but mainly because the techniques of the proof are not relevant to any of the ideas developed in our applications.

Section A.3 briefly presents a few combinatorial arguments that are applied to the analysis of the algorithms designed here.

A.2 Algebra and Number Theory

We use the following notion for the sets considered here:

$\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of all natural numbers,

$\mathbb{Z}$ is the set of all integers,

$\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ is the finite set of all natural numbers smaller than n,

$\mathbb{Q}$ is the set of rational numbers,

$\mathbb{R}$ is the set of all real numbers,

$\mathbb{Q}^+$ is the set of positive rational numbers, and

$\mathbb{R}^+$ is the set of positive real numbers.

Let a and k be natural numbers, and let d be a positive integer.

If

$$a = k \cdot d,$$

then we say that[1]

$$d \textbf{ divides } a \text{ and write } \boldsymbol{d \mid a}.$$

If $a = k \cdot d$ and, additionally, $k \geq 1$, then we say that

$$a \text{ is a } \textbf{multiple} \text{ of } d.$$

[1]It is an agreement that every positive integer divides the number 0.

If $d \mid a$, we also say that d is a **divisor** of a. For every positive integer a, we call the numbers 1 and a **trivial divisors** of a. All divisors of a except 1 and a itself are called **factors** of a. For instance, $2, 3, 4, 6, 8$, and 12 are the factors of 24.

Definition A.2.1. *A* **prime** *is any positive integer $p > 1$ that does not have any factor (i.e., 1 and p are the only divisors of p).*

An integer b that is not a prime (i.e., $b = a \cdot c$ for some $a, c > 1$) is called **composite**.

The smallest primes are $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots$. Primes play an important role in number theory and algebra, and so also in the design of randomized algorithms. The first crucial and useful property of primes is that tall positive integers greater than 1 can be represented as a product of primes. Here, one considers a single prime as a product of primes, too.

Observation A.2.2. Every $n \in \mathbb{N} - \{0, 1\}$ can be represented as a product of primes.

Proof. We prove this assertion by induction. Clearly, the prime 2 has to be considered a product of primes.

Assume that every natural number (except 0 and 1) smaller than n, $n \geq 3$, can be represented as a product of primes. We aim to show that n can be represented as a product of primes, too.

If n is a prime, then we have already the desired representation.

If n is composite, then

$$n = p \cdot q,$$

where p and q are factors of n. Since both p and q are smaller than n, the induction hypothesis says that p and q can be represented as products of primes. Hence, $n = p \cdot q$ can be represented as a product of primes. □

In what follows we present the products of primes in the form

$$n = p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k}, \tag{A.1}$$

where $p_1 < p_2 < \ldots < p_k$ are primes and $i_1, i_2, \ldots, i_k$ are positive integers. The following theorem says that every positive integer n has a unique representation of this form. We also call this form, $n = p_1^{i_1} \cdot \ldots \cdot p_k^{i_k}$, the **factorization** of n.

Theorem A.2.3. The Fundamental Theorem of Arithmetics

Every integer n, $n \geq 2$, can be uniquely represented as a product of primes

$$n = p_1^{i_1} \cdot p_2^{i_2} \cdot \ldots \cdot p_k^{i_k},$$

where $p_1 < p_2 < \ldots < p_k$ are primes and $i_1, i_2, \ldots, i_k \in \mathbb{N} - \{0\}$, $k \in \mathbb{N} - \{0\}$.

Proof. The fact that n can be expressed as a product of primes has already been shown in Observation A.2.2. Next, we have to prove that the decomposition of n into a product of primes is unique up the the rearrangement of factors (i.e., that the factorization of n is unique). We prove this for every integer $n \geq 2$ in an indirect way.

Let m be the smallest integer from $\mathbb{N} - \{0, 1\}$ such that

$$m = p_1 \cdot p_2 \cdot \ldots \cdot p_r = q_1 \cdot q_2 \cdot \ldots \cdot q_k, \tag{A.2}$$

where $p_1 \leq p_2 \leq \ldots \leq p_r$ and $q_1 \leq q_2 \leq \ldots \leq q_k$ are primes and these two representations are different (i.e., $r = k$ and $p_b = q_b$ for $b = 1, \ldots, k$ does not hold).

We distinguish three cases.

(i) *$r = 1$ or $k = 1$.*
The case $r = k = 1$ cannot occur, because it would mean $m = p_1 = q_1$ for $p_1 \neq q_1$.
The case $r = 1$ and $k > 1$ also cannot occur, because it contradicts to the assumption that p_1 is a prime. Analogously, $k = 1$ and $r \geq 2$ contradicts the primality of q_1.

(ii) *There exists an $l \in \{1, \ldots, r\}$ and an $s \in \{1, \ldots, k\}$ such that $p_l = q_s$ and $r > 1$, $k > 1$.*
In this case, the number

$$m/p_l = p_1 \cdot p_2 \cdot \ldots \cdot p_{l-1} \cdot p_{l+1} \cdot p_r = q_1 \cdot q_2 \cdot \ldots \cdot q_{s-1} \cdot q_{s+1} \cdot q_k$$

has two different factorizations. Since $m/p_l < m$, this is a contradiction to the assumption that m is the smallest number without a unique factorization.

(iii) *$\{p_1, p_2, \ldots, p_r\} \cap \{q_1, q_2, \ldots, q_k\} = \emptyset$ and $r > 1$, $k > 1$.*
Without loss of generality, we assume that $p_1 < q_1$. Consider the number

$$m' = m - (p_1 q_2 q_3 \ldots q_s). \tag{A.3}$$

Substituting the two representations (A.2) of m for m into (A.3), we may write the integer m' in either of the two forms

$$m' = (p_1 p_2 \ldots p_r) - (p_1 q_2 \ldots q_s) = p_1(p_2 p_3 \ldots p_r - q_2 q_3 \ldots q_s) \tag{A.4}$$

and

$$m' = (q_1 q_2 \ldots q_s) - (p_1 q_2 \ldots q_s) = (q_1 - p_1) \cdot (q_2 q_3 \ldots q_s). \tag{A.5}$$

Since $p_1 < q_1$, (A.3) and (A.5) together imply that

$$2 \leq m' < m.$$

From our assumption, m' must have a unique factorization. The equation (A.4) implies that the prime p_1 is a factor of m'. Since in case (iii) one assumes $p_1 \notin \{q_2, q_3, \ldots, q_s\}$, from representation (A.5) of m', we have that p_1 is a factor of $q_1 - p_1$. Hence, there is an integer $a \geq 1$, such that

$$q_1 - p_1 = p_1 \cdot a.$$

Therefore,

$$q_1 = p_1 \cdot a + p_1 = p_1 \cdot (a + 1),$$

which contradicts to the fact that q_1 is a prime.
Thus, in all three cases (i), (ii), and (iii) the assumption about the existence of a positive integer with at least two different factorizations leads to a contradiction, and so the assertion of Theorem A.2.3 holds.

□

The Fundamental Theorem of Arithmetics assures the following divisibility property, which is often applied in this book.

Corollary A.2.4. *Let p be a prime, and let a, b be positive integers. If*

$$p \mid a \cdot b,$$

then

$$p \mid a \text{ or } p \mid b.$$

Proof. Again, we give an indirect proof. Assume that $p \mid a \cdot b$, and that p is a factor of neither a nor b. Hence, the factorizations of a and b do not contain p, and so the corresponding factorization of $a \cdot b$ does not contain p.

Since p divides $a \cdot b$, there exists a positive integer t such that

$$a \cdot b = p \cdot t.$$

Clearly, the product of p and the factorization of t is also a factorization of $a \cdot b$, and this factorization of $a \cdot b$ contains the prime p.

Thus, we have two different factorizations of $a \cdot b$, which contradicts the Fundamental Theorem of Arithmetics. □

Exercise A.2.5. Prove the following assertion. For all positive integers a, b, and c, if c divides the product $a \cdot b$, then there exist two positive integers c_1 and c_2 such that

$$c = c_1 \cdot c_2, \; c_1 \mid a \text{ and } c_2 \mid b.$$

Exercise A.2.6. Prove the following assertion. Let a, b, c, and d be arbitrary positive integers. If $a = b + c$ and d divides both a and b, then d divides c, too.

Exercise A.2.7. Prove the following assertion. Let p be a prime. Then there exists no positive integer n such that p divides both n and $n - 1$.

One of the first questions that arose concerning primes was whether there are infinitely many different primes or whether the cardinality of the class of primes is an integer. The following assertion answers this first, simple question about primes.

Theorem A.2.8. *There are infinitely many primes.*

Proof. We present the original proof of Euclid as an indirect proof.

Assume there are finitely many different primes, say $p_1, p_2, \ldots, p_n$ for a positive integer n. Any other number is composite, and so must be divisible by at least one of the primes $p_1, p_2, \ldots, p_n$. Consider the specific number

$$a = p_1 \cdot p_2 \cdot \ldots \cdot p_n + 1.$$

Since $a > p_i$ for every $i \in \{1, 2, \ldots, n\}$, a must be composite, and so at least one of the primes must divide a. Clearly[2] this is impossible and so we have a contradiction. Hence, we have proved that there are infinitely many primes. $\square$

Note that the proof of Theorem A.2.8 does not provide any method for constructing an arbitrarily large sequence of primes. If $p_1, p_2, \ldots, p_n$ are primes, the the number $a = p_1 \cdot p_2 \cdot \cdots \cdot \ldots \cdot p_n + 1$ does not need to be a prime. For instance, for $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, $p_4 = 7$, $p_5 = 11$, and $p_6 = 13$, the number

$$a = 30031 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 59 \cdot 509$$

is composite. Moreover, if a is a prime and $p_1, p_2, \ldots, p_n$ are the n smallest primes, then a does not need to be the $(n+1)$-th smallest prime. For instance,

$$7 = 2 \cdot 3 + 1$$

and 7 is the fourth smallest prime, not the third one. The only right conclusion of the proof of Theorem A.2.8 is that there must exist a prime p greater than p_n and smaller than or equal to a. Therefore, once can search in the interval $(p_n, a]$ for the $(n+1)$-th smallest prime.

A large effort in number theory was devoted to finding a simple mathematical formula for efficiently generating the sequence of all primes. Unfortunately, these attempts have not succeeded up to the present time, and it is questionable whether such a formula exists. On the other hand, another important question,

How many primes are contained in $\{1, 2, \ldots, n]$*?*

[2]Let there be a $j \in \{1, \ldots, n\}$ such that $p_j \mid a$. Obviously, p_j divides the product $p_1 \cdot p_2 \cdot \ldots \cdot p_{j-1} \cdot p_j \cdot p_{j+1} \cdot \ldots \cdot p_n$. Following Exercise A.2.6, if p_j divides both a and $p_1 \cdot \ldots \cdot p_n$, and $a = p_1 \cdot \ldots \cdot p_n + 1$, then p_j must divide 1, too.

has been answered to some extent. The investigation of the density of primes in $\mathbb{N}$ is of special importance for the design of randomized algorithms and systems. The following Prime Number Theorem is one of the most fundamental discoveries of mathematics, with an enormous impact and many relations outside number theory.

In what follows we denote by *Prim* (n) the cardinality of the set PRIM (n) of all primes in $\{1, 2 \ldots, n\}$.

Theorem A.2.9. Prime Number Theorem

$$\lim_{n\to\infty} \frac{\mathit{Prim}\,(n)}{n/\ln n} = 1.$$

In other words, the Prime Number Theorem says that the density

$$(\mathit{Prim}\,(n))/n$$

of the primes among the first n positive integers tends to

$$1/\ln n$$

as n increases. Table A.1 shows that $1/\ln n$ is a good "approximation" of *Prim* $(n)/n$ already for "small" n (for which one is able to compute PRIM (n) exactly by a computer).

Table A.1.

n	$\mathit{Prim}\,(n)/n$	$1/\ln n$	$\frac{\mathit{Prim}(n)}{n/\ln n}$
10^3	0.168	0.145	1.159
10^6	0.0885	0.0724	1.084
10^9	0.0508	0.0483	1.053

We call attention to the fact, that, for $n \geq 100$, it is even provable that

$$1 \leq \frac{\mathit{Prim}\,(n)}{n/\ln n} \leq 1.23, \tag{A.6}$$

and these bounds for *Prim* (n) are in fact, what we primarily use in the analysis of error probability in this book.

Obviously, for every natural number a and every positive integer d, one can express a by means of d as follows:

$$a = k \cdot d + r \text{ for all } k, r \in \mathbb{N} \text{ and } r < d.$$

Clearly, the number k is the largest integer, for which $k \cdot d \leq a$, and we denote k as $\boldsymbol{a}$ **div** $\boldsymbol{d}$. The number r is the remainder of the division of a by d. For given a and d the representation $k \cdot d + r$ is unique, i.e., the numbers k and r are unambiguously determined. Let us prove this claim. Assume, that one has the following two representations of a:

$$k_1 \cdot d + r_1 = a = k_2 \cdot d + r_2,$$

where $r_1, r_2 \in \{0, 1, \ldots, d-1\}$, $r_2 \geq r_1$. Subtracting r_1 and $k_2 \cdot d$ from both sides of this equation results in

$$k_1 \cdot d - k_2 \cdot d = r_2 - r_1$$

and so in

$$(k_1 - k_2) \cdot d = r_2 - r_1.$$

Hence, d must be divide $r_2 - r_1$. Since $(r_2 - r_1) \in \{0, 1, \ldots, d-1\}$, one has

$$r_2 - r_1 = 0 \text{ and so } r_1 = r_2.$$

Therefore

$$(k_1 - k_2) \cdot d = 0.$$

Since $d > 0$, we obtain

$$k_2 - k_1 = 0, \text{ and so } k_1 = k_2.$$

Let a be a positive integer, and let d be a natural number. Let r and k be the unambiguously determined numbers with

$$a = k \cdot d + r \text{ and } r < d.$$

In what follows we denote r by $\boldsymbol{a}$ **mod** $\boldsymbol{d}$. As already noted, the number k is denoted by $\boldsymbol{a}$ **div** $\boldsymbol{d}$.

For all natural numbers a and b and any positive integer d, the equation

$$a \bmod d = b \bmod d$$

is denoted by the following Gauss' congruence relation.

$$\boldsymbol{a \equiv b \pmod{d}}.$$

Observation A.2.10. For all positive integers a, b, and d, $a \geq b$, the following claims are all equivalent:

(i) $a \equiv b \pmod{d}$,
(ii) d divides $a - b$,
(iii) $a = b + k \cdot d$ for a $k \in \mathbb{N}$.

Proof. We prove the implications (i) $\Rightarrow$ (ii), (ii) $\Rightarrow$ (iii), and (iii) $\Rightarrow$ (i), which together prove the equivalence claimed.

(a) (i) $\Rightarrow$ (ii)
From the definition of $a \equiv b \pmod{d}$, there exists a $z \in \mathbb{N}$, such that

$$z = a \bmod d = b \bmod d.$$

In other words,

$$a = l \cdot d + z \text{ and } b = l' \cdot d + z$$

for some $l, l' \in \mathbb{N}$. Then,

$$a - b = l \cdot d + z - (l' \cdot d + z) = l \cdot d - l' \cdot d = (l - l') \cdot d.$$

Hence, d divides $a - b$.

(b) (ii) $\Rightarrow$ (iii)
If d divides $a - b$, then

$$a - b = k \cdot d$$

for some natural number k. Adding b to both sides of this equality we directly obtain

$$a = k \cdot d + b.$$

(c) (iii) $\Rightarrow$ (ii)
Let $a = b + k \cdot d$ for a natural number k. As we already have learned, one can express a and b as follows:

$$a = l \cdot d + z_1 \text{ and } b = l' \cdot d + z_2$$

for suitable $l, l', z_1, z_2 \in \mathbb{N}$, $z_1 < d$, $z_2 < d$.
Inserting these expressions in $a = b + k \cdot d$, we obtain

$$l \cdot d + z_1 = l' \cdot d + z_2 + k \cdot d = (l' + k) \cdot d + z_2,$$

and so $z_1 = z_2$ due to the existence of unique numbers $a \bmod d$ and $a \operatorname{div} d$.

$\square$

Definition A.2.11. *For all positive integers a and b, the* **greatest common divisor of a and b** *is*

$$\mathbf{GCD}\,(\boldsymbol{a}, \boldsymbol{b}) = \max\{d \in \mathbb{N} \mid d \textit{ divides both } a \textit{ and } b\}.$$

The **lowest common multiple of a and b** *is*

$$\mathbf{LCM}\,(\boldsymbol{a}, \boldsymbol{b}) = \min\{c \in \mathbb{N} \mid a \textit{ divides } c \textit{ and } b \textit{ divides } c\}.$$

By convention $\mathrm{GCD}\,(0,0) = \mathrm{LCM}\,(0,0) = 0$.

Exercise A.2.12. Prove that for all positive integers a and b,

$$a \cdot b = \mathrm{GCD}\,(a, b) \cdot \mathrm{LCM}\,(a, b)\,.$$

The Fundamental Theorem of Arithmetics directly implies the following assertion.

Observation A.2.13. Let

$$a = p_1 \cdot \ldots \cdot p_r \cdot q_1 \cdot \ldots \cdot q_s \text{ and } b = p_1 \cdot \ldots \cdot p_r \cdot h_1 \cdot \ldots \cdot h_m$$

be factorizations of a and b, $a, b \in \mathbb{N} - \{0, 1\}$, where $\{q_1, \ldots, q_s\} \cap \{h_1, \ldots, h_m\} \neq \emptyset$. Then,

$$\mathrm{GCD}\,(a, b) = p_1 \cdot \ldots \cdot p_r$$

and

$$\mathrm{LCM}\,(a, b) = p_1 \cdot \ldots \cdot p_r \cdot q_1 \cdot \ldots \cdot q_s \cdot h_1 \cdot \ldots \cdot h_m.$$

Exercise A.2.14. Prove the assertion of Observation A.2.13.

Computing the factorization of a number is a hard algorithmic task, and so we are frequently required to efficiently estimate GCD (a, b) without knowing the factorizations of a and b. An efficient computation of GCD (a, b) is provided by Euclid' algorithm, which is based on the following assertions.

Lemma A.2.15. *Let $a, b, d \in \mathbb{N}$, $d > 0$.*

(i) If $d \mid a$ and $d \mid b$, then

$$d \mid (a \cdot x + b \cdot y)$$

for all $x, y \in \mathbb{Z}$.

(ii) If $a \mid b$ and $b \mid a$, then $a = b$.

Proof. First, we prove (i), and then, (ii).

(i) If $d \mid a$ and $d \mid b$, then

$$a = n \cdot d \text{ and } b = m \cdot d$$

for suitable natural numbers n and m. Let x and y be arbitrary integers. Then,

$$a \cdot x + b \cdot y = n \cdot d \cdot x + m \cdot d \cdot y = d \cdot (n \cdot x + m \cdot y),$$

and so d divides the number $a \cdot x + b \cdot y$.

(ii) The fact $a \mid b$ implies $a \leq b$. Analogously, $b \leq a$ follows from the fact $b \mid a$. Therefore, $a = b$.

□

The following properties of GCD (a, b) are direct consequences of Definition A.2.11.

Observation A.2.16. For all positive integers a, b, and k,

(i) GCD $(a, b) =$ GCD (b, a),

(ii) GCD $(a, k \cdot a) = a$,

(iii) $\text{GCD}\,(a,0) = a$, and
(iv) if $k \mid a$ and $k \mid b$, then $k \mid \text{GCD}\,(a,b)$.

Exercise A.2.17. Prove that GCD as an operator is associative, i.e., prove that, for all natural numbers a, b, and c,

$$\text{GCD}\,(a, \text{GCD}\,(b,c)) = \text{GCD}\,(\text{GCD}\,(a,b)\,, c)\,.$$

The most important property of $\text{GCD}\,(a,b)$ for its efficient computation is presented in the following lemma.

Lemma A.2.18. *For all $a, b \in \mathbb{N}$, $b > 0$,*

$$\text{GCD}\,(a,b) = \text{GCD}\,(b, a \bmod b)\,.$$

Proof. We shall prove that $\text{GCD}\,(a,b)$ and $\text{GCD}\,(b, a \bmod b)$ divide each other, and so by Lemma A.2.15(ii) they must be equal.

(i) *First, we show that* $\text{GCD}\,(a,b)$ *divides* $\text{GCD}\,(b, a \bmod b)$*.*
From the definition of GCD, $\text{GCD}\,(a,b) \mid a$ and $\text{GCD}\,(a,b) \mid b$ must be true. We know that we can unambiguously express a with respect to d as follows:

$$a = (a \text{ div } b) \cdot b + a \bmod b, \qquad (A.7)$$

where $a \text{ div } b \in \mathbb{N}$. Hence,

$$a \bmod b = a - (a \text{ div } b) \cdot b,$$

i.e., $a \bmod b$ is a linear combination[3] of a and b. From Lemma A.2.15(i), $\text{GCD}\,(a,b)$ divides $a - (a \text{ div } b) \cdot b$, and so

$$\text{GCD}\,(a,b) \mid a \bmod b.$$

This, together with the obvious fact

$$\text{GCD}\,(a,b) \mid b,$$

implies (see Observation A.2.16(iv), if not clear) that

$$\text{GCD}\,(a,b) \mid \text{GCD}\,(b, a \bmod b)\,.$$

(ii) *We prove* $\text{GCD}\,(b, a \bmod b)$ *divides* $\text{GCD}\,(a,b)$*.*
Obviously,

$$\text{GCD}\,(b, a \bmod b) \mid b \text{ and } \text{GCD}\,(b, a \bmod b) \mid (a \bmod b).$$

Since

$$a = (a \text{ div } b) \cdot b + a \bmod b,$$

[3] $xa + yb$ for $x = 1$ and $y = -(a \text{ div } b)$

a is a linear combination of b and $a \bmod b$, and so

$$\mathrm{GCD}\,(b, a \bmod b) \mid a$$

by Lemma A.2.15(i). The facts

$$\mathrm{GCD}\,(b, a \bmod b) \mid b \text{ and } \mathrm{GCD}\,(b, a \bmod b) \mid a$$

together imply that (see Observation A.2.16(iv))

$$\mathrm{GCD}\,(b, a \bmod b) \mid \mathrm{GCD}\,(a, b)\,.$$

□

Lemma A.2.15 directly implies the correctness of the following recursive algorithm computing $\mathrm{GCD}\,(a, b)$ for arbitrary natural numbers a and b.

Euclid's Algorithm

```
Input: two natural numbers
Recursive Procedure EUCLID$(a, b)$:
    if $b = 0$ then
        output "$a$"
    else
        EUCLID$(b, a \bmod b)$
```

Obviously, Euclid's algorithm cannot recurse indefinitely, because the second argument strictly decreases in each recursive call. More precisely, the larger of two arguments is at least halved in each recursive step. Hence, the number of recursive calls is at most $O(\log_2 b)$.

Thus, we have got an efficient algorithm for computing the greatest common divisor of two natural numbers. The following example illustrates how fast Euclid's algorithm works for $a = 127500136$ and $b = 12750$.

$$\begin{aligned}\mathrm{EUCLID}\,(127500136, 12750) &= \mathrm{EUCLID}\,(12750, 136)\\ &= \mathrm{EUCLID}\,(136, 102)\\ &= \mathrm{EUCLID}\,(102, 34)\\ &= \mathrm{EUCLID}\,(34, 0)\\ &= 34.\end{aligned}$$

An important characterization[4] of the greatest common divisor of two natural numbers is given by the following theorem.

[4]equivalent definition of GCD.

Theorem A.2.19. *For all positive integers a and b, let*

$$\mathbf{Comb}\,(\boldsymbol{a},\boldsymbol{b}) = \{a \cdot x + b \cdot y \mid x, y \in \mathbb{Z}\}$$

be the set of all linear combinations of a and b. Then,

$$\text{GCD}\,(a,b) = \min\{d \in \text{Comb}\,(a,b) \mid d \geq 1\},$$

i.e., $\text{GCD}\,(a,b)$ *is the smallest positive integer n in* $\text{Comb}\,(a,b)$.

Proof. Let $h = \min\{d \in \text{Comb}\,(a,b) \mid d \geq 1\}$, and let

$$h = a \cdot x + b \cdot y$$

for some $x, y \in \mathbb{Z}$. We prove that $h = \text{GCD}\,(a,b)$ by proving the inequalities $h \leq \text{GCD}\,(a,b)$ and $h \geq \text{GCD}\,(a,b)$ separately.

(i) First we prove that h divides a as well as b, and so $h \mid \text{GCD}\,(a,b)$, too. From the definition of $a \bmod b$, we have

$$\begin{aligned} a \bmod h &= a - \lfloor a/h \rfloor \cdot h \\ &= a - \lfloor a/h \rfloor \cdot (a \cdot x + b \cdot y) \\ &\quad \{\text{since } h = a \cdot x + b \cdot y\} \\ &= a \cdot (1 - \lfloor a/h \rfloor \cdot x) + b \cdot (-\lfloor a/h \rfloor \cdot y), \end{aligned}$$

and so $a \bmod h$ is a linear combination of a and b. Since h is the lowest positive linear combination of a and b and $a \bmod h < h$, we obtain

$$a \bmod h = 0, \text{ i.e., } h \text{ divides } a.$$

The same argument with b provides

$$b \bmod h = 0, \text{ i.e., } h \text{ divides } b.$$

Thus,

$$h \leq \text{GCD}\,(a,b)\,.$$

(ii) We show that $h \geq \text{GCD}\,(a,b)$.
Since $\text{GCD}\,(a,b)$ divides both a and b, $\text{GCD}\,(a,b)$ must divide every linear combination $a \cdot u + b \cdot v$ for $u, v \in \mathbb{Z}$. Hence, $\text{GCD}\,(a,b)$ divides every element of $\text{Comb}\,(a,b)$. Since $h \in \text{Comb}\,(a,b)$,

$$\text{GCD}\,(a,b) \mid h, \text{ i.e., } \text{GCD}\,(a,b) \leq h.$$

□

For the study of number-theoretic problems, algebra is of enormous importance. In algebra one investigates algebraic structures called algebras. An **algebra** is a pair (S, F), where

(i) S is a set of elements.
(ii) F is a set of mappings that map arguments or tuples of arguments from S to S. More precisely, F is a set of **operations** on S, and an operation $f \in F$ is a mapping from S^m to S for a nonnegative integer m. A function $f : S^m \to S$ is called an ***m*-ary operation** on S.

This definition of an algebra does not say anything else than that we are not interested in structures that are not closed according to the operations considered (i.e., we are not interested in mappings producing results outside S for arguments from S).

In what follows, we prefer the simplified notation $(S, f_1, f_2, \dots, f_k)$ to the notation $(S, \{f_1, f_2, \dots, f_k\})$ to represent algebras. For some operations f and g, we shall use the notation $\cdot$ and $+$, respectively, if f can be considered to be a version of multiplication and g can be considered to be a version of addition. In this case, we write $x \cdot y$ instead of $f(x, y)$ and $x + y$ instead of $g(x, y)$.

In relation to number theory, we are especially interested in finite structures with a multiplication modulo n or an addition modulo n, for a positive integer n.

Definition A.2.20. *A* **group** *is an algebra $(S, *)$ for which the following is true:*

(i) $*$ is a binary (2-ary) operation.
(ii) $*$ is associative, i.e., for all $x, y, z \in S$

$$(x * y) * z = x * (y * z).$$

(iii) There exists an element $e \in S$ such that

$$e * x = x = x * e$$

for every element $x \in S$. The element e is called the **neutral element according to $*$ in S**.
(iv) For each $x \in S$, there exists an element $i(x) \in S$ such that

$$i(x) * x = e = x * i(x).$$

The element $i(x)$ is called the **neutral element of x according to $*$**.

A group is said to be **commutative** *if*

$$x * y = y * x$$

for all $x, y \in S$.

In what follows, if $*$ is considered to be multiplication, the neutral element is denoted by 1. If $*$ is considered to be addition, the neutral element is denoted by 0.

Example A.2.21. We show that $(\mathbb{Z}, +)$ is a commutative group. The structure $(\mathbb{Z}, +)$ is an algebra, since the addition of two integers is an integer. The addition is a binary operation, which is associative and commutative. The neutral element with respect to $+$ in $\mathbb{Z}$ is 0, because

$$0 + x = x = x + 0$$

for all $x \in \mathbb{Z}$. For every $x \in \mathbb{Z}$, $i(x) = -x$ is the inverse element for x, because

$$x + (-x) = 0.$$

The pair $(\mathbb{Z}, \cdot)$ is an algebra, but not a group. Though the multiplication operation $\cdot$ is associative and commutative, and 1 is the neutral element according to $\cdot$ in $\mathbb{Z}$, each element of $\mathbb{Z}$, except 1, does not have an inverse element.

Exercise A.2.22. Show that for every positive integer n, the algebraic structure $(\mathbb{Z}_n, \oplus_{\bmod n})$ is a commutative group.

Exercise A.2.23. Prove the following claims:

(i) $(\mathbb{Z}_7, \odot_{\bmod 7})$ is not a group,
(ii) $(\mathbb{Z}_7 - \{0\}, \odot_{\bmod 7})$ is a commutative group, and
(iii) $(\mathbb{Z}_{12} - \{0\}, \odot_{\bmod 12})$ is not a group.

A subset of $\mathbb{Z}_n$ of special interest in number theory is the set

$$\mathbb{Z}_n^* = \{d \in \mathbb{Z}_n \mid \mathrm{GCD}\,(d, n) = 1\}.$$

Exercise A.2.24. Show that the following algebras are groups:

(i) $(\mathbb{Z}_5^*, \odot_{\bmod 5})$,
(ii) $(\mathbb{Z}_9^*, \odot_{\bmod 9})$,
(iii) $(\mathbb{Z}_{12}^*, \odot_{\bmod 12})$.

Exercise A.2.25. Prove that each commutative group has exactly one neutral element and that, for every group $(A, *)$ and all $a, b, c \in A$, the following holds:

(i) $a = i(i(a))$,
(ii) $a * b = c * b$ implies $a = c$ and
$b * a = b * c$ implies $a = c$,
(iii) $a \neq b \Leftrightarrow a * c \neq b * c \Leftrightarrow c * a \neq c * b$.

Definition A.2.26. *Let $(S, *)$ be a group with the neutral element e. For every $a \in S$ and every $j \in \mathbb{Z}$, we define the* **j-th power of a** *as follows:*

(i) $a^0 = e$, $a^1 = a$, and $a^{-1} = i(a)$,
(ii) $a^{j+1} = a * a^j$ for all $j \geq 1$, and
(iii) $a^{-j} = (i(a))^j$ for every positive integer j.

An element g of S is called a **generator of the group** $(S, *)$ *if*

$$S = \{g^i \mid i \in \mathbb{Z}\}.$$

If a group has a generator, the group is called **cyclic**.

The group $(\mathbb{Z}_5^*, \odot_{\bmod 5})$ is cyclic. A generator of this group is the element 2, because

$$\begin{aligned}
2^1 &= 2 \text{ and } 2 \bmod 5 = 2\\
2^2 &= 4 \text{ and } 4 \bmod 5 = 4\\
2^3 &= 8 \text{ and } 8 \bmod 5 = 3\\
2^4 &= 16 \text{ and } 16 \bmod 5 = 1
\end{aligned}$$

and $\mathbb{Z}_5^* = \{1, 2, 3, 4\}$.

One can use the notion of "group" in order to provide an equivalent definition of primes.

Theorem A.2.27. *For every positive integer p, $p \geq 2$,*

$$p \text{ is a prime } \Leftrightarrow (\mathbb{Z}_p - \{0\}, \odot_{\bmod p}) \text{ is a group.}$$

Proof. We show this equivalence by proving the two corresponding implications separately.

(i) *We prove, for every prime p, that the algebra $(\mathbb{Z}_p - \{0\}, \odot_{\bmod p})$ is a group.*
Clearly, $\odot_{\bmod p}$ is an associative and commutative operation, and 1 is the neutral element according to $\odot_{\bmod p}$ in $\mathbb{Z}_p - \{0\}$. It remains to show that there exists an inverse element a^{-1} for every $a \in \mathbb{Z}_p - \{0\}$.
Let a be an arbitrary element from $\mathbb{Z}_p - \{0\} = \{1, 2, \ldots, p-1\}$. We show that one of the elements of $\{1, 2, \ldots, p-1\}$ must be inverse to a. Consider the following multiples of a:

$$m_1 = 1 \cdot a, \quad m_2 = 2 \cdot a, \ldots, \quad m_{p-1} = (p-1) \cdot a.$$

Our aim is to show that one of these multiples modulo p must be the neutral element 1. First, we prove that

$$m_i \bmod p \neq m_j \bmod p$$

for all $i \neq j$. We prove it by contradiction. Let

$$m_r \equiv m_s \pmod{p}$$

for two different $r, s \in \{1, 2, \ldots, p-1\}$, $r > s$. The prime p divides the number

$$m_r - m_s = (r - s) \cdot a.$$

Following Corollary A.2.4, the prime p must divide $(r-s)$ or a. But this is impossible, because $0 < r - s < p$ and $0 < a < p$. Hence, we have proved that the numbers

$$m_1 \bmod p, \;\; m_2 \bmod p, \ldots, \;\; m_{p-1} \bmod p$$

are pairwise different, and so

$$\{1 \odot_{\bmod p} a, 2 \odot_{\bmod p} a, \ldots, (p-1) \odot_{\bmod p} a\} = \{1, 2, \ldots, p-1\}.$$

Hence, there exists a $b \in \{1, 2, \ldots, p-1\}$ such that

$$b \odot_{\bmod p} a = 1, \text{ i.e., } a^{-1} = b.$$

Since we have proved the existence of an inverse element for every $a \in \{1, 2, \ldots, p-1\}$, we can conclude that $(\mathbb{Z}_p - \{0\}, \odot_{\bmod p})$ is a group.

(ii) *We have to prove that if $(\mathbb{Z}_p - \{0\}, \odot_{\bmod p})$ is a group, then p is a prime.* Let us do this in an indirect way. Let p be composite. Then,

$$p = b \cdot d$$

for some $b, d \in \{2, 3, \ldots, p-1\}$. Then,

$$b \odot_{\bmod p} d = 0.$$

Therefore, $(\mathbb{Z}_p - \{0\}, \odot_{\bmod p})$ is not an algebra, because the multiplication modulo p maps two elements from $\mathbb{Z}_p - \{0\}$ to an element outside of $\mathbb{Z}_p - \{0\}$.

□

The above presented proof idea (part (i)) of Theorem A.2.27 can also be applied to prove the so-called Fermat's Little Theorem. This theorem is the starting point for solving several number-theoretic problems, especially for the design of efficient randomized algorithms for primality testing.

Theorem A.2.28. Fermat's Little Theorem

For every prime p and every number $a \in \mathbb{Z}_p^* = \{d \in \mathbb{Z}_p \mid \mathrm{GCD}\,(d, p) = 1\}$,

$$a^{p-1} \bmod p = 1,$$

i.e., a^{p-2} is the inverse element of a according to $\odot_{\bmod p}$.

Proof. Let p be a prime, and let a be an arbitrary element from $\mathbb{Z}_p^*$. Consider again the following $p-1$ multiples of a:

$$m_1 = 1 \cdot a, \;\; m_2 = 2 \cdot a, \ldots, \;\; m_{p-1} = (p-1) \cdot a.$$

In the proof of Theorem A.2.27 we have already shown that the numbers

$$m_1 \bmod p,\ m_2 \bmod p,\ \ldots,\ m_{p-1} \bmod p$$

are pairwise different, and so

$$|\{m_1 \bmod p,\ m_2 \bmod p,\ \ldots,\ m_{p-1} \bmod p\}| = p-1. \tag{A.8}$$

Since $a < p$ and $r < p$ for every $r \in \{1, 2, \ldots, p-1\}$, the prime p cannot divide the number $m_r = r \cdot a$. Hence, $m_r \bmod p \neq 0$ for all $r \in \{1, 2, \ldots, p-1\}$. This observation and (A.8) together imply

$$\{m_1 \bmod p,\ m_2 \bmod p,\ \ldots,\ m_{p-1} \bmod p\} = \{1, 2, \ldots, p-1\}. \tag{A.9}$$

Now, let us consider the number

$$\prod_{i=1}^{p-1} m_i = (1 \cdot a) \cdot (2 \cdot a) \cdot \ldots \cdot ((p-1) \cdot a) = 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot a^{p-1}. \tag{A.10}$$

Then (A.9) implies the congruence

$$1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot a^{p-1} \equiv 1 \cdot 2 \cdot \ldots \cdot (p-1) \pmod{p},$$

which is equivalent to

$$1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot a^{p-1} - 1 \cdot 2 \cdot \ldots \cdot (p-1) \equiv 0 \pmod{p},$$

and so is equivalent to

$$(1 \cdot 2 \cdot \ldots \cdot (p-1)) \cdot \left(a^{p-1} - 1\right) \equiv 0 \pmod{p}. \tag{A.11}$$

Since p does not divide[5] the number $1 \cdot 2 \cdot \ldots \cdot (p-1)$, (A.11) implies[6] that p must divide $a^{p-1} - 1$, and so

$$a^{p-1} - 1 \equiv 0 \pmod{p},$$

which is equivalent to

$$a^{p-1} \equiv 1 \pmod{p}.$$

$\square$

The Fermat's Little Theorem says that it may be of interest to compute the number

$$a^{p-1} \bmod p$$

for a given positive integer p and a number $a \in \{1, 2, \ldots, p-1\}$, because

$$a^{p-1} \bmod n \neq 1 \text{ proves the fact "} p \notin \text{PRIME".}$$

[5] All factors of the number $1 \cdot 2 \cdot \ldots \cdot (p-1)$ are smaller than p.

[6] because of the Fundamental Theorem of Arithmetics (and more precisely, because of Corollary A.2.4)

For an efficient computation of $a^{p-1} \bmod p$, one is not allowed to perform $(p-2)$ multiplications by a because in this way the number of executed operations would be exponential in $\lceil \log_2 p \rceil$. If one has to compute $a^b \bmod p$ for a $b = 2^k$, then one can do this efficiently in the following way:

$$\begin{aligned}
a^2 \mod p &= a \cdot a \mod p,\\
a^4 \mod p &= (a^2 \mod p) \cdot (a^2 \mod p) \mod p,\\
a^8 \mod p &= (a^4 \mod p) \cdot (a^4 \mod p) \mod p,\\
&\vdots\\
a^{2^k} \mod p &= (a^{2^{k-1}} \mod p)^2 \mod p.
\end{aligned}$$

In general, let

$$b = \sum_{i=1}^{k} b_i \cdot 2^{i-1}$$

(i.e., $b = \text{Number}(b_k b_{k-1} \ldots b_1)$) for a $k \in \mathbb{N} - \{0\}$ and $b_i \in \{0,1\}$ for $i = 1, \ldots, k$. Then,

$$a^b = a^{b_1 \cdot 2^0} \cdot a^{b_2 \cdot 2^1} \cdot a^{b_3 \cdot 2^2} \cdot \ldots \cdot a^{b_k \cdot 2^{k-1}}.$$

To calculate $a^b \mod p$, one first computes the numbers $a_i = a^{2^{i-1}} \mod p$ for all $i = 1, 2, \ldots, k$ by repeated squaring. Then, one multiplies modulo p all numbers a_i, for which $b_i = 1$, i.e., one computes

$$\prod_{\substack{i=1\\ b_i=1}}^{k} a_i.$$

Hence, for computing $a^{p-1} \bmod p$, it is sufficient to perform $2\lceil \log_2 p \rceil$ multiplications over $\mathbb{Z}_p$, i.e., over the numbers of the binary length $\lceil \log_2 p \rceil$. If one applies the school algorithm for executing multiplications, then $O\big((\log_2 p)^3\big)$ binary operations suffice to compute $a^{p-1} \bmod p$.

The structure $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is crucial for our applications. The following two assertions provide important information about $\mathbb{Z}_n^*$.

Theorem A.2.29. *For all positive integers n, $(Z_n^*, \odot_{\bmod n})$ is a commutative group.*

Proof. First, we have to show that Z_n^* is closed according to the operation $\odot_{\bmod n}$. Let a and b be two arbitrary elements of Z_n^*, i.e., $\text{GCD}(a, n) = \text{GCD}(b, n) = 1$. Since

$$\text{Comb}(a \cdot b, n) \subseteq \text{Comb}(a, n) \cap \text{Comb}(b, n),$$

and so $1 \in \text{Comb}(a \cdot b, n)$, Theorem A.2.19 implies

$$\text{GCD}(a \cdot b, n) = 1.$$

Therefore, $a \cdot b \bmod n$ is in Z_n^*, and so $(Z_n^*, \odot_{\bmod n})$ is an algebra.

Clearly, 1 is the neutral element according to $\odot_{\bmod n}$ in Z_n^*, and the operation $\odot_{\bmod n}$ is an associative and commutative operation.

It remains to show that, for every $a \in Z_n^*$, there exists an inverse element a^{-1} with respect to $\odot_{\bmod n}$. To do this, it is sufficient to show that

$$|\{a \odot_{\bmod n} 1, a \odot_{\bmod n} 2, \ldots, a \odot_{\bmod n} (n-1)\}| = n - 1,$$

which directly implies that

$$1 \in \{a \odot_{\bmod n} 1, \ldots, a \odot_{\bmod n} (n-1)\}.$$

We prove this by contradiction. Assume that there exist two numbers $i, j \in \{1, 2, \ldots, n-1\}$, $i > j$, such that

$$a \odot_{\bmod n} i = a \odot_{\bmod n} j, \text{ i.e., } a \cdot i \equiv a \cdot j \pmod{n}.$$

This means that

$$a \cdot i = n \cdot k_1 + z \text{ and } a \cdot j = n \cdot k_2 + z$$

for suitable numbers k_1, k_2 and $z < n$. Then

$$a \cdot i - a \cdot j = n \cdot k_1 - n \cdot k_2 = n \cdot (k_1 - k_2),$$

and so

$$a \cdot (i - j) = n \cdot (k_1 - k_2), \text{ i.e., } n \text{ divides } a \cdot (i - j).$$

Since $\mathrm{GCD}\,(a, n) = 1$, the number n divides $i - j$. But this is not possible because $i - j < n$.

Hence, every element $a \in \mathbb{Z}_n^*$ has a multiplicative inverse a^{-1} according to $\odot_{\bmod n}$, and so $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is a group. Since $\odot_{\bmod n}$ is commutative, $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is a commutative group. □

The next theorem shows that $\mathbb{Z}_n^*$ contains exactly those elements of $\mathbb{Z}_n$ that possess inverse elements according to $\odot_{\bmod n}$.

Theorem A.2.30. *For all positive integers n,*

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \textit{there exists an } a^{-1} \in \mathbb{Z}_n, \textit{ such that } a \odot_{\bmod n} a^{-1} = 1\}.$$

Proof. The relation $Z_n^* \subseteq \{a \in \mathbb{Z}_n \mid \exists\, a^{-1} \in \mathbb{Z}_n\}$ is a direct consequence of the fact that $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is a group.

It remains to show that

$$\{a \in \mathbb{Z}_n \mid \exists\, a^{-1} \in \mathbb{Z}_n\} \subseteq Z_n^* = \{a \in \mathbb{Z}_n \mid \mathrm{GCD}\,(a, n) = 1\}.$$

This means that we have to show that the existence of a^{-1} for an $a \in \mathbb{Z}_n$ implies that $\mathrm{GCD}\,(a, n) = 1$.

Let a be an arbitrary element from $\mathbb{Z}_n$ such that $a^{-1} \in \mathbb{Z}_n$. Since $a \cdot a^{-1} \equiv 1 \pmod{n}$, we have

$$a \cdot a^{-1} = k \cdot n + 1 \tag{A.12}$$

for a natural number k. We consider the following linear combination of a and n from $\text{Comb}\,(a, n)$:

$$a \cdot x + n \cdot y = a \cdot a^{-1} + n \cdot (-k) \underset{(A.12)}{=} k \cdot n + 1 - k \cdot n = 1$$

for $x = a^{-1}$ and $y = -k$. Consequently, 1 is in $\text{Comb}\,(a, n)$, and so Theorem A.2.19 implies

$$\text{GCD}\,(a, n) = \min\{d \in \text{Comb}\,(a, n) \mid d \geq 1\} = 1.$$

□

The following important discovery about $\mathbb{Z}_n^*$ is given without proof, which is too long and technical and not needed for understanding of the design and analysis of randomized algorithms.

Theorem A.2.31. *Let n be a positive integer. The group $(\mathbb{Z}_n^*, \odot_{\text{mod } n})$ is cyclic if and only if*

$$n \in \{2, 4, p^k, 2 \cdot p^k \mid p \in \text{PRIME} - \{2\}, k \in \mathbb{N} - \{0\}\}.$$

□

An important instrument of number theory is the Chinese Remainder Theorem. This theorem is very useful for the study of $\mathbb{Z}_n$ for $n \notin \text{PRIME}$, because it provides a well structured representation of the elements of $\mathbb{Z}_n$. This representation is in many cases more transparent and more convenient than the standard representation of $\mathbb{Z}_n$, and so it enables the construction of $\mathbb{Z}_n$ in a transparent way. Let $m = m_1 \cdot m_2 \cdot \ldots m_k$, where $\text{GCD}\,(m_i, m_j) = 1$ for all $i \neq j$. Then, the first simple version of the Chinese Remainder Theorem says that the function

$$F(m) = (m \bmod m_1, m \bmod m_2, \ldots, m \bmod m_k)$$

from $\mathbb{Z}_m$ to $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_k}$ is a bijection, and so $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_k}$ is another representation of $\mathbb{Z}_n$.

Theorem A.2.32. Chinese Remainder Theorem, first version
Let

$$m = m_1 \cdot m_2 \cdot \ldots \cdot m_k,$$

where $k \in \mathbb{N} - \{0\}$, $m_i \in \mathbb{N} - \{0, 1\}$ for $i = 1, 2, \ldots, k$, and

$$\text{GCD}\,(m_i, m_j) = 1$$

for all $i, j \in \{1, 2, \ldots, k\}$, $i \neq j$. Then, for each sequence of k numbers

$$r_1 \in \mathbb{Z}_{m_1}, r_2 \in \mathbb{Z}_{m_2}, \ldots, r_k \in \mathbb{Z}_{m_k},$$

(i.e., for every element $(r_1, r_2, \ldots, r_k) \in \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_k}$), there is a unique $r \in \mathbb{Z}_m$ such that

$$r \equiv r_i \pmod{m_i}$$

for $i = 1, 2, \ldots, k$.

Proof. First, we show that there exists at least one r with this property.

Since GCD $(m_i, m_j) = 1$ for all distinct i and j from $\{1, 2, \ldots, k\}$, we have

$$\text{GCD}\left(\frac{m}{m_l}, m_l\right) = 1 \text{ for all } l \in \{1, 2, \ldots, k\}.$$

Hence, following Theorem A.2.30, the element m/m_i has an inverse element n_i with respect to $\odot_{\bmod m_i}$ in $\mathbb{Z}_{m_i}$. We consider the number

$$e_i = n_i \cdot \frac{m}{m_i} = m_1 \cdot m_2 \cdot \ldots \cdot m_{i-1} \cdot n_i \cdot m_{i+1} \cdot \ldots \cdot m_k.$$

Since $m/m_i \equiv 0 \pmod{m_j}$ for all $j \in \{1, \ldots, k\} - \{i\}$, we have

$$e_i \equiv 0 \pmod{m_j}$$

for all is different from j.

If n_i is the inverse element of m/m_i in $\mathbb{Z}_{m_i}$, then

$$e_i \bmod m_i = n_i \cdot \frac{m}{m_i} \bmod m_i = 1.$$

If, for each $(r_1, r_2, \ldots, r_k) \in \mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \ldots \times \mathbb{Z}_{m_k}$, the number r is chosen as

$$r \equiv \left(\sum_{i=1}^{k} r_i \cdot e_i\right) \pmod{m},$$

then, clearly,

$$r \equiv r_i \pmod{m_i} \tag{A.13}$$

for all $i \in \{1, 2, \ldots, k\}$.

It remains to show that there exists at most one $r \in \mathbb{Z}_m$ that fulfills all k congruences (A.13).

Assume that there are two numbers x and y from $\mathbb{Z}_n$ such that

$$y \equiv x \equiv r_i \pmod{m_i}$$

for all $i \in \{1, 2, \ldots, k\}$. Then,

$$x - y \equiv 0 \pmod{m_i}, \text{ i.e., } m_i \mid (x - y)$$

for all $i \in \{1, 2, \ldots, k\}$. Since $m = m_1 \cdot m_2 \cdot \ldots \cdot m_k$ and $\mathrm{GCD}\,(m_i, m_j) = 1$ for all $i \neq j$, we obtain that

$$m \text{ divides } x - y.$$

Since $x - y < m$, we have $x - y = 0$, and so $x = y$. □

The second version of the Chinese Remainder Theorem does not provide only a bijection between Z_m and $\mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_k}$, but it additionally makes it possible to simulate the operations in Z_m by some operations in $\mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_k}$. If such a simulation is possible, we speak about an isomorphism between the algebraic structures considered.

Definition A.2.33. *Let $\mathcal{A} = (A, f_1, \ldots, f_k)$ and $\mathcal{B} = (B, g_1, \ldots, g_k)$ be two algebras, where f_i and g_i are d_i-ary operations, $d_i \in \mathbb{N}$ for $i = 1, \ldots, k$. We say that $\mathcal{A}$ and $\mathcal{B}$ are isomorphic if there exists a bijection $H : A \to B$ such that, for all $a_1, a_2, \ldots, a_{d_i} \in A$,*

$$H(f_i(a_1, a_2, \ldots, a_{d_i})) = g_i(H(a_1), H(a_2), \ldots, H(a_{d_i}))$$

for all $i \in \{1, 2, \ldots, k\}$.

Since, for all applications in this textbook, it is sufficient to consider the isomorphism between $\mathbb{Z}_n$ and $\mathbb{Z}_p \times \mathbb{Z}_q$ for some $n = p \cdot q$, $\mathrm{GCD}\,(p, q) = 1$, we present a second, simplified version of the Chinese Remainder Theorem.

Theorem A.2.34. Chinese Remainder Theorem, second version
Let $n = p \cdot q$ and $\mathrm{GCD}\,(p, q) = 1$. Let $\oplus_{p,q}$ and $\odot_{p,q}$ be two operations on $\mathbb{Z}_p \times \mathbb{Z}_q$, defined by

$$(a_1, a_2) \oplus_{p,q} (b_1, b_2) = ((a_1 + b_1) \bmod p, (a_2 + b_2) \bmod q)$$

and

$$(a_1, a_2) \odot_{p,q} (b_1, b_2) = ((a_1 \cdot b_1) \bmod p, (a_2 \cdot b_2) \bmod q)$$

for all $(a_1, a_2), (b_1, b_2) \in \mathbb{Z}_p \times \mathbb{Z}_q$.

Then, the algebras $(\mathbb{Z}_n, \oplus_{\bmod m}, \odot_{\bmod m})$ and $(\mathbb{Z}_p \times \mathbb{Z}_q, \oplus_{p,q}, \odot_{p,q})$ are isomorphic.

Proof. In the first version of the Chinese Remainder Theorem, we have already proved that the mapping $h : \mathbb{Z}_n \to \mathbb{Z}_p \times \mathbb{Z}_q$ defined by

$$h(a) = (a \bmod p, a \bmod q)$$

is a bijection.

It remains to show that

$$h(a \oplus_{\bmod n} b) = h(a) \oplus_{p,q} h(b) \text{ and } h(a \odot_{\bmod n} b) = h(a) \odot_{p,q} h(b)$$

for all $a, b \in \mathbb{Z}_n$. All $a, b \in \mathbb{Z}_n$ can be expressed as follows:

$$a = a_1' \cdot p + a_1 = a_2{}' \cdot q + a_2$$

$$b = b_1{}' \cdot p + b_1 = b_2{}' \cdot q + b_2$$

for suitable $a_1{}', a_1, a_2{}', a_2, b_1{}', b_1, b_2{}', b_2$ with $a_1 < p$, $a_2 < q$, $b_1 < p$ and $b_2 < q$.

Therefore $h(a) = (a_1, a_2)$ and $h(b) = (b_1, b_2)$. Now, for all $a, b \in \mathbb{Z}_n$, we have

$$\begin{aligned} h(a \oplus_{\bmod n} b) &= h((a+b) \bmod n) \\ &= ((a+b) \bmod p, (a+b) \bmod q) \\ &= ((a_1+b_1) \bmod p, (a_2+b_2) \bmod q) \\ &= (a_1, a_2) \oplus_{p,q} (b_1, b_2) \\ &= h(a) \oplus_{p,q} h(b) \end{aligned}$$

and

$$\begin{aligned} h(a \odot_{\bmod n} b) &= h((a \cdot b) \bmod n) \\ &= ((a \cdot b) \bmod p, (a \cdot b) \bmod q) \\ &= ((a_1{}' \cdot b_1{}' \cdot p^2 + (a_1 \cdot b_1{}' + a_1{}' \cdot b_1) \cdot p + a_1 \cdot b_1) \bmod p, \\ &\quad\ (a_2{}' \cdot b_2{}' \cdot q^2 + (a_2{}' \cdot b_2 + a_2 \cdot b_2{}') \cdot q + a_2 \cdot b_2) \bmod q) \\ &= ((a_1 \cdot b_1) \bmod p, (a_2 \cdot b_2) \bmod p) \\ &= (a_1, a_2) \odot_{p,q} (b_1, b_2) \\ &= h(a) \odot_{p,q} h(b). \end{aligned}$$

□

Exercise A.2.35. Formulate and prove the general second version of the Chinese Remainder Theorem for $m = m_1 \cdot m_2 \cdot \ldots \cdot m_k$ with $\mathrm{GCD}\,(m_i, m_j) = 1$ for all $i, j \in \{1, \ldots, k\}$ with $i \neq j$.

The last important and frequently used instrument we would like to introduce here is Lagrange's Theorem, which says that the cardinality of any subgroup of a finite group divides the cardinality of the group. To prove this claim, we first need some new notions and a few technical lemmas.

Definition A.2.36. *Let $(A, *)$ be a group with a neutral element 1. For each $a \in A$, the* **order of a** *is defined by*

$$\mathbf{order}\,(\boldsymbol{a}) = \min\{r \in \mathbb{N} - \{0\} \mid a^r = 1\}$$

if there exists at least one r with $a^r = 1$.

If $a^i \neq 1$ for all $i \in \mathbb{N} - \{0\}$, then we set $\mathrm{order}\,(a) = \infty$.

Lemma A.2.37. *Let $(A, *)$ be a finite group. Then, every element $a \in A$ is of a finite order (more precisely,* $\text{order}(a) \in \{1, 2, \ldots, |A|\}$*).*

Proof. Let a be an arbitrary element of A. Consider the $|A| + 1$ elements

$$a^0, a^1, \ldots, a^{|A|}$$

from A. Clearly, there exist $i, j \in \{1, \ldots, |A| + 1\}$, $0 \leq i < j \leq |A|$, such that $0 \leq i < j \leq |A|$ and

$$a^i = a^j.$$

Consequently,

$$1 = a^i \cdot \left(a^{-1}\right)^i = a^j \cdot \left(a^{-1}\right)^i = a^{j-i},$$

and so

$$\text{order}(a) \leq j - i \in \{1, 2, \ldots, |A|\}.$$

□

Definition A.2.38. *Let $(A, *)$ be a group. An algebra $(H, *)$ is a* **subgroup** *of $(A, *)$ if*

(i) $H \subseteq A$, and
(ii) $(H, *)$ is a group.

For instance, $(\mathbb{Z}, +)$ is a subgroup of $(\mathbb{Q}, +)$ and $(\{1\}, \odot_{\bmod 5})$ is a subgroup of $(\mathbb{Z}_5^*, \odot_{\bmod 5})$.

Lemma A.2.39. *Let $(H, *)$ be a subgroup of a group $(A, *)$. Then, the neutral elements of both groups are the same.*

Proof. Let e_H be the neutral element of $(H, *)$ and let e_A be the neutral element of $(A, *)$. Since e_H is the neutral element of $(H, *)$,

$$e_H * e_H = e_H. \tag{A.14}$$

Since e_A is the neutral element of $(A, *)$ and $e_H \in A$, we have

$$e_A * e_H = e_H. \tag{A.15}$$

Since the right sides of the equations (A.14) and (A.15) are the same, we obtain

$$e_H * e_H = e_A * e_H. \tag{A.16}$$

Let e_H^{-1} be the inverse element of e_H according to $*$ in A. Multiplying (A.16) with e_H^{-1} from the right side, we finally obtain

$$e_H = e_H * e_H * e_H^{-1} \underset{(A.16)}{=} e_A * e_H * e_H^{-1} = e_A.$$

□

The next theorem says that, for proving $(H, \circ)$ is a subgroup of a finite group $(A, \circ)$ for an $H \subseteq A$, it suffices to show that H is closed under $\circ$.

Theorem A.2.40. *Let $(A, \circ)$ be a finite group. Every algebra $(H, \circ)$ with $H \subseteq A$ is a subgroup of $(A, \circ)$.*

Proof. Let $H \subseteq A$ and let $(H, \circ)$ be an algebra. To prove that $(H, \circ)$ is a subgroup of $(A, \circ)$, it is sufficient to show that $(H, \circ)$ is a group. We do this by showing that e_A is the neutral element of $(H, \circ)$, and that each $b \in H$ possesses an inverse element b^{-1} in H.

Let b be an arbitrary element of H. Since $b \in A$ and A is finite, order $(b) \in \mathbb{N} - \{0\}$ and

$$b^{\text{order}(b)} = e_A.$$

Since H is closed under $\circ$, the element b^i is in H for every positive integer i, and so

$$e_A = b^{\text{order}(b)} \in H,$$

too. Since

$$e_A \circ d = d$$

for every $d \in A$ and $H \subseteq A$, the element e_A is also the neutral element of $(H, \circ)$.

Since $b^{\text{order}(b)-1} \in H$ for every $b \in H$ and

$$e_A = b^{\text{order}(b)} = b \circ b^{\text{order}(b)-1},$$

$b^{\text{order}(b)-1}$ is the inverse element of b in $(H, \circ)$. □

Note that the assumption of Theorem A.2.40 that A is finite is essential because the assertion does not need to be true for infinite groups. For instance, $(\mathbb{N}, +)$ is an algebra and $\mathbb{N} \subseteq \mathbb{Z}$, but $(\mathbb{N}, +)$ is not a group.

Exercise A.2.41. Let $(H, \circ)$ and $(G, \circ)$ be subgroups of a group $(A, \circ)$. Prove, that $(H \cap G, \circ)$ is a subgroup of $(A, \circ)$, too.

Lemma A.2.42. *Let $(A, \circ)$ be a group with a neutral element e. Let, for every $a \in A$ of a finite order,*

$$H(a) = \{e, a, a^2, \ldots, a^{\text{order}(a)-1}\}.$$

The structure $(H(a), \circ)$ is the smallest subgroup of $(A, \circ)$ that contains the element a.

Proof. First, we prove that $H(a)$ is closed under $\circ$. Let a^i and a^j be two arbitrary elements of $H(a)$.

If $i + j <$ order (a) then, clearly,

$$a^i \circ a^j = a^{i+j} \in H(a).$$

If $i + j > \text{order}(a)$, then

$$\begin{aligned} a^i \circ a^j &= a^{i+j} = a^{\text{order}(a)} \circ a^{i+j-\text{order}(a)} \\ &= e \circ a^{i+j-\text{order}(a)} = a^{i+j-\text{order}(a)} \in H(a). \end{aligned}$$

Following the definition of $H(a)$, $e \in H(a)$. For every element $a^i \in H(a)$,

$$e = a^{\text{order}(a)} = a^i \circ a^{\text{order}(a)-i},$$

and so $a^{\text{order}(a)-i}$ is the inverse element of a^i. Hence, $(H(a), \circ)$ is a group.

Since every algebra $(G, \circ)$ with $a \in G$ containing a must[7] contain the whole set $H(a)$, the group $(H(a), \circ)$ is the smallest subgroup of $(A, \circ)$ that contains a. □

Definition A.2.43. *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. For every $b \in A$, we define the sets*

$$\boldsymbol{H} \circ \boldsymbol{b} = \{h \circ b \mid h \in H\} \text{ and } \boldsymbol{b} \circ \boldsymbol{H} = \{b \circ h \mid h \in H\}.$$

The set $H \circ b$ is called the **right coset** *of H in $(A, \circ)$, and $b \circ H$ is called the* **left coset** *of H in $(A, \circ)$. If $H \circ b = b \circ H$, then $H \circ b$ is called a* **coset** *of H in $(A, \circ)$.*

For instance, $B_7 = \{7 \cdot a \mid a \in \mathbb{Z}\}$ is a subgroup of $(\mathbb{Z}, +)$. Then, for $i = 0, 1, \ldots, 6$,

$$B_7 + i = i + B_7 = \{7 \cdot a + i \mid a \in \mathbb{Z}\} = \{b \in \mathbb{Z} \mid b \bmod 7 = i\}$$

are cosets of B_7 in $(\mathbb{Z}, +)$. Observe that the class $\{B_7 + i \mid i \in \{0, 1, \ldots, 6\}\}$ is a partition of $\mathbb{Z}$ in seven disjoint classes.

Observation A.2.44. If $(H, \circ)$ is a subgroup of a commutative group $(A, \circ)$, then all right (left) cosets of H in $(A, \circ)$ are cosets.

Another important fact about the cosets of a finite H is that their cardinality is equal to the cardinality of H.

Theorem A.2.45. *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. Then,*

(i) $H \circ h = H$ for every $h \in H$.
(ii) For all $b, c \in A$,

$$\text{either } H \circ b = H \circ c \text{ or } H \circ b \cap H \circ c = \emptyset.$$

(iii) If H is finite, then

$$|H \circ b| = |H|$$

for all $b \in A$.

[7] Since G is closed under $\circ$.

Proof. We prove these three assertions separately in the given order. Let e be the neutral element of $(A, \circ)$ and $(H, \circ)$.

(i) Let $h \in H$. Since H is closed under $\circ$, $a \circ h \in H$, and so

$$H \circ h \subseteq H.$$

Since $(H, \circ)$ is a group, h^{-1} is in H. Let b be an arbitrary element of H. Then,

$$b = b \circ e = b \circ \underbrace{(h^{-1} \circ h)}_{e} = \underbrace{(b \circ h^{-1})}_{\in H} \circ h \in H \circ h,$$

and so

$$H \subseteq H \circ h.$$

Hence, $H \circ h = H$.

(ii) Assume $H \circ b \cap H \circ c \neq \emptyset$ for some $b, c \in A$. Then there exist $a_1, a_2 \in H$, such that

$$a_1 \circ b = a_2 \circ c.$$

This implies

$$c = a_2^{-1} \circ a_1 \circ b,$$

where $a_2^{-1} \in H$. Therefore,

$$H \circ c = H \circ (a_2^{-1} \circ a_1 \circ b) = H \circ (a_2^{-1} \circ a_1) \circ b. \tag{A.17}$$

Since a_2^{-1} and a_1 belong to H, the element $a_2^{-1} \circ a_1$ also belongs to H. Due to the claim (i), we obtain

$$H \circ (a_2^{-1} \circ a_1) = H. \tag{A.18}$$

Inserting (A.18) into (A.17), we finally obtain

$$H \circ c = H \circ (a_2^{-1} \circ a_1) \circ b = H \circ b.$$

(iii) Let H be finite, and let $b \in A$. Since $H \circ b = \{h \circ b \mid h \in H\}$, we immediately have

$$|H \circ b| \leq |H|.$$

Let $H = \{h_1, h_2, \ldots, h_k\}$ for a $k \in \mathbb{N}$. We have to show that

$$|\{h_1 \circ b, h_2 \circ b, \ldots, h_k \circ b\}| \geq k.$$

We prove this by contradiction. Let

$$h_i \circ b = h_j \circ b \tag{A.19}$$

for some $i, j \in \{1, 2, \ldots, k\}$, $i \neq j$. Since $(A, \circ)$ is a group, $b^{-1} \in A$. Applying (A.19), one obtains

$$h_i = h_i \circ e = h_i \circ (b \circ b^{-1}) = (h_i \circ b) \circ b^{-1} \underset{(A.19)}{=} (h_j \circ b) \circ b^{-1} = h_j \cdot (b \circ b^{-1}) = h_j,$$

which contradicts to the assumption $h_i \neq h_j$. Thus,

$$|H \circ b| = |H|.$$

□

A direct consequence of Theorem A.2.45 is that one can partition the set A of every group $(A, \circ)$ having a proper subgroup $(H, \circ)$ into pairwise disjoint subsets of A, which are the left (right) cosets of H in $(A, \circ)$.

Theorem A.2.46. *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. Then, the class $\{H \circ b \mid b \in A\}$ is a partition of A.*

Proof. The claim (ii) of Theorem A.2.45 shows that $H \circ b \cap H \circ c = \emptyset$ or $H \circ b = H \circ c$. So, it remains to show that

$$A \subseteq \bigcup_{b \in A} H \circ b.$$

But this is obvious because the identity e of $(A, \circ)$ is also the identity of $(H, \circ)$, and so

$$b = e \circ b \in H \circ b$$

for every $b \in A$. □

Definition A.2.47. *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. We define the* **index of H in $(A, \circ)$** *by*

$$\mathbf{Index}_{\boldsymbol{H}}(\boldsymbol{A}) = |\{H \circ b \mid b \in A\}|,$$

i.e., as the number of different right cosets of H in $(A, \circ)$.

The aim of the last part of this section is to present the following theorem, which provides a powerful instrument for proving that there are not too many "bad" elements[8] (or elements with a special property) in A.

Theorem A.2.48. Lagrange's Theorem

For every subgroup $(H, \circ)$ of a finite group $(A, \circ)$,

$$|A| = \mathrm{Index}_H(A) \cdot |H|,$$

i.e., $|H|$ divides $|A|$.

[8]Bad elements may, for instance, be non-witnesses among the candidates in an application of the method of abundance of witnesses.

Proof. Theorem A.2.46 implies that A can be partitioned into $\text{Index}_H(A)$ right cosets, and Theorem A.2.45 says that all right cosets are of the same cardinality. $\square$

Corollary A.2.49. *Let $(H, \circ)$ be a proper algebra of a finite group $(A, \circ)$. Then,*

$$|H| \leq |A|/2.$$

Proof. Theorem A.2.40 guarantees that every algebra $(H, \circ)$ of a finite group $(A, \circ)$ is a subgroup of $(A, \circ)$. Thus, for $(H, \circ)$, Lagrange's Theorem holds. Since $H \subset A$,[9] $\text{Index}_H(A) \geq 2$ and so

$$|H| \leq |A|/2.$$

$\square$

Corollary A.2.49 is the frequently used argument in the design of randomized algorithms by the method of abundance of witnesses. In order to show that the number on non-witnesses in the set of candidates is limited,[10] it is sufficient to show that

(i) all non-witnesses are elements of an algebra $(H, \circ)$,
(ii) the algebra $(H, \circ)$ is a subgroup of a group $(A, \circ)$, where A is the subset of the set of witness candidates, and
(iii) there exists an element $a \in A - H$, i.e., $H \subset A$.

A.3 Combinatorics

The aim of this section is to introduce a few fundamental concepts of combinatorics, focusing on those most frequently applied in the analysis of randomized algorithms. Here, we often do so without proofs and list the most important combinatorial equations. The starting point is to have a set of objects distinguishable from each other.

Definition A.3.50. *Let n be a positive integer. Let $S = \{a_1, a_2, \ldots, a_n\}$ be a set of n objects (elements). A permutation of n objects $a_1, a_2, \ldots, a_n$ is an ordered arrangement of the objects of S.*

A permutation can be viewed as an injective mapping π from S to S. This way, one can represent a permutation as the following vector:

$$(\pi(a_1), \pi(a_2), \ldots, \pi(a_n)).$$

For instance, if $S = \{a_1, a_2, a_3\}$, then there are the following six different ways to arrange the three objects a_1, a_2, a_3:

[9] i.e., $|H| < |A|$
[10] at most half the candidates.

$$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_1, a_2), (a_3, a_2, a_1).$$

To denote permutations, the simplified notation

$$(i_1, i_2, \ldots, i_n)$$

is often used instead of $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$.

Lemma A.3.51. *For every positive integer n, the number* $\boldsymbol{n!}$ *of different permutations of n objects is*

$$\mathbf{n}! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = \prod_{i=1}^{n} i.$$

Proof. One can generate all permutations in the following way. The first object in a permutation may be chosen in n different ways (i.e., from the n different objects). Once the first object has been chosen, the second object may be chosen in $n-1$ different ways (i.e., from the $n-1$ remaining objects), and so on. □

By arrangement, we set $0! = 1$.

Definition A.3.52. *Let k and n be non-negative integers, $k \leq n$. A* **combination of $\boldsymbol{k}$ objects from $\boldsymbol{n}$ objects** *is a selection of k objects without regard to the order (i.e., every subset of A of k elements is a combination).*

For example, a combination of four objects of the set $\{a_1, a_2, a_3, a_4, a_5\}$ is any of the following sets:

$$\{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_5\}, \{a_1, a_2, a_4, a_5\}, \{a_1, a_3, a_4, a_5\}, \{a_2, a_3, a_4, a_5\}.$$

Lemma A.3.53. *Let n and k be non-negative integers, $k \leq n$. The number* $\binom{\boldsymbol{n}}{\boldsymbol{k}}$ *of combinations of k objects from n objects is*

$$\binom{\boldsymbol{n}}{\boldsymbol{k}} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1)}{k!} = \frac{n!}{k! \cdot (n-k)!}.$$

Proof. As in the proof of Lemma A.3.51, we have n possibilities for the choice of the first element, $n-1$ ways of choosing the second element, and so on. Hence, there are

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1)$$

ways of choosing k objects from n objects when order is taken into account. But any order of the same k elements provides the same set of k elements. Hence,

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1)}{k!}.$$

□

We observe that

$$\binom{n}{0} = \binom{n}{n} = 1.$$

A direct consequence of

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

is that

$$\binom{n}{k} = \binom{n}{n-k}.$$

Exercise A.3.54. Prove the following equation. For all non-negative integers k and n, $k \leq n$,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

The values $\binom{n}{k}$ are also well known as the **binomial coefficients** from the following theorem.

Theorem A.3.55. Newton's Theorem[11]
For every positive integer n and every real number x,

$$\begin{aligned}(1+x)^n &= \binom{n}{0} + \binom{n}{1} \cdot x + \binom{n}{2} \cdot x^2 + \ldots + \binom{n}{n-1} \cdot x^{n-1} + \binom{n}{n} \cdot x^n \\ &= \sum_{i=0}^{n} \binom{n}{i} \cdot x^i.\end{aligned}$$

Exercise A.3.56. Prove Newton's Theorem.

Lemma A.3.57. *For every positive integer n,*

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Proof. To prove Lemma A.3.55, it is sufficient to set $x = 1$ in Newton's Theorem.

Another combinatorial argument is that, for each $k \in \{0, 1, \ldots, n\}$, $\binom{n}{k}$ is the number of all k-element subsets of an n-element set. Thus,

$$\sum_{k=0}^{n} \binom{n}{k}$$

counts the number of all subsets of a set of n elements. On the other hand, every set of n elements has exactly 2^n different subsets. Hence, both sides of the equation correspond to the number of subsets of an n-element set. □

[11] a special case of the well known **Binomial Theorem**.

Lemma A.3.58. *For any positive integer n*

$$\binom{n}{2} = \prod_{l=3}^{n} \frac{l}{l-2}.$$

Proof. Since $\binom{n}{2} = \binom{n}{n-2}$, one directly obtains

$$\binom{n}{2} = \frac{n \cdot (n-1) \cdot \ldots \cdot 4 \cdot 3}{(n-2) \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1} = \prod_{i=1}^{n-2} \frac{i+2}{i} = \prod_{l=3}^{n} \frac{l}{l-2}.$$

□

We use $\boldsymbol{e}$ to denote

$$\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n.$$

Note that $e \approx 2.7182\ldots$ is the base of the natural logarithm and that one can approximate e with arbitrary precision by applying the formula

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

The following lemmas present the most frequently used inequalities here involving e.

Lemma A.3.59. *For all $t, n \in \mathbb{R}^+$,*

(i) $\left(1 + \frac{t}{n}\right)^n \leq e^t$,
(ii) $\left(1 + \frac{1}{n}\right)^{t \cdot n} \leq e^t$.

Lemma A.3.60. *For every $n \in \mathbb{R}^+$, $n \geq 1$,*

$$\left(1 - \frac{1}{n}\right)^n < \frac{1}{e}.$$

Exercise A.3.61. Prove the assertions of Lemma A.3.59 and Lemma A.3.60.

The following result is often applied when working with the natural logarithm.

Lemma A.3.62. *For every $x \in (0, 1)$,*

$$\frac{-x}{1-x} \leq \ln(1-x) \leq -x.$$

Proof. The natural logarithm is a concave function. Therefore, the gradient (difference quotient) of the transversal (secant line) going through the points $1-x$ and 1 is upper bounded by the tangent gradient at the point $1-x$ and lower bounded by the tangent gradient at the point 1. Hence,

$$1 \leq \frac{\ln(1) - \ln(1-x)}{1-(1-x)} = \frac{-\ln(1-x)}{x} \leq \frac{1}{1-x}. \tag{A.20}$$

Multiplying (A.20) by $-x$, one obtains the assertion of the lemma. □

A useful instrument for working with permutations and combinations is the following **Stirling formula**:

$$n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12 \cdot n} + O\left(\frac{1}{n^2}\right)\right).$$

One can apply the Stirling formula in order to prove the following inequalities.

Lemma A.3.63. *For all $n, k \in \mathbb{N}$, $n \geq k \geq 0$,*

(i) $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k$,
(ii) $\binom{n}{k} \sim \frac{n^k}{k!}$ *for sufficiently large n.*

Exercise A.3.64. Prove Lemma A.3.63.

Exercise A.3.65. Prove that

$$\binom{n}{k} \leq \frac{n^k}{k!}$$

for all non-negative integers n and k, $k \leq n$.

Exercise A.3.66. Let n be an odd positive integer. Apply the Stirling formula in order to approximate $\binom{n}{n/2}$.

For every positive integer n, the ***n*-th Harmonic number Har (n)** is defined by the following series

$$\text{Har}\,(n) = \sum_{i=1}^{n} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$$

First, we observe that Har (n) tends to infinity with increasing n. The simplest way to see this is to partition the terms of Har (n) into infinitely many groups of cardinality 2^k for $k = 1, 2, \ldots$.

$$\underbrace{\frac{1}{1}}_{\text{Group 1}} + \underbrace{\frac{1}{2} + \frac{1}{2}}_{\text{Group 2}} + \underbrace{\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}}_{\text{Group 3}} +$$

$$\underbrace{\frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \frac{1}{11} + \frac{1}{12} + \frac{1}{13} + \frac{1}{14} + \frac{1}{15}}_{\text{Group 4}} + \ldots$$

Both terms in group 2 are between $1/4$ and $1/2$, and so the sum of that group is between $2 \cdot (1/4) = 1/2$ and $2 \cdot (1/2) = 1$. All four terms in group 3 are between $4 \cdot (1/8) = 1/2$ and $4 \cdot (1/4) = 1$. In general, for every positive integer k, all 2^{k+1} terms of group k are between 2^{-k} and 2^{-k+1}, and hence the sum of the terms of group k is between

$$\frac{1}{2} = 2^{k-1} \cdot 2^{-k} \text{ and } 1 = 2^{k-1} \cdot 2^{-k+1}.$$

This grouping procedure shows us that if n is in group k, then

$$\mathrm{Har}\,(n) \geq \frac{k}{2} \text{ and } \mathrm{Har}\,(n) \leq k.$$

Consequently,

$$\frac{\lfloor \log_2 n \rfloor}{2} + \frac{1}{2} < \mathrm{Har}\,(n) \leq \lfloor \log_2 n \rfloor + 1$$

for all positive integers n.

The following exercise provides a more precise approximation of Har (n).

Exercise A.3.67.* Prove that

$$\mathrm{Har}\,(n) = \ln n + O(1)$$

for every positive integer n.

Analyzing the complexity of algorithms (especially of those designed by the "divide and conquer" method) one often reduces the analysis to solving a specific recurrence. The typical recurrences are of the form

$$T(n) = a \cdot T\left(\frac{n}{c}\right) + f(n),$$

where a and c are positive integers and f is a function from $\mathbb{N}$ to $\mathbb{R}^+$. In what follows we provide a general solution of this recurrence if $f(n) \in \Theta(n)$.

Theorem A.3.68. Master Theorem
Let a, b, and c be positive integers. Let

$$T(1) = 0,$$
$$T(n) = a \cdot T\left(\frac{n}{c}\right) + b \cdot n.$$

Then,

$$T(n) = \begin{cases} O(n) & \text{if } a < c, \\ O(n \cdot \log n) & \text{if } a = c, \\ O\left(n^{\log_c n}\right) & \text{if } c < a. \end{cases}$$

Proof. For simplicity, we assume $n = c^k$ for some positive integer k. A multiple application of the recurrence provides the following expression for $T(n)$:

$$\begin{aligned} T(n) &= a \cdot T\left(\frac{n}{c}\right) + b \cdot n \\ &= a \cdot \left[a \cdot T\left(\frac{n}{c^2}\right) + b \cdot \frac{n}{c}\right] + b \cdot n \\ &= a^2 \cdot T\left(\frac{n}{c^2}\right) + b \cdot n \cdot \left(\frac{a}{c} + 1\right) \end{aligned}$$

$$= a^k \cdot T(1) + b \cdot n \cdot \sum_{i=0}^{k-1} \left(\frac{a}{c}\right)^i$$
$$= b \cdot n \cdot \left(\sum_{i=0}^{(\log_c n)-1} \left(\frac{a}{c}\right)^i \right).$$

Now we distinguish the three cases according to the relation between a and c.

(i) Let $a < c$. Then,

$$\sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i \leq \sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i = \frac{1}{1-\frac{a}{c}} \in O(1).$$

Hence, $T(n) \in O(n)$.

(ii) Let $a = c$. Obviously,

$$\sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i = \sum_{i=0}^{\log_c n-1} 1 = \log_c n \in O(\log n).$$

So, we obtain $T(n) \in O(n \cdot \log n)$.

(iii) Let $a > c$. Then,

$$\begin{aligned} T(n) &= b \cdot n \cdot \sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i \\ &= b \cdot n \cdot \left(\frac{1-(\frac{a}{c})^{\log_c n}}{1-\frac{a}{c}} \right) \\ &= \frac{b}{\frac{a}{c}-1} \cdot n \cdot \left(\left(\frac{a}{c}\right)^{\log_c n} - 1 \right) \\ &= \frac{b}{\frac{a}{c}-1} \cdot n \cdot \left(\frac{a^{\log_c n}}{n} - 1 \right) \\ &\in O\big(a^{\log_c n}\big) \\ &= O\big(n^{\log_c a}\big). \end{aligned}$$

□

We close this section with a combinatorial lemma, that one needs for the analysis of the algorithm SCHÖNING in Section 5.3. For all natural numbers, let **Word (i, j)** denote the set of all strings (words) over the alphabet $\{0, 1\}$ of length $j + 2i > 0$, $j > 0$, that have the following properties:

(i) The number of symbols 1 is exactly $j + i$ (i.e., the number of 0s is exactly i), and

(ii) every suffix of the string contains more 1s than 0s.

Lemma A.3.69. *Let i and j be non-negative integers, $0 < j$. Then,*

$$|\text{Word}\,(i,j)\,| = \binom{j+2i}{i} \cdot \frac{j}{j+2i}. \tag{A.21}$$

Proof. We prove this assertion by induction with respect to $n = j + 2i$.

(i) We prove the claim for $n \leq 3$.
First, consider the case $i = 0$. Then the set Word $(0, j)$ contains only the word 1^j for every $j \in \mathbb{N} - \{0\}$, and, correspondingly,

$$\binom{j}{0} \cdot \frac{j}{j} = 1.$$

Consequently, (A.21) holds for $n \in \{1, 2\}$, because property (ii) forces $i = 0$ in these cases.
If $n = 3$, then either $i = j = 1$, or ($i = 0$ and $j = 3$). So, it remains to solve the case $i = j = 1$. We immediately see that Word $(1,1) = \{011\}$ and, correspondingly,

$$\binom{3}{1} \cdot \frac{1}{3} = 1.$$

(ii) Let $n > 3$.
Assuming that (A.21) is true for all sets Word (l, r) with $r + 2l < n$, we have to prove that (A.21) holds for all sets Word (i, j) with $j + 2i = n$. To be able to apply the induction hypothesis, we consider two possibilities with respect to the first symbol of the words in Word (i, j).
From the induction hypothesis, the number of words in Word (i, j) beginning with the symbol 1 is exactly[12]

$$|\text{Word}\,(i,j-1)\,| = \binom{j-1+2i}{i} \cdot \frac{j-1}{j-1+2i}. \tag{A.22}$$

The number of words in Word (i, j) that start with 0 is, by the induction hypothesis, exactly

$$|\text{Word}\,(i-1,j+1)\,| = \binom{j+1+2(i-1)}{i-1} \cdot \frac{j+1}{j+1+2(i-1)}. \tag{A.23}$$

Now, we distinguish two cases, namely $j = 1$ and $j > 1$.

(ii).1 Let $j = 1$.
Because of the property (ii), no word in Word (i, j) can begin with 1. Hence,

[12]Observe that this case is possible only if $j > 1$.

$$\begin{aligned}
|\mathrm{Word}\,(i,1)\,| &= |\mathrm{Word}\,(i-1,2)\,| \\
&\underset{(A.23)}{=} \binom{2+2(i-1)}{i-1}\cdot\frac{2}{2(i-1)+2} \\
&= \binom{2i}{i-1}\cdot\frac{1}{i} = \frac{(2i)!}{(i-1)!\cdot(i+1)!}\cdot\frac{1}{i} \\
&= \frac{(2i)!\cdot(2i+1)}{i!\cdot(i+1)!\cdot(2i+1)} = \binom{2i+1}{i}\cdot\frac{1}{2i+1} \\
&= \binom{j+2i}{i}\cdot\frac{1}{j+2i} \\
&\quad \{\text{because } j=1\} \qquad .
\end{aligned}$$

Thus, (A.21) holds for $j = 1$.

(ii).2 Let $j > 1$.
From (A.22) and (A.23), we have

$$\begin{aligned}
|\mathrm{Word}\,(i,j)\,| &= |\mathrm{Word}\,(i,j-1)\,| + |\mathrm{Word}\,(i-1,j+1)\,| \\
&= \binom{j-1+2i}{i}\cdot\frac{j-1}{2i+j-1} \\
&\quad +\binom{j+1+2(i-1)}{i-1}\cdot\frac{j+1}{2(i-1)+j+1} \\
&= \binom{j-1+2i}{j-1+i}\frac{j-1}{2i+j-1} \\
&\quad +\binom{j+2i-1}{i-1}\cdot\frac{j+1}{2i+j-1} \\
&= \frac{j}{2i+j}\cdot\binom{j+2i}{j+i} \\
&\quad \cdot\left(\frac{(j-1)(j+i)}{j\cdot(2i+j-1)}+\frac{(j+1)\cdot i}{j\cdot(2i+j-1)}\right) \\
&= \frac{j}{2i+j}\cdot\binom{j+2i}{j+i} \\
&= \binom{j+2i}{i}\cdot\frac{j}{j+2i}.
\end{aligned}$$

Thus, the induction hypothesis holds for every $j > 1$, too.

□

A.4 Summary

In this appendix, the fundamentals of number theory, algebra, and combinatorics were presented. We also gave proofs of some claims if the proof details were useful for the understanding of the design of algorithms here.

Here, we view mathematics as a collection of instruments for solving various problems. In what follows we list once more the most important and frequently applied mathematical discoveries.

The Prime Number Theorem shows the density of primes among positive integers, namely how many primes are among the n smallest positive integers. This is important to know, because in several applications we consider the set $\text{PRIM}(n)$ of primes smaller than or equal to n as the set of candidates for a witness (or as a kind of fingerprinting), and then one needs to know the number of candidates for analysis of the error probability. Moreover, the Prime Number Theorem is used for designing an algorithm for the generation of random primes.

Lagrange's Theorem is also an instrument for analyzing randomized algorithms, based on the method of abundance of witnesses. In order to bound the number of non-witnesses among the candidates, we search for a group $(A, \circ)$ such that A is a subset of the set of candidates and all non-witnesses are included in a proper subgroup of $(A, \circ)$. In this way one can show that the number of non-witnesses is at most half the number of candidates. We have applied this approach to the analysis of the Solovay-Strassen algorithm for primality testing.

The Chinese Remainder Theorem provides a structured representation of elements of $\mathbb{Z}_n$ for composite integers n. This is important because, for composite n, the set $\mathbb{Z}_n - \{0\}$ is not a group (a field). For instance, we used this representation to find an element outside a subgroup of non-witnesses, and so to show that this subgroup in a proper subgroup of the group of candidates considered.

Fermat's Little Theorem is the starting point for the search for a suitable kind of witness for primality testing, as well as for proving Euler's Criterion, which is the base for the design of the efficient Las Vegas algorithm for generating quadratic non-residues.

The combinatorial relations presented above are especially useful for the analysis of randomized algorithms and for the study of probability success amplification by executing several independent runs on the same input.

For an excellent introduction to discrete mathematics, we warmly recommend the textbooks by Graham, Knuth, and Patashnik [GKP94], Yan [Yan00], Johnsonbaugh [Joh97], and the German textbook by Steger [Ste01].

This page intentionally left blank

References

[AGP92] W. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. In *University of Georgia Mathematics Preprint Series.* 1992.

[AH87] L. Adleman and M. Huang. Recognizing primes in random polynomial time. In *Proc. 19th ACM STOC*, pages 482–469. ACM, 1987.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. Primes in P. Manuscript, 2002.

[AMM77] L. Adleman, K. Manders, and G. Miller. On taking roots in finite fields. In *Proc. 18th IEEE FOCS*, pages 151–163. IEEE, 1977.

[BB96] G. Brassard and P. Bradley. *Fundamentals of Algorithmics.* Prentice Hall, 1996.

[BEY98] A Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[Car12] R. Carmichael. On composite numbers p which satisfy the Fermat congruence $a^{p-1} \equiv 1$. *American Mathematical Monthly*, 19:22–27, 1912.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1990.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2001.

[CW79] J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[Dan49] G. Dantzig. Programming of independant activities, ii. Mathematical model. *Econometrics*, 17:200–211, 1949.

[Die04] M. Dietzfelbinger. Primality testing in polynomial time: From randomized algorithms to “Primes is in P”. In *Lecture Notes in Computer Science 3000.* Springer, 2004.

[DKM$^+$94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan Rohnert. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.

[FKS84] M. Fredman, J. Komlós, and E. Szemerédi. Storing a parse table with $o(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[Fre77] R. Freivalds. Probabilistic machines can use less running time. In *Proc. of IFIP Congress*, *Information Processing*, 77:839–842. North-Holland, 1977.

[FW98] A. Fiat and G. Woeginger. *Online Algorithms. The State of the Art.* In *Lecture Notes in Computer Science 1492.* Springer, 1998.

[GKP94] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1994.

[Gon84] G. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1984.

[Hro97] J. Hromkovič. *Communication Complexity and Parallel Computing.* Springer, 1997.

[Hro03] J. Hromkovič. *Algorithmics for Hard Problems.* Springer, 2nd edition, 2003.

[Hro04] J. Hromkovič. *Theoretical Computer Science.* Springer, 2004.

[HSW01] J. Hromkovič, A. Steinhöfel, and P. Widmayer. Job shop scheduling with unit length tasks: bounds and algorithms. In *Proc. of ICTCS 2001, Lecture Notes in Computer Science*, pages 90–106. Springer, 2001.

[Joh97] R. Johnsonbaugh. *Discrete Mathematics.* Prentice Hall, 4th edition, 1997.

[Kar91] R. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165–201, 1991.

[Kar93] D. Karger. Global min-cuts in RNC and other ramifications of a simple min-cut algorithm. In *ACM–SIAM Symposium on Discrete Algorithms*, volume 4, pages 21–30, 1993.

[Kha94] L. Khachian. A polynomial algorithm for linear programming. In *Doklady Akad. Nauk USSR 224*, volume 5, pages 1093–1096. 1979 (in Russian), Translation *Soviet. Math. Doklady* 20, 191–194.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm? In *Inequalities III*, pages 159–175. Academic Press, 1972.

[KN97] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.

[Knu73] D. Knuth. *Sorting and Searching. Vol 3 of The Art of Computer Programming.* Addison-Wesley, 1973.

[KS93] D. Karger and C. Stein. An $\theta(n^2)$ algorithm for minimum cuts. In *Proc. ACM STOC*, pages 757–765. ACM, 1993.

[MPS98] E. Mayr, M. Prömel, and A. Steger. *Lectures on Proof Verification and Approximation Algorithms.* In *Lecture Notes in Computer Science 1967.* Springer, 1998.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[MS82] K. Mehlhorn and E. Schmidt. Las Vegas is better than determinism in VLSI and distributed computing. In *Proc. of 14th ACM STOC*, ACM, pages 330–337, 1982.

[OW02] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, 4th edition, 2002. in German.

[Pra75] V. Pratt. Every prime has a succint certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[PRRR00] P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim, editors. *Handbook of Randomized Computing.* Kluwer, 2000.

[PS82] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[Rab76] M. Rabin. Probabilistic algorithms. In J. Traub, editor, *Algorithms and Complexity: Recent Results and New Directions*, pages 21–39. Academic Press, 1976.

[Rab80] M. Rabin. Probabilistic algorithm for primality testing. *Journal of Number Theory*, 12:128–138, 1980.

[Ros00] S. Ross. *Introduction to Probability Models*. Academic Press, 2000.

[Sch99] U. Schöning. A probabilistic algorithm for kSAT and constraint satisfaction problems. In *Proc. 40th IEEE FOCS*, pages 410–414. IEEE, 1999.

[Sch01] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001. in German.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[SS77] R. Solovay and V. Strassen. A fast Monte Carlo for primality. *SIAM Journal of Computing*, pages 84–85, 1977.

[SS01] T. Schickinger and A. Steger. *Diskrete Strukturen, Vol 2, Wahrscheinlichkeitstheorie und Statistik*. Springer, 2001. in German.

[Ste01] A. Steger. *Diskrete Strukturen, Vol 1, Kombinatorik–Graphentheorie–Algebra*. Springer, 2001. in German.

[Str96] V. Strassen. Zufalls-Primzahlen und Kryptographie. In I. Wegener, editor, *Highlights aus der Informatik*, pages 253–266. Springer, 1996. in German.

[Vaz01] V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[Weg03] I. Wegener. *Komplexitätstheorie*. Springer, 2003. in German.

[Yan00] S. Yan. *Number Theory for Computing*. Springer, 2000.

This page intentionally left blank

Index