Complexity of Algorithms

Lecture Notes

László Lovász

Eötvös Loránd University

Institute of Mathematics

The first version of this lecture notes was translated and supplemented by Péter Gács (Boston University)

Edited by: Zoltán Király and Dömötör Pálvölgyi

Vetted by: Katalin Friedl

Version: 1.3
December 31, 2013

Contents

Introduction **6**

1 Models of Computation **9**
1.1 Finite automata 10
1.2 The Turing machine 13
1.3 The Random Access Machine 22
1.4 Boolean functions and Boolean circuits 28

2 Algorithmic decidability **36**
2.1 Recursive and recursively enumerable languages 37
2.2 Other undecidable problems 41
2.3 Computability in logic 46
2.3.1 Godel's incompleteness theorem 46
2.3.2 First-order logic 48

3 Computation with resource bounds **54**
3.1 Polynomial time 56
3.2 Other complexity classes 66
3.3 General theorems on space and time complexity 69

4 Non-deterministic algorithms **77**
4.1 Non-deterministic Turing machines 77
4.2 Witnesses and the complexity of non-deterministic algorithms 79
4.3 Examples of languages in NP 84
4.4 NP-completeness 90
4.5 Further NP-complete problems 95

5 Randomized algorithms **104**
5.1 Verifying a polynomial identity 104
5.2 Primality testing 107
5.3 Randomized complexity classes 111

6 Information complexity **115**
6.1 Information complexity 116
6.2 Self-delimiting information complexity 121
6.3 The notion of a random sequence 124
6.4 Kolmogorov complexity, entropy and coding 126

7 Pseudorandom numbers **132**
7.1 Classical methods . . . 133
7.2 The notion of a pseudorandom number generator . . . 134
7.3 One-way functions . . . 138
7.4 Candidates for one-way functions . . . 141
7.4.1 Discrete square roots . . . 142

8 Decision trees **145**
8.1 Algorithms using decision trees . . . 145
8.2 Nondeterministic decision trees . . . 150
8.3 Lower bounds on the depth of decision trees . . . 153

9 Algebraic computations **159**
9.1 Models of algebraic computation . . . 159
9.2 Multiplication . . . 161
9.2.1 Arithmetic operations on large numbers . . . 161
9.2.2 Matrix multiplication . . . 162
9.2.3 Inverting matrices . . . 164
9.2.4 Multiplication of polynomials . . . 165
9.2.5 Discrete Fourier transform . . . 166
9.3 Algebraic complexity theory . . . 168
9.3.1 The complexity of computing square-sums . . . 169
9.3.2 Evaluation of polynomials . . . 169
9.3.3 Formula complexity and circuit complexity . . . 172

10 Parallel algorithms **175**
10.1 Parallel random access machines . . . 175
10.2 The class NC . . . 179

11 Communication complexity **184**
11.1 Communication matrix and protocol-tree . . . 185
11.2 Examples . . . 189
11.3 Non-deterministic communication complexity . . . 190
11.4 Randomized protocols . . . 194

12 An application of complexity: cryptography **196**
12.1 A classical problem . . . 196
12.2 A simple complexity-theoretic model . . . 197
12.3 Public-key cryptography . . . 198
12.4 The Rivest-Shamir-Adleman code (RSA code) . . . 199

13 Circuit complexity **202**
13.1 Lower bound for the Majority Function . . . 203
13.2 Monotone circuits . . . 206

14 Interactive proofs **207**
14.1 How to save the last move in chess? 207
14.2 How to check a password – without knowing it? 208
14.3 How to use your password – without telling it? 209
14.4 How to prove non-existence? 211
14.5 How to verify proofs that keep the main result secret? 213
14.6 How to referee exponentially long papers? 213
14.7 Approximability 215

Introduction

The need to be able to measure the complexity of a problem, algorithm or structure, and to obtain bounds and quantitative relations for complexity arises in more and more sciences: besides computer science, the traditional branches of mathematics, statistical physics, biology, medicine, social sciences and engineering are also confronted more and more frequently with this problem. In the approach taken by computer science, complexity is measured by the quantity of computational resources (time, storage, program, communication) used up by a particular task. These notes deal with the foundations of this theory.

Computation theory can basically be divided into three parts of different character. First, the exact notions of algorithm, time, storage capacity, etc. must be introduced. For this, different mathematical machine models must be defined, and the time and storage needs of the computations performed on these need to be clarified (this is generally measured as a function of the size of input). By limiting the available resources, the range of solvable problems gets narrower; this is how we arrive at different complexity classes. The most fundamental complexity classes provide an important classification of problems arising in practice, but (perhaps more surprisingly) even for those arising in classical areas of mathematics; this classification reflects the practical and theoretical difficulty of problems quite well. The relationship between different machine models also belongs to this first part of computation theory.

Second, one must determine the resource need of the most important algorithms in various areas of mathematics, and give efficient algorithms to prove that certain important problems belong to certain complexity classes. In these notes, we do not strive for completeness in the investigation of concrete algorithms and problems; this is the task of the corresponding fields of mathematics (combinatorics, operations research, numerical analysis, number theory). Nevertheless, a large number of algorithms will be described and analyzed to illustrate certain notions and methods, and to establish the complexity of certain problems.

Third, one must find methods to prove "negative results", i.e., to show that some problems are actually unsolvable under certain resource restrictions. Often, these questions can be formulated by asking whether certain complexity classes are different or empty. This problem area includes the question whether a problem is algorithmically solvable at all; this question can today be considered classical, and there are many important results concerning it; in particular, the decidability or undecidability of most problems of interest is known.

The majority of algorithmic problems occurring in practice is, however, such that algorithmic solvability itself is not in question, the question is only what resources must be used for the solution. Such investigations, addressed to lower bounds, are

very difficult and are still in their infancy. In these notes, we can only give a taste of this sort of results. In particular, we discuss complexity notions like communication complexity or decision tree complexity, where by focusing only on one type of rather special resource, we can give a more complete analysis of basic complexity classes.

It is, finally, worth noting that if a problem turns out to be "difficult" to solve, this is not necessarily a negative result. More and more areas (random number generation, communication protocols, cryptography, data protection) need problems and structures that are guaranteed to be complex. These are important areas for the application of complexity theory; from among them, we will deal with random number generation and cryptography, the theory of secret communication.

We use basic notions of number theory, linear algebra, graph theory and (to a small extent) probability theory. However, these mainly appear in examples, the theoretical results — with a few exceptions — are understandable without these notions as well.

I would like to thank LÁSZLÓ BABAI, GYÖRGY ELEKES, ANDRÁS FRANK, GYULA KATONA, ZOLTÁN KIRÁLY and MIKLÓS SIMONOVITS for their advice regarding these notes, and DEZSŐ MIKLÓS for his help in using MATEX, in which the Hungarian original was written. The notes were later translated into English by PÉTER GÁCS and meanwhile also extended, corrected by him.

László Lovász

Some notation and definitions

A finite set of symbols will sometimes be called an *alphabet*. A finite sequence formed from some elements of an alphabet Σ is called a *word*. The empty word will also be considered a word, and will be denoted by $\emptyset$. The set of words of length n over Σ is denoted by Σ^n, the set of all words (including the empty word) over Σ is denoted by Σ^*. A subset of Σ^*, i.e., an arbitrary set of words, is called a *language*.

Note that the empty language is also denoted by $\emptyset$ but it is different, from the language $\{\emptyset\}$ containing only the empty word.

Let us define some orderings of the set of words. Suppose that an ordering of the elements of Σ is given. In the *lexicographic ordering* of the elements of Σ^*, a word α precedes a word β if either α is a prefix (beginning segment) of β or the first letter which is different in the two words is smaller in α. (E.g., 35244 precedes 35344 which precedes 353447.) The lexicographic ordering does not order all words in a single sequence: for example, every word beginning with 0 precedes the word 1 over the alphabet $\{0, 1\}$. The *increasing order* is therefore often preferred: here, shorter words precede longer ones and words of the same length are ordered lexicographically. This is the ordering of $\{0, 1\}^*$ we get when we write down the natural numbers in the binary number system without the leading 1.

The set of real numbers will be denoted by $\mathbf{R}$, the set of integers by $\mathbf{Z}$ and the set of rational numbers (fractions) by $\mathbf{Q}$. The sign of the set of non-negative real (integer, rational) numbers is $\mathbf{R}_+$ ($\mathbf{Z}_+$, $\mathbf{Q}_+$). When the base of a logarithm will not be indicated it will be understood to be 2.

Let f and g be two real (or even complex) functions defined over the natural numbers. We write

$$f = O(g)$$

if there is a constant $c > 0$ such that for all n large enough we have $|f(n)| \leq c|g(n)|$. We write

$$f = o(g)$$

if g is 0 only at a finite number of places and $f(n)/g(n) \to 0$ if $n \to \infty$. We will also use sometimes an inverse of the big O notation: we write

$$f = \Omega(g)$$

if $g = O(f)$. The notation

$$f = \Theta(g)$$

means that both $f = O(g)$ and $g = O(f)$ hold, i.e., there are constants $c_1, c_2 > 0$ such that for all n large enough we have $c_1 g(n) \leq f(n) \leq c_2 g(n)$. We will also use this notation within formulas. Thus,

$$(n + 1)^2 = n^2 + O(n)$$

means that $(n + 1)^2$ can be written in the form $n^2 + R(n)$ where $R(n) = O(n)$. Keep in mind that in this kind of formula, the equality sign is not symmetrical. Thus, $O(n) = O(n^2)$ but $O(n^2) \neq O(n)$. When such formulas become too complex it is better to go back to some more explicit notation.

Chapter 1

Models of Computation

In this chapter, we will treat the concept of "computation" or algorithm. This concept is fundamental to our subject, but we will not define it formally. Rather, we consider it an intuitive notion, which is amenable to various kinds of formalization (and thus, investigation from a mathematical point of view).

An *algorithm* means a mathematical procedure serving for a computation or construction (the computation of some function), and which can be carried out mechanically, without thinking. This is not really a definition, but one of the purposes of this course is to demonstrate that a general agreement can be achieved on these matters. (This agreement is often formulated as *Church's thesis*.) A computer program in a programming language is a good example of an algorithm specification. Since the "mechanical" nature of an algorithm is its most important feature, instead of the notion of algorithm, we will introduce various concepts of a *mathematical machine*.

Mathematical machines compute some *output* from some *input*. The input and output can be a word (finite sequence) over a fixed alphabet. Mathematical machines are very much like the real computers the reader knows but somewhat idealized: we omit some inessential features (e.g., hardware bugs), and add an infinitely expandable memory.

Here is a typical problem we often solve on the computer: Given a list of names, sort them in alphabetical order. The input is a string consisting of names separated by commas: Bob, Charlie, Alice. The output is also a string: Alice, Bob, Charlie. The problem is to compute a *function* assigning to each string of names its alphabetically ordered copy.

In general, a typical algorithmic problem has infinitely many *instances*, which then have arbitrarily large size. Therefore, we must consider either an infinite family of finite computers of growing size, or some idealized infinite computer. The latter approach has the advantage that it avoids the questions of what infinite families are allowed.

Historically, the first pure infinite model of computation was the *Turing machine*, introduced by the English mathematician Turing in 1936, thus before the invention of programmable computers. The essence of this model is a central part (control unit) that is bounded (has a structure independent of the input) and an infinite storage (memory). More precisely, the memory is an infinite one-dimensional array of cells. The control is a finite automaton capable of making arbitrary local changes to the scanned memory cell and of gradually changing the scanned position. On Turing

machines, all computations can be carried out that could ever be carried out on any other mathematical machine models. This machine notion is used mainly in theoretical investigations. It is less appropriate for the definition of concrete algorithms since its description is awkward, and mainly since it differs from existing computers in several important aspects.

The most important weakness of the Turing machine in comparison to real computers is that its memory is not accessible immediately: in order to read a distant memory cell, all intermediate cells must also be read. This is remedied by the Random Access Machine (RAM). The RAM can reach an arbitrary memory cell in a single step. It can be considered a simplified model of real world computers along with the abstraction that it has unbounded memory and the capability to store arbitrarily large integers in each of its memory cells. The RAM can be programmed in an arbitrary programming language. For the description of algorithms, it is practical to use the RAM since this is closest to real program writing. But we will see that the Turing machine and the RAM are equivalent from many points of view; what is most important, the same functions are computable on Turing machines and the RAM.

Despite their seeming theoretical limitations, we will consider logic circuits as a model of computation, too. A given logic circuit allows only a given size of input. In this way, it can solve only a finite number of problems; it will be, however, evident, that for a fixed input size, every function is computable by a logical circuit. If we restrict the computation time, however, then the difference between problems pertaining to logic circuits and to Turing-machines or the RAM will not be that essential. Since the structure and work of logic circuits is the most transparent and tractable, they play a very important role in theoretical investigations (especially in the proof of lower bounds on complexity).

If a clock and memory registers are added to a logic circuit we arrive at the interconnected finite automata that form the typical hardware components of today's computers.

Let us note that a fixed finite automaton, when used on inputs of arbitrary size, can compute only very primitive functions, and is not an adequate computation model.

One of the simplest models for an infinite machine is to connect an infinite number of similar automata into an array. This way we get a *cellular automaton*.

The key notion used in discussing machine models is *simulation*. This notion will not be defined in full generality, since it refers also to machines or languages not even invented yet. But its meaning will be clear. We will say that machine M simulates machine N if the internal states and transitions of N can be traced by machine M in such a way that from the same inputs, M computes the same outputs as N.

1.1 Finite automata

A *finite automaton* is a very simple and very general computing device. All we assume is that if it gets an input, then it changes its internal state and issues an output. More exactly, a finite automaton has

- an *input alphabet*, which is a finite set Σ,

- an *output alphabet*, which is another finite set Σ', and
- a set Γ of internal states, which is also finite.

To describe a finite automaton, we need to specify, for every input letter $a \in \Sigma$ and state $s \in \Gamma$, the output $\alpha(a, s) \in \Sigma'$ and the new state $\omega(a, s) \in \Gamma$. To make the behavior of the automata well-defined, we specify a *starting state* START.

At the beginning of a computation, the automaton is in state $s_0 = \text{START}$. The input to the computation is given in the form of a string $a_1 a_2 \dots a_n \in \Sigma^*$. The first input letter a_1 takes the automaton to state $s_1 = \omega(a_1, s_0)$; the next input letter takes it into state $s_2 = \omega(a_2, s_1)$ etc. The result of the computation is the string $b_1 b_2 \dots b_n$, where $b_k = \alpha(a_k, s_{k-1})$ is the output at the k-th step.

Thus a finite automaton can be described as a 6-tuple $\langle \Sigma, \Sigma', \Gamma, \alpha, \omega, s_0 \rangle$, where Σ, Σ', Γ are finite sets, $\alpha : \Sigma \times \Gamma \to \Sigma'$ and $\omega : \Sigma \times \Gamma \to \Gamma$ are arbitrary mappings, and $s_0 = \text{START} \in \Gamma$.

Remarks. 1. There are many possible variants of this notion, which are essentially equivalent. Often the output alphabet and the output signal are omitted. In this case, the result of the computation is read off from the state of the automaton at the end of the computation.

In the case of automata with output, it is often convenient to assume that Σ' contains the *blank symbol* $*$; in other words, we allow that the automaton does not give an output at certain steps.

2. Your favorite computer can be modeled by a finite automaton where the input alphabet consists of all possible keystrokes, and the output alphabet consists of all texts that it can write on the screen following a keystroke (we ignore the mouse, ports etc.) Note that the number of states is more than astronomical (if you have 1 GB of disk space, than this automaton has something like $2^{10^{10}}$ states). At the cost of allowing so many states, we could model almost anything as a finite automaton. We will be interested in automata where the number of states is much smaller - usually we assume it remains bounded while the size of the input is unbounded.

Every finite automaton can be described by a directed graph. The nodes of this graph are the elements of Γ, and there is an edge labeled (a, b) from state s to state s' if $\alpha(a, s) = b$ and $\omega(a, s) = s'$. The computation performed by the automaton, given an input $a_1 a_2 \dots a_n$, corresponds to a directed path in this graph starting at node START, where the first labels of the edges on this path are $a_1, a_2, \dots, a_n$. The second labels of the edges give the result of the computation (Figure 1.1).

Example 1.1.1. Let us construct an automaton that corrects quotation marks in a text in the following sense: it reads a text character-by-character, and whenever it sees a quotation like " . . . ", it replaces it by “. . . ”. All the automaton has to remember is whether it has seen an even or an odd number of " symbols. So it will have two states: START and OPEN (i.e., quotation is open). The input alphabet consists of whatever characters the text uses, including ". The output alphabet is the same, except that instead of " we have two symbols “ and ”. Reading any character other than ", the automaton outputs the same symbol and does not change its state. Reading ", it outputs “ if it is in state START and outputs " if it is in state OPEN; and it changes its state (Figure 1.2).

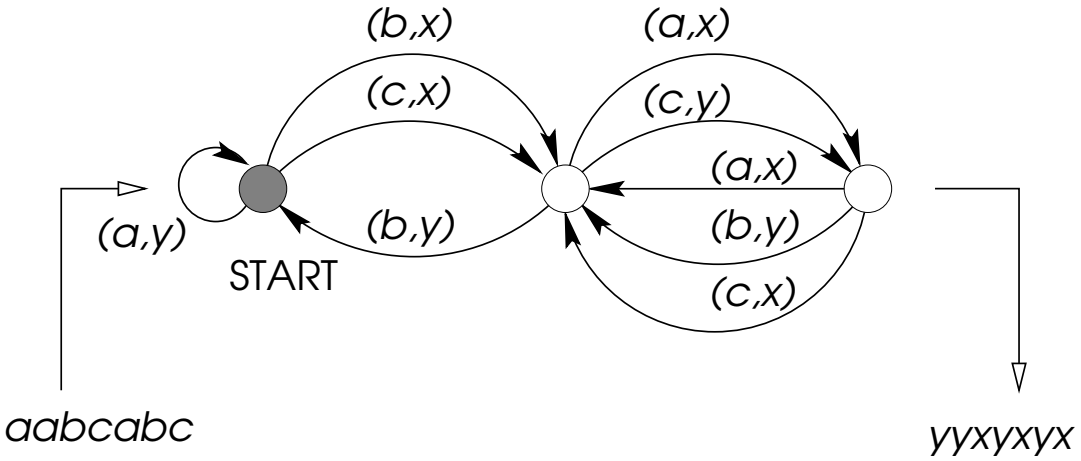

Figure 1.1: A finite automaton

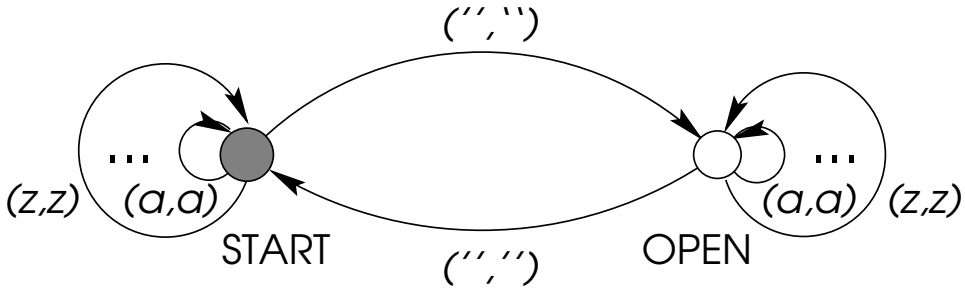

Figure 1.2: An automaton correcting quotation marks

Exercise 1.1.1. *Construct a finite automaton with a bounded number of states that receives two integers in binary and outputs their sum. The automaton gets alternatingly one bit of each number, starting from the right. From the point when we get past the first bit of one of the input numbers, a special symbol • is passed to the automaton instead of a bit; the input stops when two consecutive • symbols occur.*

Exercise 1.1.2. *Construct a finite automaton with as few states as possible that receives the digits of an integer in decimal notation, starting from the left, and the last output is* 1 *(=YES) if the number is divisible by* 7*, and* 0 *(=NO) if it is not.*

Exercise 1.1.3. *a) For a fixed positive integer k, construct a finite automaton that reads a word of length $2k$, and its last output is* 1 *(=YES) if the first half of the word is the same as the second half, and* 0 *(=NO) otherwise.*

b) Prove that the automaton must have at least 2^k states.

The following simple lemma and its variations play a central role in complexity theory. Given words $a, b, c \in \Sigma^*$, let ab^ic denote the word where we first write a, then i copies of b and finally c.

Lemma 1.1.1 (Pumping lemma)**.** *For every regular language $\mathcal{L}$ there exists a natural number k, such that all $x \in \mathcal{L}$ with $|x| \geq k$ can be written as $x = abc$ where $|ab| \leq k$ and $|b| > 0$, such that for every natural number i we have $ab^ic \in \mathcal{L}$.* □

Exercise 1.1.4. *Prove the pumping lemma.*

Exercise 1.1.5. *Prove that* $\mathcal{L} = \{0^n 1^n \mid n \in \mathbb{N}\}$ *is not a regular language.*

Exercise 1.1.6. *Prove that the language of palindromes:*
$\mathcal{L} = \{x_1 \dots x_n x_n \dots x_1 : \quad x_1 \dots x_n \in \Sigma^n\}$ *is not regular.*

1.2 The Turing machine

Informally, a Turing machine is a finite automaton equipped with an unbounded memory. This memory is given in the form of one or more *tapes*, which are infinite in both directions. The tapes are divided into an infinite number of cells in both directions. Every tape has a distinguished *starting cell* which we will also call the 0th cell. On every cell of every tape, a symbol from a finite alphabet Σ can be written. With the exception of finitely many cells, this symbol must be a special symbol $*$ of the alphabet, denoting the "empty cell".

To access the information on the tapes, we supply each tape by a *read-write head*. At every step, this sits on a cell of the tape.

The read-write heads are connected to a *control unit*, which is a finite automaton. Its possible states form a finite set Γ. There is a distinguished starting state "START" and a halting state "STOP". Initially, the control unit is in the "START" state, and the heads sit on the starting cells of the tapes. In every step, each head reads the symbol in the given cell of the tape, and sends it to the control unit. Depending on these symbols and on its own state, the control unit carries out three things:

- it sends a symbol to each head to overwrite the symbol on the tape (in particular, it can give the direction to leave it unchanged);
- it sends one of the commands "MOVE RIGHT", "MOVE LEFT" or "STAY" to each head;
- it makes a transition into a new state (this may be the same as the old one);

The heads carry out the first two commands, which completes one step of the computation. The machine halts when the control unit reaches the "STOP" state.

While the above informal description uses some engineering jargon, it is not difficult to translate it into purely mathematical terms. For our purposes, a *Turing machine* is completely specified by the following data: $T = \langle k, \Sigma, \Gamma, \alpha, \beta, \gamma \rangle$, where $k \geq 1$ is a natural number, Σ and Γ are finite sets, $* \in \Sigma$, $START, STOP \in \Gamma$, and α, β, γ are arbitrary mappings:

$$\begin{aligned}
\alpha &: \Gamma \times \Sigma^k \to \Gamma, \\
\beta &: \Gamma \times \Sigma^k \to \Sigma^k, \\
\gamma &: \Gamma \times \Sigma^k \to \{-1, 0, 1\}^k.
\end{aligned}$$

Here α specifies the new state, β gives the symbols to be written on the tape and γ specifies how the heads move.

In what follows we fix the alphabet Σ and assume that it contains, besides the blank symbol $*$, at least two further symbols, say 0 and 1 (in most cases, it would be sufficient to confine ourselves to these two symbols).

Under the *input* of a Turing machine, we mean the k words initially written on the tapes. We always assume that these are written on the tapes starting at the 0 field and the rest of the tape is empty ($*$ is written on the other cells). Thus, the input of a k-tape Turing machine is an ordered k-tuple, each element of which is a word in Σ^*. Most frequently, we write a non-empty word only on the first tape for input. If we say that the input is a word x then we understand that the input is the k-tuple $(x, \emptyset, \ldots, \emptyset)$.

The *output* of the machine is an ordered k-tuple consisting of the words on the tapes after the machine halts. Frequently, however, we are really interested only in one word, the rest is "garbage". If we say that the output is a single word and don't specify which, then we understand the word on the last tape.

It is practical to assume that the input words do not contain the symbol $*$. Otherwise, it would not be possible to know where is the end of the input: a simple problem like "find out the length of the input" would not be solvable: no matter how far the head has moved, it could not know whether the input has already ended. We denote the alphabet $\Sigma \setminus \{*\}$ by Σ_0. (Another solution would be to reserve a symbol for signaling "end of input" instead.) We also assume that during its work, the Turing machine reads its whole input; with this, we exclude only trivial cases.

Remarks. 1. Turing machines are defined in many different, but from all important points of view equivalent, ways in different books. Often, tapes are infinite only in one direction; their number can virtually always be restricted to two and in many respects even to one; we could assume that besides the symbol $*$ (which in this case we identify with 0) the alphabet contains only the symbol 1; about some tapes, we could stipulate that the machine can only read from them or can only write onto them (but at least one tape must be both readable and writable) etc. The equivalence of these variants (from the point of view of the computations performable on them) can be verified with more or less work but without any greater difficulty and so is left as an exercise to the reader. In this direction, we will prove only as much as we need, but this should give a sufficient familiarity with the tools of such simulations.

2. When we describe a Turing machine, we omit defining the functions at unimportant places, e.g., if the state is "STOP". We can consider such machines as taking $\alpha =$ "STOP", $\beta = *^k$ and $\gamma = 0^k$ at such undefined places. Moreover, if the head writes back the same symbol, then we omit giving the value of β and similarly, if the control unit stays in the same state, we omit giving the value of γ.

Exercise 1.2.1. *Construct a Turing machine that computes the following functions:*

a) $x_1 \ldots x_n \mapsto x_n \ldots x_1$.

b) $x_1 \ldots x_n \mapsto x_1 \ldots x_n x_1 \ldots x_n$.

c) $x_1 \ldots x_n \mapsto x_1 x_1 \ldots x_n x_n$.

d) for an input of length n consisting of all 1's, it outputs the binary form of n; for all other inputs, it outputs "SUPERCALIFRAGILISTICEXPIALIDOCIOUS".

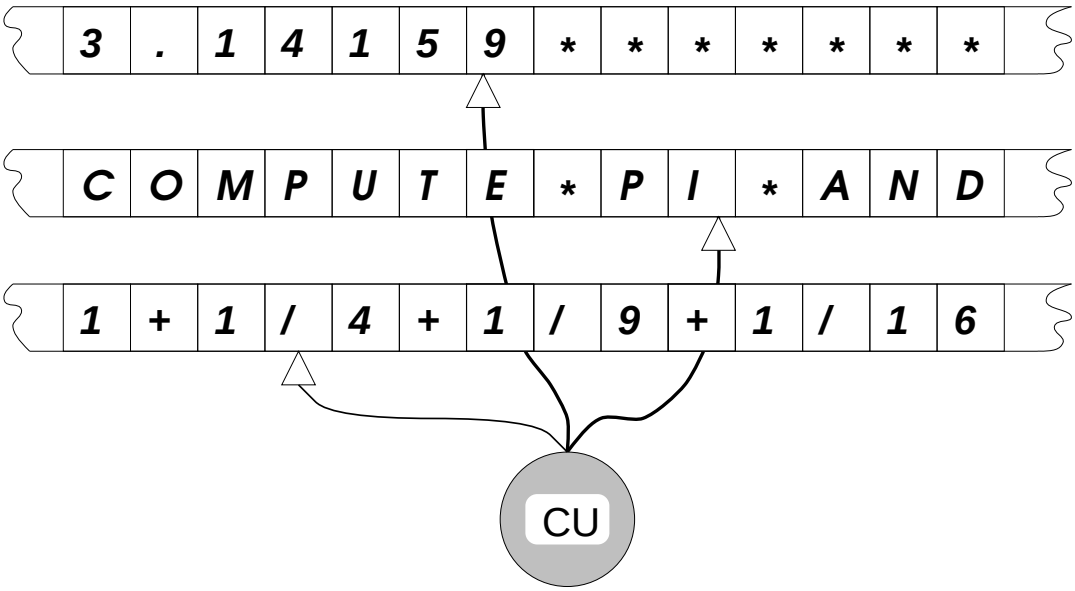

Figure 1.3: A Turing machine with three tapes

e) if the input is the binary form of n, it outputs n 1's (otherwise "SUPERCALIFRAGILISTICEXPIALIDOCIOUS").

f) Solve d) and e) with a machine making at most $O(n)$ steps.

Exercise 1.2.2. *Assume that we have two Turing machines, computing the functions $f : \Sigma_0^* \to \Sigma_0^*$ and $g : \Sigma_0^* \to \Sigma_0^*$. Construct a Turing machine computing the function $f \circ g$.*

Exercise 1.2.3. *Construct a Turing machine that makes $2^{|x|}$ steps for each input x.*

Exercise 1.2.4. *Construct a Turing machine that on input x, halts in finitely many steps if and only if the symbol 0 occurs in x.*

Exercise* 1.2.5. *Show that single tape Turing-machines that are not allowed to write on their tape recognize exactly the regular languages.*

Based on the preceding, we can notice a significant difference between Turing machines and real computers: for the computation of each function, we constructed a separate Turing machine, while on real program-controlled computers, it is enough to write an appropriate program. We will now show that Turing machines can also be operated this way: a Turing machine can be constructed on which, using suitable "programs", everything is computable that is computable on any Turing machine. Such Turing machines are interesting not just because they are more like programmable computers but they will also play an important role in many proofs.

Let $T = \langle k+1, \Sigma, \Gamma_T, \alpha_T, \beta_T, \gamma_T \rangle$ and $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$ be two Turing machines ($k \geq 1$). Let $p \in \Sigma_0^*$. We say that T *simulates* S with program p if for arbitrary words $x_1, \ldots, x_k \in \Sigma_0^*$, machine T halts in finitely many steps on input $(x_1, \ldots, x_k, p)$ if and only if S halts on input $(x_1, \ldots, x_k)$ and if at the time of the stop, the first k tapes of T each have the same content as the tapes of S.

We say that a $(k+1)$-tape Turing machine is *universal* (with respect to k-tape Turing

machines) if for every k-tape Turing machine S over Σ, there is a word (program) p with which T simulates S.

Theorem 1.2.1. *For every number $k \geq 1$ and every alphabet Σ there is a $(k+1)$-tape universal Turing machine.*

Proof. The basic idea of the construction of a universal Turing machine is that on tape $k+1$, we write a table describing the work of the Turing machine S to be simulated. Besides this, the universal Turing machine T writes it up for itself, which state of the simulated machine S is currently in (even if there is only a finite number of states, the fixed machine T must simulate all machines S, so it "cannot keep in mind" the states of S, as S might have more states than T). In each step, on the basis of this, and the symbols read on the other tapes, it looks up in the table the state that S makes the transition into, what it writes on the tapes and what moves the heads make.

First, we give the construction using $k+2$ tapes. For the sake of simplicity, assume that Σ contains the symbols "0", "1", and "–1". Let $S = \langle k, \Sigma, \Gamma_S, \alpha_S, \beta_S, \gamma_S \rangle$ be an arbitrary k-tape Turing machine. We identify each element of the state set Γ_S with a word of length r over the alphabet Σ_0^*. Let the "code" of a given position of machine S be the following word:

$$gh_1 \dots h_k \alpha_S(g, h_1, \dots, h_k) \beta_S(g, h_1, \dots, h_k) \gamma_S(g, h_1, \dots, h_k)$$

where $g \in \Gamma_S$ is the given state of the control unit, and $h_1, \dots, h_k \in \Sigma$ are the symbols read by each head. We concatenate all such words in arbitrary order and obtain so the word a_S, this is what we write on tape $k+1$. On tape $k+2$, we write a state of machine S (initially the name of the START state), so this tape will always have exactly r non-$*$ symbols.

Further, we construct the Turing machine T' which simulates one step or S as follows. On tape $k+1$, it looks up the entry corresponding to the state remembered on tape $k+2$ and the symbols read by the first k heads, then it reads from there what is to be done: it writes the new state on tape $k+2$, then it lets its first k heads write the appropriate symbol and move in the appropriate direction.

For the sake of completeness, we also define machine T' formally, but we also make some concession to simplicity in that we do this only for the case $k = 1$. Thus, the machine has three heads. Besides the obligatory "START" and "STOP" states, let it also have states NOMATCH-ON, NOMATCH-BACK-1, NOMATCH-BACK-2, MATCH-BACK, WRITE, MOVE and AGAIN. Let $h(i)$ denote the symbol read by the i-th head ($1 \leq i \leq 3$). We describe the functions α, β, γ by the table in Figure 1.4 (wherever we do not specify a new state the control unit stays in the old one).

In the run in Figure 1.5, the numbers on the left refer to lines in the above program. The three tapes are separated by triple vertical lines, and the head positions are shown by underscores. To keep the table transparent, some lines and parts of the second tape are omitted.

Now return to the proof of Theorem 1.2.1. We can get rid of the $(k+2)$-nd tape easily: its contents (which is always just r cells) will be placed on cells $-1, -2, \dots, -r$ of the $k+1$-th tape. It seems, however, that we still need two heads on this tape: one moves on its positive half, and one on the negative half (they don't need to cross over). We solve this by doubling each cell: the original symbol stays in its left half, and in its

START:

1: if $h(2) = h(3) \neq *$ then 2 and 3 moves right;

2: if $h(2), h(3) \neq *$ and $h(2) \neq h(3)$ then "NOMATCH-ON" and 2,3 move right;

8: if $h(3) = *$ and $h(2) \neq h(1)$ then "NOMATCH-BACK-1" and 2 moves right, 3 moves left;

9: if $h(3) = *$ and $h(2) = h(1)$ then "MATCH-BACK", 2 moves right and 3 moves left;

18: if $h(3) \neq *$ and $h(2) = *$ then "STOP";

NOMATCH-ON:

3: if $h(3) \neq *$ then 2 and 3 move right;

4: if $h(3) = *$ then "NOMATCH-BACK-1" and 2 moves right, 3 moves left;

NOMATCH-BACK-1:

5: if $h(3) \neq *$ then 3 moves left, 2 moves right;

6: if $h(3) = *$ then "NOMATCH-BACK-2", 2 moves right;

NOMATCH-BACK-2:

7: "START", 2 and 3 moves right;

MATCH-BACK:

10: if $h(3) \neq *$ then 3 moves left;

11: if $h(3) = *$ then "WRITE" and 3 moves right;

WRITE:

12: if $h(3) \neq *$ then 3 writes the symbol $h(2)$ and 2,3 moves right;

13: if $h(3) = *$ then "MOVE", head 1 writes $h(2)$, 2 moves right and 3 moves left;

MOVE:

14: "AGAIN", head 1 moves $h(2)$;

AGAIN:

15: if $h(2) \neq *$ and $h(3) \neq *$ then 2 and 3 move left;

16: if $h(2) \neq *$ but $h(3) = *$ then 2 moves left;

17: if $h(2) = h(3) = *$ then "START", and 2,3 move right.

Figure 1.4: A universal Turing machine

line	Tape 3	Tape 2																								Tape 1
1	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
2	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
3	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
4	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
5	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
6	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
7	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
1	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
8	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
9	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
10	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
11	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
12	∗010∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
13	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗11∗
14	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗
15	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗
16	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗
17	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗
1	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗
18	∗111∗	∗	000	0	000	0	0	010	0	000	0	0	010	1	111	0	1	∗	∗01∗

Figure 1.5: Example run of the universal Turing machine

right half there is a 1 if the corresponding head would just be there (the other right half cells stay empty). It is easy to describe how a head must move on this tape in order to be able to simulate the movement of both original heads. □

Exercise 1.2.6. *Show that if we simulate a k-tape machine on the $(k+1)$-tape universal Turing machine, then on an arbitrary input, the number of steps increases only by a multiplicative factor proportional to the length of the simulating program.*

Exercise 1.2.7. *Let T and S be two one-tape Turing machines. We say that T simulates the work of S by program p (here $p \in \Sigma_0^*$) if for all words $x \in \Sigma_0^*$, machine T halts on input $p * x$ in a finite number of steps if and only if S halts on input x and at halting, we find the same content on the tape of T as on the tape of S. Prove that there is a one-tape Turing machine T that can simulate the work of every other one-tape Turing machine in this sense.*

Our next theorem shows that, in some sense, it is not essential, how many tapes a Turing machine has.

Theorem 1.2.2. *For every k-tape Turing machine S there is a one-tape Turing machine T which replaces S in the following sense: for every word $x \in \Sigma_0^*$, machine T halts in finitely many steps on input x if and only if S halts on input x, and at halt, the same is written on the last tape of T as on the tape of S. Further, if S makes N steps then T makes $O(N^2)$ steps.*

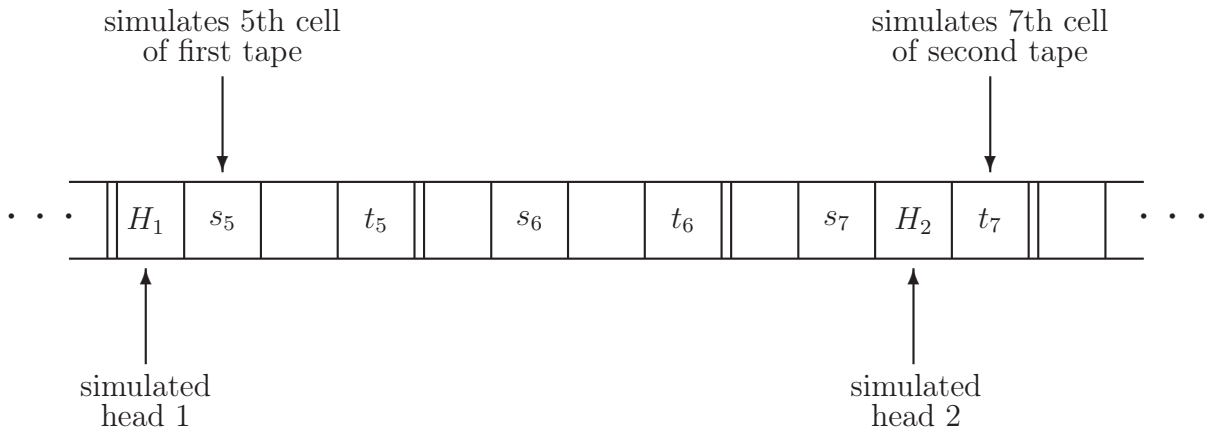

Figure 1.6: One tape simulating two tapes

Proof. We must store the content of the tapes of S on the single tape of T. For this, first we "stretch" the input written on the tape of T: we copy the symbol found on the i-th cell onto the $(2ki)$-th cell. This can be done as follows: first, starting from the last symbol and stepping right, we copy every symbol right by $2k$ positions. In the meantime, we write $*$ on positions $1, 2, \ldots, 2k-1$. Then starting from the last symbol, it moves every symbol in the last block of nonblanks $2k$ positions to right, etc.

Now, position $2ki + 2j - 2$ $(1 \leq j \leq k)$ will correspond to the i-th cell of tape j, and position $2ki + 2j - 3$ will hold a 1 or $*$ depending on whether the corresponding head of S, at the step corresponding to the computation of S, is scanning that cell or not. Also to remember how far the heads ever reached, let us mark by a 0 the two odd-numbered cells of the tape that are such that never contained a 1 yet but each odd-numbered cell between them already did. Thus, we assigned a configuration of T to each configuration of the computation of S.

Now we show how T can simulate the steps of S. First of all, T stores in its states (used as an internal memory) which state S is in. It also knows what is the remainder of the number of the cell modulo $2k$ scanned by its own head. Starting from right, let the head now make a pass over the whole tape. By the time it reaches the end it knows what are the symbols read by the heads of S at this step. From here, it can compute what will be the new state of S, what will its heads write and which direction they will move. Starting backwards, for each 1 found in an odd cell, it can rewrite correspondingly the cell after it, and can move the 1 by $2k$ positions to the left or right if needed. (If in the meantime, it would pass beyond the beginning or ending 0 of the odd cells, then it would move that also by $2k$ positions in the appropriate direction.)

When the simulation of the computation of S is finished, the result must still be "compressed": the content of cell $2ki + 2k - 2$ must be copied to cell i. This can be done similarly to the initial "stretching".

Obviously, the above described machine T will compute the same thing as S. The number of steps is made up of three parts: the times of "stretching", the simulation and the "compression". Let M be the number of cells on machine T which will ever be

scanned by the machine; obviously, $M = O(N)$. The "stretching" and "compression" need time $O(M^2)$. The simulation of one step of S needs $O(M)$ steps, so the simulation needs $O(MN)$ steps. All together, this is still only $O(N^2)$ steps. □

Exercise* **1.2.8.** *Show that every k-tape Turing machine can be simulated by a two-tape one in such a way that if on some input, the k-tape machine makes N steps then the two-tape one makes at most $O(N \log N)$. [Hint: Rather than moving the simulated heads, move the simulated tapes!]*

As we have seen, the simulation of a k-tape Turing machine by a 1-tape Turing machine is not completely satisfactory: the number of steps increases quadratically. This is not just a weakness of the specific construction we have described; there are computational tasks that can be solved on a 2-tape Turing machine in some N steps but *any* 1-tape Turing machine needs N^2 steps to solve them. We describe a simple example of such a task.

A *palindrome* is a word (say, over the alphabet $\{0,1\}$) that does not change when reversed; i.e., $x_1 \dots x_n$ is a palindrome if and only if $x_i = x_{n-i+1}$ for all i. Let us analyze the task of recognizing a palindrome.

Theorem 1.2.3. *(a) There exists a 2-tape Turing machine that decides whether the input word $x \in \{0,1\}^n$ is a palindrome in $O(n)$ steps. (b) Every one-tape Turing machine that decides whether the input word $x \in \{0,1\}^n$ is a palindrome has to make $\Omega(n^2)$ steps in the worst case.*

Proof. Part (a) is easy: for example, we can copy the input on the second tape in n steps, then move the first head to the beginning of the input in n further steps (leave the second head at the end of the word), and compare x_1 with x_n, x_2 with x_{n-1}, etc., in another n steps. Altogether, this takes only $3n$ steps.

Part (b) is more difficult to prove. Consider any one-tape Turing machine that recognizes palindromes. To be specific, say it ends up with writing a "1" on the starting field of the tape if the input word is a palindrome, and a "0" if it is not. We are going to argue that for every n, on some input of length n, the machine will have to make $\Omega(n^2)$ moves.

It will be convenient to assume that n is divisible by 3 (the argument is very similar in the general case). Let $k = n/3$. We restrict the inputs to words in which the middle third is all 0, i.e., to words of the form $x_1 \dots x_k 0 \dots 0 x_{2k+1} \dots x_n$. (If we can show that already among such words, there is one for which the machine must work for $\Omega(n^2)$ time, we are done.)

Fix any j such that $k \le j \le 2k$. Call the dividing line between fields j and $j+1$ of the tape the *cut* after j. Let us imagine that we have a little daemon sitting on this line, and recording the state of the central unit any time the head crosses this line. At the end of the computation, we get a sequence $g_1 g_2 \dots g_t$ of elements of Γ (the length t of the sequence may be different for different inputs), the *j-log* of the given input. The key to the proof is the following observation.

Lemma 1.2.4. *Let $x = x_1 \dots x_k 0 \dots 0 x_k \dots x_1$ and $y = y_1 \dots y_k 0 \dots 0 y_k \dots y_1$ be two different palindromes and $k \le j \le 2k$. Then their j-logs are different.*

Proof of the lemma. Suppose that the j-logs of x and y are the same, say $g_1 \dots g_t$. Consider the input $z = x_1 \dots x_k 0 \dots 0 y_k \dots y_1$. Note that in this input, all the x_i are to the left from the cut and all the y_i are to the right.

We show that the machine will conclude that z is a palindrome, which is a contradiction.

What happens when we start the machine with input z? For a while, the head will move on the fields left from the cut, and hence the computation will proceed exactly as with input x. When the head first reaches field $j+1$, then it is in state g_1 by the j-log of x. Next, the head will spend some time to the right from the cut. This part of the computation will be identical with the corresponding part of the computation with input y: it starts in the same state as the corresponding part of the computation of y does, and reads the same characters from the tape, until the head moves back to field j again. We can follow the computation on input z similarly, and see that the portion of the computation during its m-th stay to the left of the cut is identical with the corresponding portion of the computation with input x, and the portion of the computation during its m-th stay to the right of the cut is identical with the corresponding portion of the computation with input y. Since the computation with input x ends with writing a "1" on the starting field, the computation with input z ends in the same way. This is a contradiction. □

Now we return to the proof of the theorem. For a given m, the number of different j-logs of length less than m is at most

$$1 + |\Gamma| + |\Gamma|^2 + \dots + |\Gamma|^{m-1} = \frac{|\Gamma|^m - 1}{|\Gamma| - 1} < 2|\Gamma|^{m-1}.$$

This is true for any choice of j; hence the number of palindromes whose j-log for some j has length less than m is at most

$$2(k+1)|\Gamma|^{m-1}.$$

There are 2^k palindromes of the type considered, and so the number of palindromes for whose j-logs have length at least m for all j is at least

$$2^k - 2(k+1)|\Gamma|^{m-1}. \tag{1.1}$$

Therefore, if we choose m so that this number is positive, then there will be a palindrome for which the j-log has length at least m for all j. This implies that the daemons record at least $(k+1)m$ moves, so the computation takes at least $(k+1)m$ steps.

It is easy to check that the choice $m = n/\lceil 6 \log |\Gamma| \rceil$ makes (1.1) positive (if n is large), and so we have found an input for which the computation takes at least $(k+1)m > n^2/(18 \log |\Gamma|)$ steps. □

Exercise 1.2.9. *In the simulation of k-tape machines by one-tape machines given above the finite control of the simulating machine T was somewhat bigger than that of the simulated machine S; moreover, the number of states of the simulating machine depends on k. Prove that this is not necessary: there is a one-tape machine that can simulate arbitrary k-tape machines.*

Exercise 1.2.10. *Two-dimensional tape.*

a) Define the notion of a Turing machine with a two-dimensional tape.

b) Show that a two-tape Turing machine can simulate a Turing machine with a two-dimensional tape. [Hint: Store on tape 1, with each symbol of the two-dimensional tape, the coordinates of its original position.]

c) Estimate the efficiency of the above simulation.

Exercise* 1.2.11. *Let $f : \Sigma_0^* \to \Sigma_0^*$ be a function. An* online *Turing machine contains, besides the usual tapes, two extra tapes. The* input tape *is readable only in one direction, the* output tape *is writable only in one direction. An online Turing machine T computes function f if in a single run; for each n, after receiving n symbols $x_1, \ldots, x_n$, it writes $f(x_1 \ldots x_n)$ on the output tape.*

Find a problem that can be solved more efficiently on an online Turing machine with a two-dimensional working tape than with a one-dimensional working tape.

[Hint: On a two-dimensional tape, any one of n bits can be accessed in $\sqrt{n}$ steps. To exploit this, let the input represent a sequence of operations on a "database": insertions and queries, and let f be the interpretation of these operations.]

Exercise 1.2.12. *Tree tape.*

a) Define the notion of a Turing machine with a tree-like tape.

b) Show that a two-tape Turing machine can simulate a Turing machine with a tree-like tape.

c) Estimate the efficiency of the above simulation.

d) Find a problem which can be solved more efficiently with a tree-like tape than with any finite-dimensional tape.

1.3 The Random Access Machine

Trying to design Turing machines for different tasks, one notices that a Turing machine spends a lot of its time by just sending its read-write heads from one end of the tape to the other. One might design tricks to avoid some of this, but following this line of thought we would drift farther and farther away from real-life computers, which have a "random-access" memory, i.e., which can access any field of their memory in one step. So one would like to modify the way we have equipped Turing machines with memory so that we can reach an arbitrary memory cell in a single step.

Of course, the machine has to know which cell to access, and hence we have to assign addresses to the cells. We want to retain the feature that the memory is unbounded; hence we allow arbitrary integers as addresses. The address of the cell to access must itself be stored somewhere; therefore, we allow arbitrary integers to be stored in each cell (rather than just a single element of a finite alphabet, as in the case of Turing machines).

Finally, we make the model more similar to everyday machines by making it programmable (we could also say that we define the analogue of a universal Turing machine). This way we get the notion of a *Random Access Machine* or RAM.

Now let us be more precise. The *memory* of a Random Access Machine is a doubly infinite sequence $\ldots x[-1], x[0], x[1], \ldots$ of memory registers. Each register can store an arbitrary integer. At any given time, only finitely many of the numbers stored in memory are different from 0.

The *program store* is a (one-way) infinite sequence of registers called *lines*. We write here a program of some finite length, in a certain programming language similar to the assembly language of real machines. It is enough, for example, to permit the following statements:

```
x[i]:=0;      x[i]:=x[i]+1;      x[i]:=x[i]-1;
x[i]:=x[i]+x[j];     x[i]:=x[i]-x[j];
x[i]:=x[x[j]];     x[x[i]]:=x[j];
IF x[i]≤ 0 THEN GOTO p.
```

Here, i and j are the addresses of memory registers (i.e., arbitrary integers), p is the address of some program line (i.e., an arbitrary natural number). The instruction before the last one guarantees the possibility of immediate access. With it, the memory behaves as an array in a conventional programming language like Pascal. The exact set of basic instructions is important only to the extent that they should be sufficiently simple to implement, expressive enough to make the desired computations possible, and their number be finite. For example, it would be sufficient to allow the values $-1, -2, -3$ for i, j. We could also omit the operations of addition and subtraction from among the elementary ones, since a program can be written for them. On the other hand, we could also include multiplication, etc.

The *input* of the Random Access Machine is a finite sequence of natural numbers written into the memory registers $x[0], x[1], \ldots$. The Random Access Machine carries out an arbitrary finite program. It stops when it arrives at a program line with no instruction in it. The *output* is defined as the content of the registers $x[i]$ after the program stops.

It is easy to write RAM subroutines for simple tasks that repeatedly occur in programs solving more difficult things. Several of these are given as exercises. Here we discuss three tasks that we need later on in this chapter.

Example 1.3.1 (Value assignment)**.** Let i and j be two integers. Then the assignment

`x[`i`]:=`j

can be realized by the RAM program

```
 x[i]:=0
x[i]:=x[i]+1;
⋮                 } j times
x[i]:=x[i]+1;
```

if j is positive, and

```
x[i]:=0
x[i]:=x[i]-1;
⋮                  } |j| times
x[i]:=x[i]-1;
```

if j is negative.

Example 1.3.2 (Addition of a constant)**.** Let i and j be two integers. Then the statement

`x[`i`]:=x[`i`]+`j

can be realized in the same way as in the previous example, just omitting the first row.

Example 1.3.3 (Multiple branching)**.** Let $p_0, p_1, \ldots, p_r$ be indices of program rows, and suppose that we know that for every i the content of register i satisfies $0 \leq x[i] \leq r$. Then the statement

`GOTO` $p_{x[i]}$

can be realized by the RAM program

```
IF x[i]≤0 THEN GOTO p_0;
x[i]:=x[i]-1:
IF x[i]≤0 THEN GOTO p_1;
x[i]:=x[i]-1:
⋮
IF x[i]≤0 THEN GOTO p_r.
```

Attention must be paid when including this last program segment in a program, since it changes the content of $x[i]$. If we need to preserve the content of $x[i]$, but have a "scratch" register, say $x[-1]$, then we can do

```
x[-1]:=x[i];
IF x[-1]≤0 THEN GOTO p_0;
x[-1]:=x[-1]-1:
IF x[-1]≤0 THEN GOTO p_1;
x[-1]:=x[-1]-1:
⋮
IF x[-1]≤0 THEN GOTO p_r.
```

If we don't have a scratch register than we have to make room for one; since we won't have to go into such details, we leave it to the exercises.

Exercise 1.3.1. *Write a program for the RAM that for a given positive number a*

a) determines the largest number m with $2^m \leq a$;

b) computes its base 2 representation (the i-th bit of a is written to $x[i]$);

c) computes the product of given natural numbers a and b.

If the number of digits of a and b is k, then the program should make $O(k)$ steps involving numbers with $O(k)$ digits.

Note that the number of steps the RAM makes is not the best measure of its working time, as it can make operations involving arbitrarily large numbers. Instead of this, we often speak of *running time*, where the cost of one step is the number of digits of the involved numbers (in base two). Another way to overcome this problem is to specify the number of steps and the largest number of digits an involved number can have (as in Exercise 1.3.1). In Chapter 3 we will return to the question of how to measure running time in more detail.

Now we show that the RAM and the Turing machine can compute essentially the same functions, and their running times do not differ too much either. Let us consider (for simplicity) a 1-tape Turing machine, with alphabet $\{0, 1, 2\}$, where (deviating from earlier conventions but more practically here) let 0 stand for the blank space symbol.

Every input $x_1 \dots x_n$ of the Turing machine (which is a 1–2 sequence) can be interpreted as an input of the RAM in two different ways: we can write the numbers n, $x_1, \dots, x_n$ into the registers $x[1], \dots, x[n]$, or we could assign to the sequence $x_1 \dots x_n$ a single natural number by replacing the 2's with 0 and prefixing a 1. The output of the Turing machine can be interpreted similarly to the output of the RAM.

We will only consider the first interpretation as the second can be easily transformed into the first as shown by Exercise 1.3.1.

Theorem 1.3.1. *For every (multitape) Turing machine over the alphabet $\{0, 1, 2\}$, one can construct a program on the Random Access Machine with the following properties. It computes for all inputs the same outputs as the Turing machine, and if the Turing machine makes N steps then the Random Access Machine makes $O(N)$ steps with numbers of $O(\log N)$ digits.*

Proof. Let $T = \langle 1, \{0, 1, 2\}, \Gamma, \alpha, \beta, \gamma \rangle$. Let $\Gamma = \{1, \dots, r\}$, where 1 = START and r = STOP. During the simulation of the computation of the Turing machine, in register $2i$ of the RAM we will find the same number (0,1 or 2) as in the i-th cell of the Turing machine. Register $x[1]$ will remember where is the head on the tape and store its double (as that register corresponds to it), and the state of the control unit will be determined by where we are in the program.

Our program will be composed of parts P_i $(1 \le i \le r)$ and $Q_{i,j}$ $(1 \le i \le r-1, 0 \le j \le 2)$. Lines P_i for $1 \le i \le r-1$ are accessed if the Turing machine is in state i. They read the content of the tape at the actual position, $x[1]/2$, (from register $x[1]$) and jump accordingly to $Q_{i,x[x[1]]}$.

$x[3] := x[x[1]]$;
IF $x[3] \le 0$ THEN GOTO $Q_{i,0}$;
$x[3] := x[3] - 1$;
IF $x[3] \le 0$ THEN GOTO $Q_{i,1}$;
$x[3] := x[3] - 1$;
IF $x[3] \le 0$ THEN GOTO $Q_{i,2}$;

P_r consists of a single empty program line (so here we stop).

The program parts $Q_{i,j}$ are only a bit more complicated, they simulate the action of the Turing machine when in state i it reads symbol.

$x[3] := 0;$
$x[3] := x[3] + 1;$
$\vdots$
$x[3] := x[3] + 1;$
} $\beta(i,j)$ times
$x[x[1]] := x[3];$
$x[1] := x[1] + \gamma(i,j);$
$x[1] := x[1] + \gamma(i,j);$
$x[3] := 0;$
IF $x[3] \leq 0$ THEN GOTO $P_{\alpha(i,j)}$;

(Here $x[1] := x[1] + \gamma(i,j)$ means $x[1] := x[1] + 1$ resp. $x[1] := x[1] - 1$ if $\gamma(i,j) = 1$ resp. -1, and we omit it if $\gamma(i,j) = 0$.)

The program itself looks as follows.

$x[1] := 0;$
P_1
P_2
$\vdots$
P_r
Q_{00}
$\vdots$
$Q_{r-1,2}$

With this, we have described the simulation of the Turing machine by the RAM. To analyze the number of steps and the size of the number used, it is enough to note that in N steps, the Turing machine can write only to tape positions between $-N$ and N, so in each step of the Turing machine we work with numbers of length $O(\log N)$. □

Remark. In the proof of Theorem 1.3.1, we did not use the instruction $x[i] := x[i] + x[j]$; this instruction is needed when computing the digits of the input if given in a single register (see Exercise 1.3.1). Even this could be accomplished without the addition operation if we dropped the restriction on the number of steps. But if we allow arbitrary numbers as inputs to the RAM then, without this instruction, the number of steps obtained would be exponential even for very simple problems. Let us e.g., consider the problem that the content a of register $x[1]$ must be added to the content b of register $x[0]$. This is easy to carry out on the RAM in a bounded number of steps. But if we exclude the instruction $x[i] := x[i] + x[\mathrm{j}]$ then the time it needs is at least $\min\{|a|, |b|\}$.

Let a program be given now for the RAM. We can interpret its input and output each as a word in $\{0, 1, -, \#\}^*$ (denoting all occurring integers in binary, if needed with a sign, and separating them by $\#$). In this sense, the following theorem holds.

Theorem 1.3.2. *For every Random Access Machine program there is a Turing machine computing for each input the same output. If the Random Access Machine has running time N then the Turing machine runs in $O(N^2)$ steps.*

Proof. We will simulate the computation of the RAM by a four-tape Turing machine.

We write on the first tape the contents of registers $x[i]$ (in binary, and with sign if it is negative). We could represent the content of all non-zero registers. This would cause a problem, however, because of the immediate ("random") access feature of the RAM. More exactly, the RAM can write even into the register with number 2^N using only one step with an integer of N bits. Of course, then the content of the overwhelming majority of the registers with smaller indices remains 0 during the whole computation; it is not practical to keep the content of these on the tape since then the tape will be very long, and it will take exponential time for the head to walk to the place where it must write. Therefore, we will store on the tape of the Turing machine only the content of those registers into which the RAM actually writes. Of course, then we must also record the number of the register in question.

What we will do therefore is that whenever the RAM writes a number y into a register $x[z]$, the Turing machine simulates this by writing the string $\#\#y\#z$ to the end of its first tape. (It never rewrites this tape.) If the RAM reads the content of some register $x[z]$ then on the first tape of the Turing machine, starting from the back, the head looks up the first string of form $\#\#u\#z$; this value u shows what was written in the z-th register the last time. If it does not find such a string then it treats $x[z]$ as 0.

Each instruction of the "programming language" of the RAM is easy to simulate by an appropriate Turing machine using only the three other tapes. Our Turing machine will be a "supermachine" in which a set of states corresponds to every program line. These states form a Turing machine which carries out the instruction in question, and then it brings the heads to the end of the first tape (to its last nonempty cell) and to cell 0 of the other tapes. The STOP state of each such Turing machine is identified with the START state of the Turing machine corresponding to the next line. (In case of the conditional jump, if $x[i] \leq 0$ holds, the "supermachine" goes into the starting state of the Turing machine corresponding to line p.) The START of the Turing machine corresponding to line 0 will also be the START of the supermachine. Besides this, there will be yet another STOP state: this corresponds to the empty program line.

It is easy to see that the Turing machine thus constructed simulates the work of the RAM step-by-step. It carries out most program lines in a number of steps proportional to the number of digits of the numbers occurring in it, i.e., to the running time of the RAM spent on it. The exception is readout, for which possibly the whole tape must be searched. Since the length of the tape is $O(N)$, the total number of steps is $O(N^2)$. □

Exercise 1.3.2. *Let $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ be a polynomial with integer coefficients $a_0, \ldots, a_n$. Write a RAM program computing the coefficients of the polynomial $(p(x))^2$ from those of $p(x)$. Estimate the running time of your program in terms of n and $K = \max\{|a_0|, \ldots, |a_n|\}$.*

Exercise 1.3.3. *Prove that if a RAM is not allowed to use the instruction $x[i] := x[i] + x[j]$, then adding the content a of $x[1]$ to the content b of $x[2]$ takes at least $\min\{|a|, |b|\}$ steps.*

Exercise 1.3.4. *Since the RAM is a single machine the problem of universality cannot be stated in exactly the same way as for Turing machines: in some sense, this single RAM is universal. However, the following "self-simulation" property of the RAM comes close. For a RAM program p and input x, let $R(p, x)$ be the output of the RAM. Let*

$\langle p, x\rangle$ be the input of the RAM that we obtain by writing the symbols of p one-by-one into registers $1, 2, \ldots$, encoding each symbol by some natural number, followed by a -1, and then by the registers containing the original sequence x. Prove that there is a RAM program u such that for all RAM programs p and inputs x we have $R(u, \langle p, x\rangle) = R(p, x)$.

1.4 Boolean functions and Boolean circuits

A *Boolean function* is a mapping $f : \{0,1\}^n \to \{0,1\}$. The values 0,1 are sometimes identified with the values False, True and the variables in $f(x_1, \ldots, x_n)$ are sometimes called *Boolean* (or *logical*) *variables* (or *data types*). In many algorithmic problems, there are n input Boolean variables and one output bit. For example: given a graph G with N nodes, suppose we want to decide whether it has a Hamiltonian cycle. In this case, the graph can be described with $\binom{N}{2}$ Boolean variables: the nodes are numbered from 1 to N and $x_{i,j}$ $(1 \le i < j \le N)$ is 1 if i and j are connected and 0 if they are not. The value of the function $f(x_{1,2}, x_{1,3}, \ldots, x_{n-1,n})$ is 1 if there is a Hamiltonian cycle in G and 0 if there is not. The problem is to compute the value of this (implicitly given) Boolean function.

There are only four one-variable Boolean functions: the identically 0, the identically 1, the identity and the *negation*: $x \to \overline{x} = 1 - x$. We also use the notation $\neg x$. There are 16 Boolean functions with 2 variables (because there are 2^4 mappings of $\{0,1\}^2$ into $\{0,1\}$). We describe only some of these two-variable Boolean functions: the operation of *conjunction* (logical AND):

$$x \wedge y = \begin{cases} 1, & \text{if } x = y = 1, \\ 0, & \text{otherwise,} \end{cases}$$

this can also be considered as the common or mod 2 multiplication, the operation of *disjunction* (logical OR)

$$x \vee y = \begin{cases} 0, & \text{if } x = y = 0, \\ 1, & \text{otherwise,} \end{cases}$$

the *binary addition* (logical exclusive OR a.k.a. XOR)

$$x \oplus y \equiv x + y \bmod 2.$$

Among Boolean functions with several variables, one has the logical AND, OR and XOR defined in the natural way. A more interesting function is MAJORITY, which is defined as follows:

$$\text{MAJORITY}(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if at least } n/2 \text{ of the variables is 1;} \\ 0, & \text{otherwise.} \end{cases}$$

The bit-operations are connected by a number of useful identities. All three operations AND, OR and XOR are associative and commutative. There are several

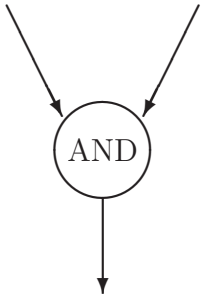

Figure 1.7: A node of a logic circuit

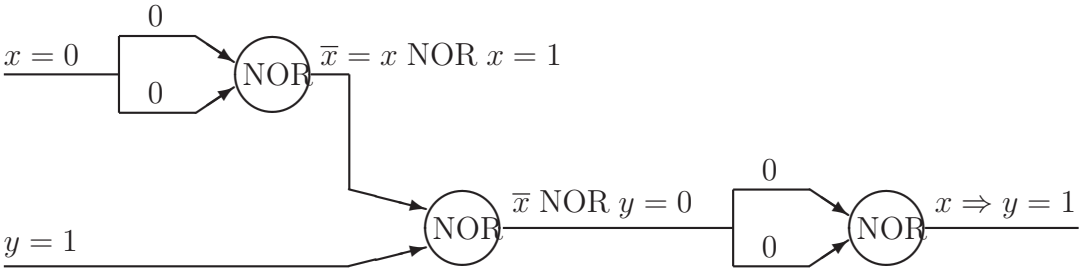

Figure 1.8: A NOR circuit computing $x \Rightarrow y$, with assignment on edges

distributivity properties:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

and

$$x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$$

The De Morgan identities connect negation with conjunction and disjunction:

$$\overline{x \wedge y} = \overline{x} \vee \overline{y},$$
$$\overline{x \vee y} = \overline{x} \wedge \overline{y}$$

Expressions composed using the operations of negation, conjunction and disjunction are called *Boolean polynomials*.

Lemma 1.4.1. *Every Boolean function is expressible as a Boolean polynomial.*

Proof. Let $a_1, \ldots, a_n \in \{0, 1\}$. Let

$$z_i = \begin{cases} x_i, & \text{if } a_i = 1, \\ \overline{x}_i, & \text{if } a_i = 0, \end{cases}$$

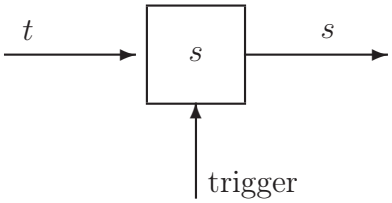

Figure 1.9: A shift register

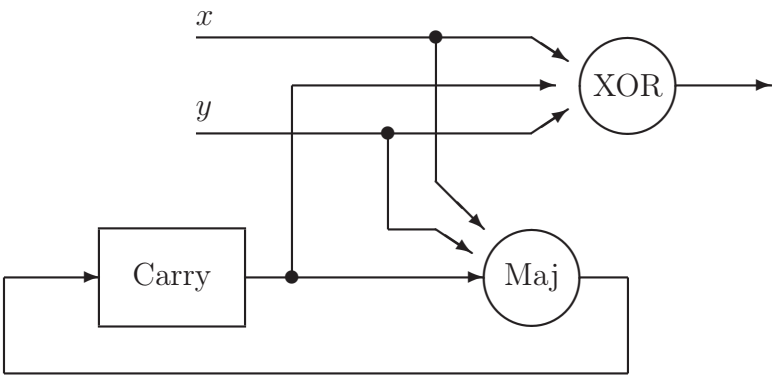

Figure 1.10: A binary adder

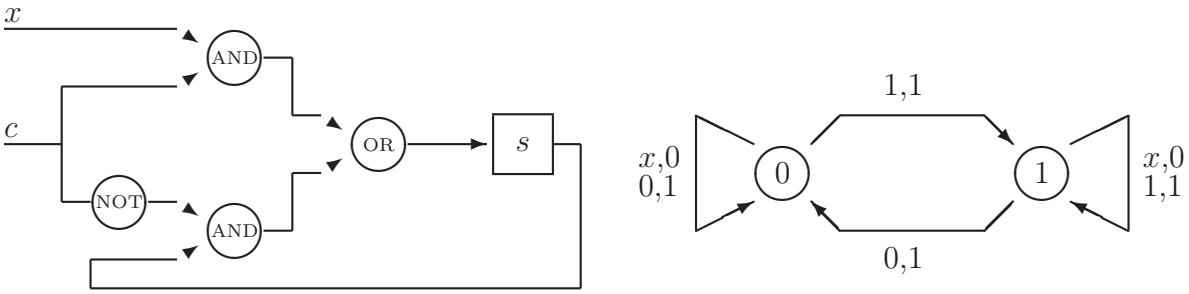

Figure 1.11: Circuit and state-transition diagram of a memory cell

and $E_{a_1,\ldots,a_n}(x_1,\ldots,x_n) = z_1 \wedge \cdots \wedge z_n$. Notice that $E_{a_1,\ldots,a_n}(x_1,\ldots,x_n) = 1$ holds if and only if $(x_1,\ldots,x_n) = (a_1,\ldots,a_n)$. Hence

$$f(x_1,\ldots,x_n) = \bigvee_{f(a_1,\ldots,a_n)=1} E_{a_1,\ldots,a_n}(x_1,\ldots,x_n).$$

□

The Boolean polynomial constructed in the above proof has a special form. A Boolean polynomial consisting of a single (negated or unnegated) variable is called a *literal*. We call an *elementary conjunction* a Boolean polynomial in which variables and negated variables are joined by the operation "∧". (As a degenerate case, the constant 1 is also an elementary conjunction, namely the empty one.) A Boolean polynomial is a *disjunctive normal form* if it consists of elementary conjunctions, joined by the operation "∨". We allow also the empty disjunction, when the disjunctive normal form has no components. The Boolean function defined by such a normal form is identically 0. In general, let us call a Boolean polynomial *satisfiable* if it is not identically 0.

By a *disjunctive k-normal form*, we understand a disjunctive normal form in which every conjunction contains at most k literals.

Example 1.4.1. Here is an important example of a Boolean function expressed by disjunctive normal form: the *selection function*. Borrowing the notation from the programming language C, we define it as

$$x?y:z = \begin{cases} y, & \text{if } x = 1, \\ z, & \text{if } x = 0. \end{cases}$$

It can be expressed as $x?y:z = (x \wedge y) \vee (\neg x \wedge z)$.

Interchanging the role of the operations "∧" and "∨", we can define the *elementary disjunction* and *conjunctive normal form*. The empty conjunction is also allowed, it is the constant 1. In general, let us call a Boolean polynomial a *tautology* if it is identically 1.

We have seen that all Boolean functions can be expressed by a disjunctive normal form. From the disjunctive normal form, we can obtain a conjunctive normal form, applying the distributivity property repeatedly, this is a way to decide whether the polynomial is a tautology. Similarly, an algorithm to decide whether a polynomial is satisfiable is to bring it to a disjunctive normal form. Both algorithms can take very long time.

In general, one and the same Boolean function can be expressed in many ways as a Boolean polynomial. Given such an expression, it is easy to compute the value of the function. However, most Boolean functions can be expressed only by very large Boolean polynomials; this may even be so for Boolean functions that can be computed fast, e.g. the MAJORITY function.

One reason why a computation might be much faster than the size of the Boolean polynomial is that the size of a Boolean polynomial does not reflect the possibility of reusing partial results. This deficiency is corrected by the following more general formalism.

Let G be a directed graph with numbered nodes (called gates) that does not contain any directed cycle (i.e., is *acyclic*, a.k.a. *DAG*). The sources, i.e., the nodes without incoming edges, are called *input nodes*. We assign a literal (a variable or its negation) to each input node. The sinks of the graph, i.e., the nodes without outgoing edges, will be called *output nodes*. (In what follows, we will deal most frequently with the case when there is only one output node.)

Each node v of the graph that is not a source, i.e., which has some indegree $d = d^+(v) > 0$, computes a Boolean function $F_v : \{0,1\}^d \to \{0,1\}$. The incoming edges of the node are numbered in some increasing order and the variables of the function F_v are made to correspond to them in this order. Such a graph is called a *circuit*.

The *size* of the circuit is the number of gates (including the input gates); its *depth* is the maximal length of paths leading from input nodes to output nodes.

Every circuit H determines a function. We assign to each input node the value of the assigned literal. This is the *input assignment*, or *input* of the computation. From this, we can compute at each node v a value $x(v) \in \{0,1\}$: if the start nodes $u_1, \ldots, u_d$ of the incoming edges have already received a value then v receives the value $F_v(x(u_1), \ldots, x(u_d))$. The value at the sinks give the *output* of the computation. We will say that the function defined this way is *computed* by the circuit H. Single sink circuits determine Boolean functions.

Exercise 1.4.1. *Prove that in the above definition, the circuit computes a unique output for every possible input assignment.*

Example 1.4.2. A NOR (negated OR) circuit computing $x \Rightarrow y$. We use the formulas

$$x \Rightarrow y = \neg(\neg x \,\mathrm{NOR}\, y),\ \neg x = x \,\mathrm{NOR}\, x.$$

If the states of the input nodes of the circuit are x and y, then the state of the output node is $x \Rightarrow y$. The assignment can be computed in 3 stages, since the longest path has 3 edges. See Figure 1.8.

Example 1.4.3. For a natural number n we can construct a circuit that will simultaneously compute all the functions $E_{a_1,\ldots,a_n}(x_1, \ldots, x_n)$ (as defined above in the proof of Lemma 1.4.1) for all values of the vector $(a_1, \ldots, a_n)$. This circuit is called the *decoder circuit* since it has the following behavior: for each input $x_1, \ldots, x_n$ only one output node, namely $E_{x_1,\ldots,x_n}$ will be true. If the output nodes are consecutively numbered then we can say that the circuit decodes the binary representation of a number k into the k-th position in the output. This is similar to addressing into a memory and is indeed the way a "random access" memory is addressed. Suppose that a decoder circuit is given for n. To obtain one for $n+1$, we split each output $y = E_{a_1,\ldots,a_n}(x_1, \ldots, x_n)$ in two, and form the new nodes

$$E_{a_1,\ldots,a_n,1}(x_1, \ldots, x_{n+1}) = y \wedge x_{n+1},$$
$$E_{a_1,\ldots,a_n,0}(x_1, \ldots, x_{n+1}) = y \wedge \neg x_{n+1},$$

using a new copy of the input x_{n+1} and its negation.

Of course, every Boolean function is computable by a trivial (depth 1) circuit in which a single (possibly very complicated) gate computes the output immediately from the input. The notion of circuits is interesting if we restrict the gates to some simple

operations (AND, OR, exclusive OR, implication, negation, etc.). If each gate is a conjunction, disjunction or negation then using the De Morgan rules, we can push the negations back to the inputs which, as literals, can be negated variables anyway. If all gates are disjunctions or conjunctions then the circuit is called *Boolean*.

The in-degree of the nodes is called *fan-in*. This is often restricted to 2 or to some fixed maximum. Sometimes, bounds are also imposed on the out-degree, or *fan-out*. This means that a partial result cannot be "freely" distributed to an arbitrary number of places.

Exercise 1.4.2. *Prove that for every Boolean circuit of size N, there is a Boolean circuit of size at most N^2 with indegree 2, computing the same Boolean function.*

Exercise 1.4.3. *Prove that for every circuit of size N and indegree 2 there is a Boolean circuit of size $O(N)$ and indegree at most 2 computing the same Boolean function.*

Exercise 1.4.4. *A Boolean function is* monotone *if its value does not decrease whenever any of the variables is increased. Prove that for every Boolean circuit computing a monotone Boolean function there is another one that computes the same function and uses only nonnegated variables and constants as inputs.*

Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function and let

$$f(x_1, \ldots, x_n) = E_1 \vee \cdots \vee E_N$$

be its representation by a disjunctive normal form. This representation corresponds to a depth 2 circuit in the following manner: let its input points correspond to the variables $x_1, \ldots, x_n$ and the negated variables $\overline{x}_1, \ldots, \overline{x}_n$. To every elementary conjunction E_i, let there correspond a vertex into which edges run from the input points belonging to the literals occurring in E_i, and which computes the conjunction of these. Finally, edges lead from these vertices into the output point t which computes their disjunction. Note that this circuit has large fan-in and fan-out.

Exercise 1.4.5. *Prove that the Boolean polynomials are in one-to-one correspondence with those Boolean circuits that are trees.*

We can consider each Boolean circuit as an algorithm serving to compute some Boolean function. It can be seen immediately, however, that circuits are less flexible less than e.g., Turing machines: a circuit can deal only with inputs and outputs of a given size. It is also clear that (since the graph is acyclic) the number of computation steps is bounded. If, however, we fix the length of the input and the number of steps then by an appropriate circuit, we can already simulate the work of every Turing machine computing a single bit. We can express this also by saying that every Boolean function computable by a Turing machine in a certain number of steps is also computable by a suitable, not too big, Boolean circuit.

Theorem 1.4.2. *For every Turing machine T and every pair $n, N \geq 1$ of numbers there is a Boolean circuit with n inputs, depth $O(N)$, indegree at most 2, that on an input $(x_1, \ldots, x_n) \in \{0,1\}^n$ computes 1 if and only if after N steps of the Turing machine T, on the 0'th cell of the first tape, there is a 1.*

(Without the restrictions on the size and depth of the Boolean circuit, the statement would be trivial since every Boolean function can be expressed by a Boolean circuit.)

Proof. Let us be given a Turing machine $T = \langle k, \Sigma, \alpha, \beta, \gamma \rangle$ and $n, N \geq 1$. For simplicity, assume $k = 1$. Let us construct a directed graph with vertices $v[t, g, p]$ and $w[t, p, h]$ where $0 \leq t \leq N$, $g \in \Gamma$, $h \in \Sigma$ and $-N \leq p \leq N$. An edge runs into every point $v[t+1, g, p]$ and $w[t+1, p, h]$ from the points $v[t, g', p+\varepsilon]$ and $w[t, p+\varepsilon, h']$ ($g' \in \Gamma$, $h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$). Let us take n input points $s_0, \ldots, s_{n-1}$ and draw an edge from s_i into the points $w[0, i, h]$ ($h \in \Sigma$). Let the output point be $w[N, 0, 1]$.

In the vertices of the graph, the logical values computed during the evaluation of the Boolean circuit (which we will denote, for simplicity, just like the corresponding vertex) describe a computation of the machine T as follows: the value of vertex $v[t, g, p]$ is true if after step t, the control unit is in state g and the head scans the p-th cell of the tape. The value of vertex $w[t, p, h]$ is true if after step t, the p-th cell of the tape holds symbol h.

Certain ones among these logical values are given. The machine is initially in the state START, and the head starts from cell 0:

$$v[0, g, p] = \begin{cases} 1, & \text{if } g = START \text{ and } p = 0, \\ 0, & \text{otherwise,} \end{cases}$$

further we write the input onto cells $0, \ldots, n-1$ of the tape:

$$w[0, p, h] = \begin{cases} 1, & \text{if } (p < 0 \text{ or } p \geq n), \text{ and } h = *, \\ & \text{or if } 0 \leq p \leq n-1 \text{ and } h = x_p, \\ 0, & \text{otherwise.} \end{cases}$$

The rules of the Turing machine tell how to compute the logical values corresponding to the rest of the vertices:

$$v[t+1, g, p] = \bigvee_{\substack{g' \in \Gamma \\ h' \in \Sigma \\ \alpha(g', h') = g}} \Big(v[t, g', p - \gamma(g', h')] \wedge w[t, p - \gamma(g', h'), h'] \Big)$$

$$w[t+1, p, h] = \Big(w[t, p, h] \wedge \bigwedge_{g' \in \Gamma} \overline{v[t, g', p]} \Big) \vee \Big(\bigvee_{\substack{g' \in \Gamma \\ h' \in \Sigma \\ \beta(g', h') = h}} \big(v[t, g', p] \wedge w[t, p, h'] \big) \Big)$$

It can be seen that these recursions can be taken as logical functions which turn the graph into a Boolean circuit computing the desired functions. The size of the circuit will be $O(N^2)$, its depth $O(N)$. Since the in-degree of each point is at most $3|\Sigma| \cdot |\Gamma| = O(1)$, we can transform the circuit into a Boolean circuit of similar size and depth. □

Remark. Our construction of a universal Turing machine in Theorem 1.2.1 is inefficient and unrealistic. For most commonly used transition functions α, β, γ, a table is a very inefficient way to store the description. A Boolean circuit (with a Boolean vector output) is often a vastly more economical representation. It is possible to construct a universal one-tape Turing machine V_1 taking advantage of such a representation. The beginning of the tape of this machine would not list the table of the transition function of the simulated machine, but would rather describe the Boolean circuit computing it, along with a specific state of this circuit. Each stage of the simulation would first simulate the Boolean circuit to find the values of the functions α, β, γ and then proceed as before.

Exercise 1.4.6. *Consider that x_1x_0 is the binary representation of an integer $x = 2x_1 + x_0$ and similarly, y_1y_0 is a binary representation of a number y. Let $f(x_0, x_1, y_0, y_1, z_0, z_1)$ be the Boolean formula which is true if and only if z_1z_0 is the binary representation of the number $x + y \bmod 4$.*

Express this formula using only conjunction, disjunction and negation.

Exercise 1.4.7. *Convert into disjunctive normal form the following Boolean functions.*

a) $x + y + z \bmod 2$,

b) $x + y + z + t \bmod 2$.

Exercise 1.4.8. *Convert the formula $(x \wedge y \wedge z) \Rightarrow (u \wedge v)$ into conjunctive normal form.*

Exercise 1.4.9. *For each n, construct a Boolean circuit whose gates have indegree ≤ 2, with size $O(2^n)$ with $2^n + n$ inputs and which is universal in the following sense: for all binary strings p of length 2^n and binary string x of length n, the output of the circuit with input xp is the value, with argument x, of the Boolean function whose truth table (i.e., output values) is given by p. [Hint: use the decoder circuit of Example 1.4.3.]*

Exercise 1.4.10. *The gates of the Boolean circuits in this exercise are assumed to have indegree ≤ 2.*

a) Prove the existence of a constant c, such that for all n, there is a Boolean function for which each Boolean circuit computing it has size at least $c \cdot 2^n/n$. [Hint: count the number of circuits of size k.]

b) For a Boolean function f with n inputs, show that the size of the Boolean circuit needed for its implementation is $O(2^n/n)$.*

Chapter 2

Algorithmic decidability

In this chapter, we study the question: which problems can be solved by any algorithm (or computing device) at all?

Until the 1930's, it was the consensus among mathematicians — mostly not spelled out precisely — that every mathematical question that can be formulated precisely, can also be solved. This statement has two interpretations. We can talk about a single yes-or-no question (say: is every planar graph 4-colorable? is every even integer larger than 2 expressible as the sum of two primes?), and then the decision means that it can be proved or disproved from the axioms of set theory (which were, and still are, generally accepted as the axioms of mathematics). This belief was destroyed by the the Austrian mathematician Kurt Gödel, who published a famous result in 1931, the First Incompleteness Theorem of logic, which implies that there are perfectly well formulated mathematical questions that cannot be answered from the axioms of set theory.

Now one could think that this is a weakness of this particular system of axioms: perhaps by adding some generally accepted axioms (which had been overlooked) one could get a new system that would allow us to decide the truth of every well-formulated mathematical statement. The First Incompleteness Theorem, however, proves that this hope was also vain: no matter how we extend the axiom system of set theory (allowing even infinitely many axioms, subject to some reasonable restrictions: no contradiction should be derivable and it should be possible to decide about a statement whether it is an axiom or not), still there remain unsolvable problems.

The second meaning of the question of decidability is when we are concerned with a *family* of questions and are looking for an *algorithm* that decides each of them. In 1936, Church formulated a family of problems for which he could prove that they are not decidable by any algorithm. For this statement to make sense, the mathematical notion of an algorithm had to be created. Church used tools from logic, the notion of *recursive functions*, to formalize the notion of algorithmic solvability.

Similarly as in connection with Gödel's Theorem, it seems quite possible that one could define algorithmic solvability in a different way, or extend the arsenal of algorithms with new tools, allowing the solution of new problems. In the same year when Church published his work, Turing created the notion of a Turing machine. Nowadays we call something *algorithmically computable* if it can be computed by some Turing machine. But it turned out that Church's original model is equivalent to the Turing machine in the sense that the same computational problems can be solved by them. We

have seen in the previous chapter that the same holds for the Random Access Machine. Many other computational models have been proposed (some are quite different from the Turing machine, RAM, or any real-life computer, like quantum computing or DNA computing), but nobody found a machine model that could solve more computational problems than the Turing machine.

Church in fact anticipated this by formulating the so-called *Church Thesis*, according to which every "calculation" can be formalized in the system he gave. Today we state this hypothesis in the form that all functions computable on any computing device are computable on a Turing machine. As a consequence of this thesis (if we accept it) we can simply speak of computable functions without referring to the specific type of machine on which they are computable.

(One could perhaps make one exception from the Church Thesis for algorithms using randomness. These can solve algorithmically unsolvable computational problems so that the answer is correct with large probability. See Chapter 6 on Information Complexity.)

2.1 Recursive and recursively enumerable languages

Let Σ be a finite alphabet that contains the symbol "$*$". We will allow as input for a Turing machine words that do not contain this special symbol: only letters from $\Sigma_0 = \Sigma \setminus \{*\}$.

We call a function $f : \Sigma_0^* \to \Sigma_0^*$ *recursive* or *computable* if there exists a Turing machine that for any input $x \in \Sigma_0^*$ will stop after finite time with $f(x)$ written on its first tape.

Remark. We have seen in the previous chapter that the definition does not change if we assume that $k = 1$, i.e., the Turing machine has only one tape.

The notions of recursive, as well as that of "recursively enumerable" and "partial recursive" defined below can be easily extended, in a unique way, to functions and sets over some countable sets different from Σ_0^*, like the set of natural numbers, the set N^* of finite strings of natural numbers, etc. The extension goes with help of some standard coding of, e.g., the set of natural numbers by elements of Σ_0^*. Therefore, even though we define these notions only over Σ_0^*, we sometimes use them in connection with functions defined over other domains. This is a bit sloppy but does not lead to any confusion.

We call a language L *recursive* if its characteristic function

$$f_{\mathcal{L}}(x) = \begin{cases} 1, & \text{if } x \in \mathcal{L}, \\ 0, & \text{otherwise,} \end{cases}$$

is recursive. Instead of saying that a language $\mathcal{L}$ is recursive, we can also say that the property defining $\mathcal{L}$ is decidable. If a Turing machine calculates this function then we say that it *decides* the language. It is obvious that every finite language is recursive. Also if a language is recursive then its complement is also recursive.

Remark. It is obvious that there is a continuum of languages (and so uncountably many) but only countably many Turing machines. So there must exist non-recursive languages. At the end of this section, we will see some concrete languages that are non-recursive.

We call the language $\mathcal{L}$ *recursively enumerable* if $\mathcal{L} = \emptyset$ or there exists a recursive function f such that the range of f is $\mathcal{L}$. This means that we can enumerate the elements of $\mathcal{L}$: $\mathcal{L} = \{f(w_0), f(w_1), \ldots\}$, where $\Sigma_0^* = \{w_0, w_1, \ldots\}$. Here, the elements of $\mathcal{L}$ do not necessarily occur in increasing order and repetition is also allowed.

We give an alternative definition of recursively enumerable languages through the following lemma. Let us order the elements of Σ_0^* in *increasing order*, where shorter words precede longer ones and words of the same length are ordered lexicographically. It is easy to construct a Turing machine that enumerates all the words in increasing order, or outputs the j-th word of the ordering for input j.

Lemma 2.1.1. *A language $\mathcal{L}$ is recursively enumerable if and only if there is a Turing machine T such that if we write x on the first tape of T the machine stops if and only if $x \in \mathcal{L}$.*

Proof. Let $\mathcal{L}$ be recursively enumerable. We can assume that it is nonempty. Let $\mathcal{L}$ be the range of f. We construct a Turing machine which on input x stops if and only if $x \in \mathcal{L}$. For each input x the machine calculates $f(y)$ for every $y \in \Sigma_0^*$ (e.g., by taking them in increasing order) and stops if it finds a y such that $f(y) = x$.

On the other hand, let us assume that $\mathcal{L}$ consists of the words on which T stops. We can assume that $\mathcal{L}$ is not empty and $a \in \mathcal{L}$. We construct a Turing machine T_0 that does the following. If the input on its first tape is a natural number i, then it computes the $(i - \lfloor\sqrt{i}\rfloor^2)$-th word of Σ_0^* (say x), simulates T on this input, x, for i steps. If T stops, then T_0 outputs x (by writing it on its last tape). If T does not stop, then T_0 outputs a. Since every word of Σ_0^* will occur for infinitely many values of i the range of T_0 will be $\mathcal{L}$. □

There is nothing really tricky about the function $(i - \lfloor\sqrt{i}\rfloor^2)$; all we need is that for $i = 0, 1, 2, \ldots$ its value takes every non-negative integer infinitely many times. The technique used in this proof, that of simulating infinitely many computations by a single one, is sometimes called "dovetailing".

Now we study the relationship between recursive and recursively enumerable languages.

Lemma 2.1.2. *Every recursive language is recursively enumerable.*

Proof. Let $\mathcal{L}$ be a recursive language. If $\mathcal{L} = \emptyset$, then $\mathcal{L}$ is recursively enumerable by definition, so suppose this is not the case and take some $a \in \mathcal{L}$. Define f as follows.

$$f(x) = \begin{cases} x, & \text{if } x \in \mathcal{L}, \\ a, & \text{if } x \notin \mathcal{L}. \end{cases}$$

Since f is recursive and its range is $\mathcal{L}$, we are done. □

The next theorem characterizes the relation of recursively enumerable and recursive languages.

Theorem 2.1.3. *A language $\mathcal{L}$ is recursive if and only if both languages $\mathcal{L}$ and $\Sigma_0^* \setminus \mathcal{L}$ are recursively enumerable.*

Proof. If $\mathcal{L}$ is recursive then its complement is also recursive, and by the previous lemma, both are recursively enumerable.

On the other hand, let us assume that both $\mathcal{L}$ and its complement are recursively enumerable. We can construct two machines that enumerate them, and a third one simulating both that detects if one of them lists x. Sooner or later this happens and then we know where x belongs. □

Let us call a language *co-recursively enumerable* if its complement is recursively enumerable. So the Theorem 2.1.3 says that a language is recursive if and only if it is recursively enumerable and co-recursively enumerable. It is clear that the class of recursively enumerable languages is countable, so there must be languages that are neither recursively enumerable nor co-recursively enumerable (see also Exercise 2.3.3). What is much less obvious is that there are recursively enumerable languages that are not recursive, since both classes are countable. The construction of such a language is our next goal.

For a Turing machine T let $\mathcal{L}_T$ be the set of those words $x \in \Sigma_0^*$ for which T stops when we write x on *all* of its tapes.

Theorem 2.1.4. *If T is a universal Turing machine with $k+1$ tapes then $\mathcal{L}_T$ is recursively enumerable, but it is not recursive.*

Proof. The first statement can be proved similarly to Lemma 2.1.1. For simplicity, we prove the second statement only for $k = 1$.

Let us assume, by way of contradiction, that $\mathcal{L}_T$ is recursive. Then $\Sigma_0^* \setminus \mathcal{L}_T$ would be recursively enumerable, so there would exist a 1-tape Turing machine T_1 that on input x would stop if and only if $x \notin \mathcal{L}_T$. The machine T_1 can be simulated on T by writing an appropriate program p on the second tape of T. Then writing p on both tapes of T, it would stop if T_1 would stop on input p because of the simulation. The machine T_1 was defined, on the other hand, to stop on p if and only if T does not stop with input p on both tapes (i.e., when $p \notin \mathcal{L}_T$). This is a contradiction. □

This proof uses the so called *diagonalization* technique originating from set theory (where it was used by Cantor to show that the set of all real numbers is not countable). The technique forms the basis of many proofs in logic, set-theory and complexity theory. We will see more of these in what follows.

There is a number of variants of the previous result, asserting the undecidability of similar problems.

Let T be a Turing machine. The *halting problem* for T is the problem to decide, for all possible inputs x, whether T halts on x. Thus, the decidability of the halting problem of T means the decidability of the set of those x for which T halts. We can also speak about the halting problem in general, which means that a pair (T, x) is given where T is a Turing machine (given by its transition table) and x is an input.

Theorem 2.1.5. *There is a 1-tape Turing machine whose halting problem is undecidable.*

Proof. Suppose that the halting problem is decidable for all one-tape Turing machines. Let T be a 2-tape universal Turing machine and let us construct a 1-tape machine T_0 similarly to the proof of Theorem 1.2.2 (with $k = 2$), with the difference that at the start, we write the i-th letter of word x not only in cell $4i$ but also in cell $4i - 2$. Then on an input x, machine T_0 will simulate the work of T, when the latter starts with x on both of its tapes. Since it is undecidable whether T halts for a given input (x, x), it is also undecidable about T_0 whether it halts on a given input x. □

The above proof, though simple, is the prototype of a great number of undecidability proofs. These proceed by taking any problem P_1 known to be undecidable (in this case, membership in $\mathcal{L}_T$) and showing that it can be *reduced* to the problem P_2 at hand (in this case, the halting problem of T_0). The reduction shows that if P_2 is decidable then so is P_1. But since we already know that P_1 is undecidable, we conclude that P_2 is undecidable as well. The reduction of a problem to some seemingly unrelated problem is, of course, often very tricky.

A *description* of a Turing machine is the listing of the sets Σ, Γ (where, as before, the elements of Γ are coded by words over the set Σ_0), and the table of the functions α, β, γ.

Corollary 2.1.6. *It is algorithmically undecidable whether a Turing machine (given by its description) halts on the empty input.*

Proof. Let T be a Turing machine whose halting problem is undecidable. We show that its halting problem can be reduced to the general halting problem on the empty input. Indeed, for each input x, we can construct a Turing machine T_x which, when started with an empty input, writes x on the input tape and then simulates T. If we could decide whether T_x halts then we could decide whether T halts on x. □

Corollary 2.1.7. *It is algorithmically undecidable for a one-tape Turing machine T given by its description whether the set $\mathcal{L}_T$ is empty.*

Proof. For any given one-tape machine S, let us construct a machine T that does the following: it first erases everything from the tape and then turns into the machine S. The description of T can obviously be easily constructed from the description of S. Thus, if S halts on the empty input in finitely many steps then T halts on all inputs in finitely many steps, hence $\mathcal{L}_T = \Sigma_0^*$ is not empty. If S works for infinite time on the empty input then T works infinitely long on all inputs, and thus $\mathcal{L}_T$ is empty. Therefore, if we could decide whether $\mathcal{L}_T$ is empty we could also decide whether S halts on the empty input, which is undecidable. □

Obviously, just as its emptiness, we cannot decide any other property P of $\mathcal{L}_T$ either if the empty language has it and Σ_0^* has not, or vice versa. Even a more general negative result is true. We call a property of a language *trivial* if either all recursively enumerable languages have it or none.

Theorem 2.1.8 (Rice's Theorem)**.** *For any non-trivial language-property P, it is undecidable whether the language $\mathcal{L}_T$ of an arbitrary Turing machine T (given by its description) has this property.*

Thus, it is undecidable on the basis of the description of T whether $\mathcal{L}_T$ is finite, regular, contains a given word, etc.

Proof. We can assume that the empty language does not have property P (otherwise, we can consider the negation of P). Let T_1 be a Turing machine for which $\mathcal{L}_{T_1}$ has property P. For a given Turing machine S, let us make a machine T as follows: for input x, first it simulates S on the empty input. When the simulated S stops it simulates T_1 on input x. Thus, if S does not halt on the empty input then T does not halt on any input, so $\mathcal{L}_T$ is the empty language. If S halts on the empty input then T halts on exactly the same inputs as T_1, and thus $\mathcal{L}_T = \mathcal{L}_{T_1}$. Thus if we could decide whether $\mathcal{L}_T$ has property P we could also decide whether S halts on empty input. □

2.2 Other undecidable problems

All undecidable problems discussed so far concerned Turing machines; and their undecidability could be attributed to the fact that we were trying to decide something about Turing machines using Turing machines. Could it be that if we get away from this dangerous area of self-referencing, all problems that we want to decide will be decidable?

Unfortunately, the answer is negative, and there are quite "normal" questions, arising in all sorts of mathematical areas, that are algorithmically undecidable. In this section we describe a few of these, and prove the undecidability of the first one. (For the others, the proof of undecidability is too involved to be included in this book.)

First we discuss a problem of geometrical nature. A *tile*, or *domino*, is a square, divided into four triangles by its two diagonals, such that each of these triangles is colored with some color. A *kit* is a finite set of different tiles, one of which is a distinguished "favorite tile". We have an infinite supply of each tile in the kit.

Given a kit K, a *tiling* of the whole plane with K (if it exists) assigns to each position with integer coordinates a tile which is a copy of a tile in K, in such a way that

- neighboring dominoes have the same color on their adjacent sides;
- the favorite domino occurs at least once.

(Note that rotation is forbidden, otherwise we could tile the whole plane with our favorite domino alone.)

Problem 2.2.1. *Given a kit K, can we tile the whole plane with K?*

It is easy to give a kit of dominoes with which the plane can be tiled (e.g., a single square that has the same color on each side) and also a kit with which this is impossible (e.g., a single square that has different colors on each side). It is, however, a surprising fact that it is algorithmically undecidable whether a kit allows the tiling of the whole plane!

For the exact formulation, let us describe each kit by a word over $\Sigma_0 = \{0, 1, +\}$. This can be done, for example, by describing each color by a positive integer in binary

notation, describing each tile in the kit by writing down the numbers representing the colors of the four triangles, separated by the symbol "+", beginning at the top side, clockwise, and then we join the expressions obtained this way with "+" signs, starting with the favorite domino. (But the details of this encoding are of course not important.) Let $\mathcal{L}_{\text{TLNG}}$ [resp. $\mathcal{L}_{\text{NTLNG}}$] be the set of codes of those kits which tile the plane [resp. do not tile the plane].

Theorem 2.2.1. *The tiling problem is undecidable, i.e., the language $\mathcal{L}_{\text{TLNG}}$ is not recursive.*

Accepting, for the moment this statement, we can conclude by Theorem 2.1.3 that either the tiling or the nontiling kits must form a language that is not recursively enumerable. Which one? At first look, we might think that $\mathcal{L}_{\text{TLNG}}$ is recursively enumerable: the fact that the plane is tileable by a kit can be "proved" by exhibiting the tiling. This is, however, not a finite proof, and in fact the truth is just the opposite:

Theorem 2.2.2. *The language $\mathcal{L}_{\text{NTLNG}}$ is recursively enumerable.*

Taken together with Theorem 2.2.1, we see that $\mathcal{L}_{\text{TLNG}}$ can not even be recursively enumerable.

In the proof of Theorem 2.2.2, the following lemma will play an important role.

Lemma 2.2.3. *The plane can be tiled by a kit if and only if for every natural number n the $(2n+1) \times (2n+1)$ square can be tiled with the favorite tile in the center.*

Proof. The "only if" part of the statement is trivial. For the proof of the "if" part, consider a sequence $N_1, N_2, \ldots$ of tilings of squares such that they all have odd sidelength and their sidelength tends to infinity. We will construct a tiling of the whole plane. Without loss of generality, we may assume that the center of each square is at the origin, so that the origin is covered by the favorite tile.

Let us consider the 3×3 square centered at the origin. This is tiled in some way in each N_i. Since it can only be tiled in finite number of ways, there is an infinite number of tilings N_i in which it is tiled in the same way. Let us keep only these tilings and throw away the rest. So we get an infinite sequence of tilings of larger and larger squares such that the 3×3 square in the middle is tiled in the same way in each N_i. These nine tiles can now be fixed.

Proceeding in a similar fashion, assume that the sequence has been "thinned out" so that in every remaining tiling N_i, the $(2k+1) \times (2k+1)$ square centered at the origin is tiled in the same way. We fix these $(2k+1)^2$ tiles. Then in the remaining tilings N_i, the $(2k+3) \times (2k+3)$ square centered at the origin is tiled only in a finite number of ways, and therefore one of these tilings occurs an infinite number of times. If we keep only these tilings N_i, then every remaining tiling tiles the $(2k+3) \times (2k+3)$ square centered at the origin in the same way, and this tiling contains the tiles fixed previously. Now we can fix the new tiles around the perimeter of the bigger square.

Every tile will be fixed sooner or later, i.e., we have obtained a tiling of the whole plane. Since the condition imposed on the covering is "local", i.e., it refers only to two adjacent dominoes, the tiles will be correctly matched in the final tiling, too. □

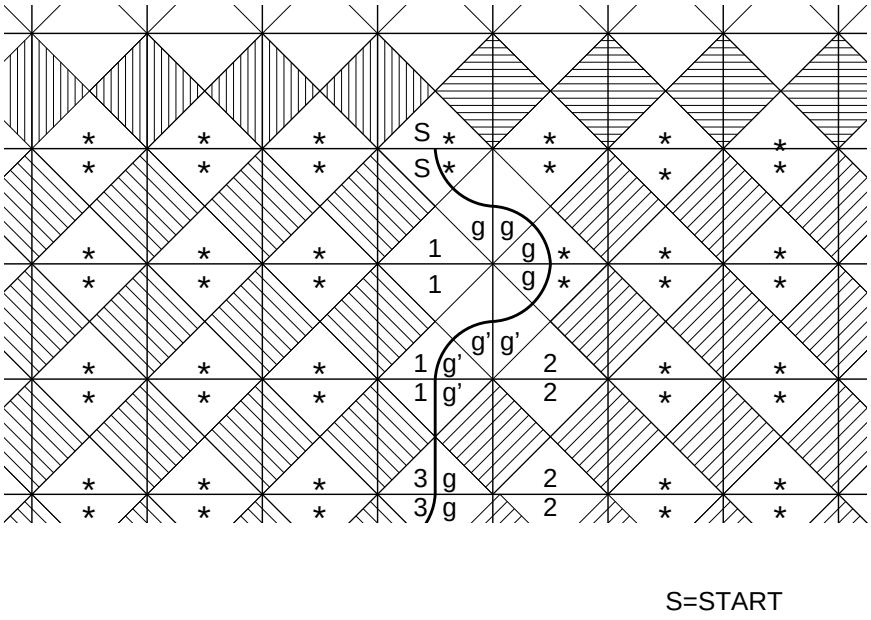

Figure 2.1: Tiling resulting from a particular computation

Proof of Theorem 2.2.2 Let us construct a Turing machine that does the following. For a word $x \in \Sigma_0^*$, it first of all decides whether the word encodes a kit (this is easy); if not then it goes into an infinite cycle. If yes, then with this set, it tries to tile the squares 1×1, 3×3, 5×5, ... with the favorite tile in the center. For each concrete square, it is decidable in a finite number of steps, whether it can be tiled, since the sides can only be numbered in finitely many ways by the numbers occurring in the kit, and it is easy to verify whether among the tilings obtained this way there is one for which every tile comes from the given kit. If the machine finds a square that cannot be tiled by the given kit then it halts.

It is obvious that if $x \notin \mathcal{L}_{\text{NTLNG}}$, i.e., x either does not code a kit or codes a kit which tiles the plane then this Turing machine does not stop. On the other hand, if $x \in \mathcal{L}_{\text{NTLNG}}$, i.e., x codes a kit that does not tile the plane then according to Lemma 2.2.3, for a large enough k already the square $(2k+1) \times (2k+1)$ is not tileable either, and therefore the Turing machine stops after finitely many steps. Thus, according to Lemma 2.1.1, the language $\mathcal{L}_{\text{NTLNG}}$ is recursively enumerable. □

Proof of Theorem 2.2.1 Let $T = \langle k, \Sigma, \alpha, \beta, \gamma \rangle$ be an arbitrary Turing machine; we will construct from it (using its description) a kit K which can tile the plane if and only if T does not stop on the empty input. This is, however, undecidable due to Corollary 2.1.6, so it is also undecidable whether the constructed kit can tile the plane.

For simplicity, assume $k = 1$. Assume that T does not halt on the empty input.

From the machine's computation, we construct a tiling (see Figure 2.1). We lay down the tape before the first step, to get an infinite strip of squares; we put below it the tape before the second step, below it the time before the third step, etc. We can think of the upper half of the plane as filled by empty tapes.

Now we color each square as follows. Each symbol in Σ as well as each state of the control unit corresponds to a color. Moreover, we need colors for pairs gh, where $g \in \Gamma$ and $h \in \Sigma$. Finally, there are five additional colors, which we call WHITE, RIGHT, LEFT, Right, and Left.

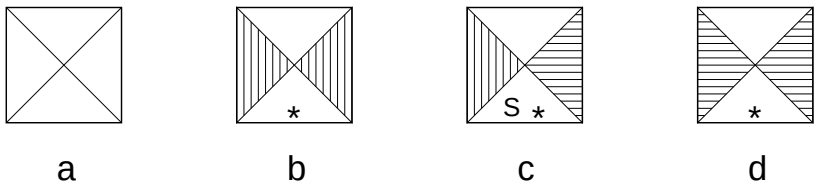

Figure 2.2: Tiles in line 0 (b-d) and above (a). S stands for the START state, vertical strips for color Left and horizontal strips for color Right. Tile c) is the favorite tile.

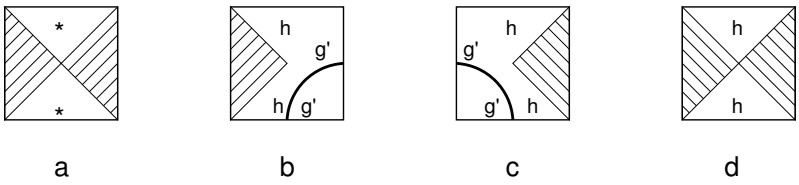

Figure 2.3: Tiles for cells (a) left of the head, (b) just entered from the right, (c) just entered from the left, (d) right of the head.

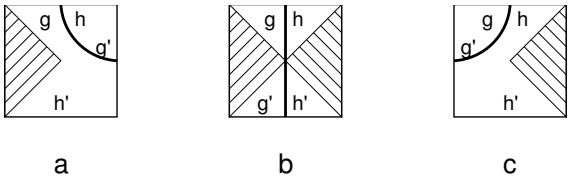

Figure 2.4: The most important tiles: the head reading h and in state g, writes h', goes to state g', and (a) moves right, (b) stays, (c) moves left.

If the content of a cell is a symbol h, and the head is not in this cell before the corresponding step, then we color the bottom triangle of the cell, as well as the top triangle of the cell below it with color h. If the head scans this cell before a given step, and the control unit is in state g, then we color the top triangle of the square as well as the bottom triangle of the square above it with color gh. If at this step, the head moves from one cell to an adjacent cell, we color the two triangles adjacent to the edge crossed with color g. All triangles in this row adjacent to vertical edges to the left of the head are colored LEFT, and those adjacent to vertical edges to the right of the head are colored RIGHT, except in row 0, where instead the colors Left and Right are used, and in rows with negative index, where color WHITE is used. Also WHITE is the color of all upper triangles in row 0 and the tile with the START in the bottom triangle is the favorite tile. See Figures 2.2, 2.3 and 2.4 for all kinds of tiles that occur in this tiling. Note that this kit is constructed from the description of the Turing machine easily; in fact, only one type is dependent on the transition rules of the machine.

We thus obtain a kit K_T, with a favorite tile as marked. Our construction shows that if T runs for an infinite number of steps on the empty input, then the plane can be tiled with this kit. Conversely, assume that the plane can be tiled with the kit K_T. The favorite tile must occur by definition; left and right we can only continue with colors Left and Right, and above, with all-white. Moving down row-by-row we can see that the covering is unique and corresponds to a computation of machine T on empty input.

Since we have covered the whole plane, this computation is infinite. □

Exercise 2.2.1. *Show that there is a kit of dominoes with the property that it tiles the plane but does not tile it doubly periodically (which means that for some linearly independent integer vectors (p, q) and (r, s) we have the same domino at (x, y) as at $(x + p, y + q)$ and at $(x + r, y + s)$).*

Exercise 2.2.2. *Prove that the kits of dominoes that tile the plane doubly periodically are recursively enumerable.*

Exercise 2.2.3. *Prove the following.*

a) There is a function $F : \mathbb{Z}_+ \to \mathbb{Z}_+$ for which if the length of the code of a kit is n and the $(2F(n) + 1) \times (2F(n) + 1)$ square can be tiled with the kit such that the favorite domino is in the center, then the whole plane can be tiled.

b) Such an F cannot be recursive.

Remark. The tiling problem is undecidable even if we do not distinguish an initial domino. But the proof of this result is much harder.

We mention some more algorithmically undecidable problems without showing the proof of undecidability. The proof is in each case a complicated encoding of the halting problem into the problem at hand.

In 1900, Hilbert formulated 23 problems that he considered then the most exciting in mathematics. These problems had a great effect on the development of the mathematics of the century. (It is interesting to note that Hilbert thought some of his problems will resist science for centuries; until today, essentially all of them are solved.) One of these problems was the following.

Problem 2.2.2 (Diophantine equation)**.** *Given a polynomial $p(x_1, \ldots, x_n)$ with integer coefficients and n variables, decide whether the equation $p = 0$ has integer solutions.*

In Hilbert's time, the notion of algorithms was not formalized but he thought that a universally acceptable and always executable procedure could eventually be found that decides for every Diophantine equation whether it is solvable. After the clarification of the notion of algorithms and the finding of the first algorithmically undecidable problems, it became more probable that this problem is algorithmically undecidable. Davis, Robinson and Myhill reduced this conjecture to a specific problem of number theory which was eventually solved by Matiyasevich in 1970. It was found therefore that the problem of solvability of Diophantine equations is algorithmically undecidable.

Next, an important problem from algebra. Let us be given n symbols: $a_1, \ldots, a_n$. The *free group* generated from these symbols is the set of all finite words formed from

the symbols $a_1, \ldots, a_n$, $a_1^{-1}, \ldots, a_n^{-1}$ in which the symbols a_i and a_i^{-1} never follow each other (in any order). We multiply two such words by writing them after each other and repeatedly erasing any pair of the form $a_i a_i^{-1}$ or $a_i^{-1} a_i$ whenever they occur. It takes some, but not difficult, reasoning to show that the multiplication defined this way is associative. We also permit the empty word, this will be the unit element of the group. If we reverse a word and change all symbols a_i in it to a_i^{-1} and vice versa then we obtain the inverse of the word. In this very simple structure, the following problem is algorithmically undecidable.

Problem 2.2.3 (Word problem of groups)**.** *In the free group generated by the symbols $a_1, \ldots, a_n$, we are given $n+1$ words: $\alpha_1, \ldots, \alpha_n$ and β. Is β in the subgroup generated by $\alpha_1, \ldots, \alpha_n$?*

Next, a problem from the field of topology. Let $e_1, \ldots, e_n$ be the unit vectors of the n-dimensional Euclidean space. The convex hull of the points $0, e_1, \ldots, e_n$ is called the *standard* simplex. The *faces* of this simplex are the convex hulls of subsets of the set $\{0, e_1, \ldots, e_n\}$. A *polyhedron* is the union of an arbitrary set of faces of the standard simplex. Here is a fundamental topological problem concerning a polyhedron P:

Problem 2.2.4 (contractibility of polyhedra)**.** *Can a given polyhedron be contracted to a single point (continuously, staying within itself)?*

We define this more precisely, as follows: we mark a point p in the polyhedron first and want to move each point of the polyhedron in such a way within the polyhedron (say, from time 0 to time 1) that it will finally slide into point p and during this, the polyhedron "is not torn". Let $F(x,t)$ denote the position of point x at time t for $0 \leq t \leq 1$. The mapping $F : P \times [0,1] \to P$ is thus continuous (in both of its variables together), having $F(x,0) = x$ and $F(x,1) = p$ for all x. If there is such an F then we say that P is "contractible". For example, a triangle, taken with the area inside it, is contractible. The perimeter of the triangle (the union of the three sides without the interior) is not contractible. (In general, we could say that a polyhedron is contractible if no matter how a thin circular rubber band is tied on it, it is possible to slide this rubber band to a single point.) The property of contractibility turns out to be algorithmically undecidable.

Finally one more, seemingly simple problem that is undecidable.

Problem 2.2.5 (Post correspondence problem)**.** *Given two finite lists, (u_i, v_i); $1 \leq i \leq N$, where $u_i \in \Sigma_0^*$ and $v_i \in \Sigma_0^*$, is there a sequence $i_1, i_2, \ldots, i_K$ such that $u_{i_1} u_{i_2} \ldots u_{i_K} = v_{i_1} v_{i_2} \ldots v_{i_K}$?*

2.3 Computability in logic

2.3.1 Godel's incompleteness theorem

Mathematicians have long held the conviction that a mathematical proof, when written out in all detail, can be checked unambiguously. Aristotle made an attempt to formalize the rules of deduction, but the correct formalism was found only by Frege and Russell

at the end of the nineteenth century. It was championed as a sufficient foundation of mathematics by Hilbert. We try to give an overview of the most important results concerning decidability in logic.

Mathematics deals with *sentences*, statements about some mathematical objects. All sentences will be strings in some finite alphabet. We will always assume that the set of sentences (sometimes also called a *language*) is decidable: it should be easy to distinguish (formally) meaningful sentences from nonsense. Let us also agree that there is an algorithm computing from each sentence φ, another sentence ψ called its *negation*.

A *proof* of some sentence T is a finite string P that is proposed as an argument that T is true. A *formal system*, or *theory* $\mathbf{F}$ is an algorithm to decide, for any pairs (P, T) of strings whether P is an acceptable proof T. A sentence T for which there is a proof in $\mathbf{F}$ is called a *theorem* of the theory $\mathbf{F}$.

Example 2.3.1. Let L_1 be the language consisting of all expressions of the form "$l(a, b)$" and "$l'(a, b)$" where a, b are natural numbers (in their usual, decimal representation). The sentences $l(a, b)$ and $l'(a, b)$ are each other's negations. Here is a simple theory T_1. Let us call *axioms* all "$l(a, b)$" where $b = a + 1$. A *proof* is a sequence $S_1, \ldots, S_n$ of sentences with the following property. If S_i is in the sequence then either it is an axiom or there are $j, k < i$ and integers a, b, c such that $S_j =$"$l(a, b)$", $S_k =$"$l(b, c)$" and $S_i =$"$l(a, c)$". This theory has a proof for all formulas of the form $l(a, b)$ where $a < b$.

A theory is called *consistent* if for no sentence can both the sentence itself and its negation be a theorem. Inconsistent theories are uninteresting, but sometimes we do not know whether a theory is consistent.

A sentence S is called *undecidable* in a theory $\mathcal{T}$ if neither S nor its negation is a theorem in $\mathcal{T}$. A consistent theory is *complete* if it has no undecidable sentences.

The toy theory of Example 2.3.1 is incomplete since it will have no proof of either $l(5, 3)$ nor $l'(5, 3)$. But it is easy to make it complete e.g., by adding as axioms all formulas of the form $l'(a, b)$ where $a \geq b$ are natural numbers and adding the corresponding proofs.

Incompleteness simply means that the theory formulates only certain properties of a kind of system: other properties depend exactly on which system we are considering. Completeness is therefore not always even a desirable goal for certain theories. It is, however, if the goal of our theory is to describe a certain system as completely as we can. We may want e.g., to have a complete theory of the set of natural numbers in which all true sentences have proofs. Also, complete theories have the following desirable algorithmic property, as shown by the theorem below. If there are no (logically) undecidable sentences in a theory then the truth of all sentences (with respect to that theory) is algorithmically decidable.

Theorem 2.3.1. *If a theory $\mathcal{T}$ is complete then there is an algorithm that for each sentence S finds in $\mathcal{T}$ a proof either for S or for the negation of S.*

Proof. The algorithm starts enumerating all possible finite strings P and checking whether P is a proof for S or a proof for the negation of S. Sooner or later, one of the proofs must turn up, since it exists. Consistency implies that if one turns up the other does not exist. □

Suppose that we want to develop a complete theory of natural numbers. Since all sentences about strings, tables, etc. can be encoded into sentences about natural numbers this theory must express all statements about such things as well. In this way, in the language of natural numbers, one can even speak about Turing machines, and about when a Turing machine halts.

Let L be some fixed recursively enumerable set of integers that is not recursive. An arithmetical theory $\mathcal{T}$ is called *minimally adequate* if for every number n, the theory contains a sentence φ_n expressing the statement "$n \in L$"; moreover, this statement is a theorem in $\mathcal{T}$ if and only if it is true.

It is reasonable to expect that a theory of natural numbers with a goal of completeness be minimally adequate, i.e., that it should provide proofs for at least those facts that are verifiable anyway directly by computation, as "$n \in L$" indeed is. (In the next section, we will describe a minimally adequate theory.) Now we are in a position to prove one of the most famous theorems of mathematics, which has not ceased to exert its fascination on people with philosophical interests:

Theorem 2.3.2 (Gödel's incompleteness theorem)**.** *Every minimally adequate theory is incomplete.*

Proof. If the theory were complete then, according to Theorem 2.3.1 it would give a procedure to decide all sentences of the form $n \in L$, which is impossible since L is not recursive. □

Remarks. 1. Looking more closely into the last proof, we see that for any adequate theory $\mathcal{T}$ there is a natural number n such that though the sentence "$n \notin L$" is expressible in $\mathcal{T}$ and true but is not provable in $\mathcal{T}$. There are other, more interesting sentences that are not provable, if only the theory $\mathcal{T}$ is assumed strong enough: Gödel proved that the assertion of the consistency of $\mathcal{T}$ is among these. This so-called Second Incompleteness Theorem of Gödel is beyond our scope.

2. Historically, Gödel's theorems preceded the notion of computability by 3-4 years.

2.3.2 First-order logic

Formulas Let us develop the formal system found most adequate to describe mathematics. A first-order language uses the following symbols:

- An infinite supply of variables: $x, y, z, x_1, x_2, \ldots$, to denote elements of the universe (the set of objects) to which the language refers.

- Some function symbols like $f, g, h, +, \cdot, f_1, f_2, \ldots$, where each function symbol has a property called "arity" specifying the number of arguments of the function it will represent. A function of arity 0 is called a *constant*. It refers to some fixed element of the universe. Some functions, like $+, \cdot$ are used in infix notation.

- Some predicate symbols like $<, >, \subset, \supset, P, Q, R, P_1, P_2, \ldots$, also of different arities. A predicate symbol with arity 0 is also called a *propositional symbol*. Some predicate symbols, like $<$, are used with infix notation. The *equality "="* is a distinguished predicate symbol.

- Logical connectives: $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \ldots$.
- Quantifiers: $\forall, \exists$.
- Punctuation: (,).

A *term* is obtained by taking some constants and variables and applying function symbols to them a finite number of times: e.g., $(x+2)+y$ or $f(f(x,y),g(c))$ are terms (here, 2 is a constant).

An *atomic formula* has the form $P(t_1,\ldots,t_k)$ where P is a predicate symbol and t_i are terms: e.g., $x+y<(x\cdot x)+1$ is an atomic formula.

A *formula* is formed from atomic formulas by applying repeatedly the Boolean operations and the adding of prefixes of the form $\forall x$ and $\exists x$: e.g., $\forall x(x<y)\Rightarrow\exists z g(c,z)$ or $x=x\vee y=y$ are formulas. In the formula $\exists y(\forall x(F)\Rightarrow G)$, the subformula F is called the *scope* of the x-quantifier. An occurrence of a variable x in a formula is said to be *bound* if it is in the scope of an x-quantifier; otherwise the occurrence is said to be *free*. A formula with no free (occurrences of) variables is said to be a *sentence*; sentences make formulas which under any given "interpretation" of the language, are either true of false.

Let us say that a term t is *substitutable* for variable x in formula A if no variable y occurs in t for which some free occurrence of x in A is in the scope of some quantifier of y. If t is substitutable for x in A then we write $A[t/x]$ for the result of substituting t into every free occurrence of x in A: e.g., if $A=(x<3-x)$ and $t=(y^2)$ then $A[t/x]=(y^2<3-y^2)$.

From now on, all our formal systems are some language of first-order logic, so they only differ in what function symbols and predicate symbols are present.

There are natural ways to give *interpretations* to all terms and formulas of a first-order language in such a way that under such an interpretation, all sentences become true or false. This interpretation introduces a set called the *universe* and assigns functions and predicates over this universe to the functions (and constants) and predicates of the language.

Example 2.3.2. Consider the language with constants c_0, c_1 and the two-argument function symbol f. In one interpretation, the universe is the set of natural numbers, $c_0=0, c_1=1$, $f(a,b)=a+b$. In another interpretation, the universe is $\{0,1\}$, $c_0=0$, $c_1=1$, $f(a,b)=a\cdot b$. There are certain sentences that are true in both of these interpretations but not in all possible ones: such is $\forall x\forall y\, f(x,y)=f(y,x)$.

For a given theory T, an interpretation of its language is called a *model* of T if the axioms (and thus all theorems) of the theory are true in it. In the above Example 2.3.2, both interpretations are models of the theory T_1 defined by the single axiom $\forall x\forall y\, f(x,y)=f(y,x)$.

It has been recognized long ago that the proof checking algorithm can be made independent of the theory: theories are different only in their axioms. The algorithm is exactly what we mean by "pure logical reasoning"; for first order logic, it was first formalized in the book Principia Mathematica by Russell and Whitehead at the beginning of the 20th century. We will outline one such algorithm at the end of the present section. Gödel proved in 1930 that if $\mathcal{B}$ implies T in all interpretations of the sentences

then there is a proof of the Principia Mathematica type for it. The following theorem is a consequence.

Theorem 2.3.3 (Gödel's completeness theorem)**.** *Let $\mathcal{P}$ be the set of all pairs $(\mathcal{B}, T)$ where $\mathcal{B}$ is a finite set of sentences and T is a sentence that is true in all interpretations in which the elements of $\mathcal{B}$ are true. The set $\mathcal{P}$ is recursively enumerable.*

Tarski proved that the algebraic theory of real numbers (and with it, all Euclidean geometry) is complete. This is in contrast to the theories of natural numbers, among which the minimally adequate ones are incomplete. (In the algebraic theory of real numbers, we cannot speak of an "arbitrary integer", only of an "arbitrary real number".) Theorem 2.3.1 implies that there is an algorithm to decide the truth of an arbitrary algebraic sentence on real numbers. The known algorithms for doing this take a very long time, but are improving.

Proofs: A *proof* is a sequence $F_1, \ldots, F_n$ of formulas in which each formula is either an axiom or is obtained from previous formulas in the sequence using one of the rules given below. In these rules, A, B, C are arbitrary formulas, and x is an arbitrary variable.

There is an infinite number of formulas that we will require to be part of the set of axioms of each theory: these are therefore called *logical axioms*. These will not all necessarily be sentences: they may contain free variables. To give the axioms, some more notions must be defined.

Let $F(X_1, \ldots, X_n)$ be a Boolean formula of the variables $X_1, \ldots, X_n$, with the property that it gives the value 1 for all possible substitutions of 0 or 1 into $X_1, \ldots, X_n$. Let $\varphi_1, \ldots, \varphi_n$ be arbitrary formulas. Formulas of the kind $F(\varphi_1, \ldots, \varphi_n)$ are called *tautologies*.

The logical axioms of our system consist of the following groups:

Tautologies: All tautologies are axioms.

Equality axioms: Let $t_1, \ldots, t_n$, $u_1, \ldots, u_n$ be terms, f a function symbol and P a predicate symbol, of arity n. Then

$$(t_1 = u_1 \wedge \cdots \wedge t_n = u_n) \Rightarrow f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n),$$
$$(t_1 = u_1 \wedge \cdots \wedge t_n = u_n) \Rightarrow (P(t_1, \ldots, t_n) \Leftrightarrow P(u_1, \ldots, u_n))$$

are axioms.

The definition of $\exists$: For all formulas A and variables x, the formula $\exists x\, A \Leftrightarrow \neg\forall x\, \neg A$ is an axiom.

Specialization: If term t is substitutable for variable x in formula A then $\forall x\, A \Rightarrow A[t/x]$ is an axiom.

The system has two rules:

Modus ponens: From $A \Rightarrow B$ and $B \Rightarrow C$, we can derive $A \Rightarrow C$.

Generalization: If the variable x does not occur free in A then from $A \Rightarrow B$ we can derive $A \Rightarrow \forall x\, B$.

Remark. The generalization rule says that if we can derive a statement B containing the variable x without using any properties of x in our assumptions then it is true for arbitrary values of x. It does *not* say that $B \Rightarrow \forall x B$ is true.

For the system above, the following stronger form of Gödel's completeness theorem holds.

Theorem 2.3.4. *Suppose that $\mathcal{B}$ is a set of sentences and T is a sentence that is true in all interpretations in which the elements of $\mathcal{B}$ are true. Then there is a proof of T in the proof system if we add the sentences of $\mathcal{B}$ to the axioms.*

A simple theory of arithmetic and Church's Theorem This theory N contains two constants, 0 and 1, the function symbols $+, \cdot$ and the predicate symbol $<$. There is only a finite number of simple nonlogical axioms (all of them without quantifier).

$$\begin{aligned}
\neg(x+1) &= 0.\\
1+x=1+y &\Rightarrow x=y.\\
x+0 &= x.\\
x+(1+y) &= 1+(x+y).\\
x\cdot 0 &= 0.\\
x\cdot(1+y) &= (x\cdot y)+x.\\
\neg(x &< 0).\\
x<(1+y) &\Leftrightarrow x<y\vee x=y.\\
x<y\vee x=y\vee y<x.
\end{aligned}$$

Theorem 2.3.5. *The theory N is minimally adequate. Thus, there is a minimally adequate consistent theory of arithmetic with a finite system of axioms.*

This fact implies the following theorem of Church, showing that the problem of logical provability is algorithmically undecidable.

Theorem 2.3.6 (Undecidability Theorem of Predicate Calculus)**.** *The set $\mathcal{P}$ of all sentences that can be proven without any axioms, is undecidable.*

Proof. Let $\mathcal{N}$ be a finite system of axioms of a minimally adequate consistent theory of arithmetic, and let N be the sentence obtained by taking the conjunction of all these axioms and applying universal quantification. Let us remember the definition of "minimally adequate": we used there a nonrecursive r.e. set L of natural numbers. In arithmetic, we can write down a formula $Q(n)$ saying $N \Rightarrow (n \in L)$. There is a proof for "$n \in L$" in $\mathcal{N}$ if and only if there is a proof for $Q(n)$ from the empty axiom system. But from the remark after Theorem 2.3.2 it follows that there is an n for which "$n \in L$" is not provable in $\mathcal{N}$, so $Q(n)$ is also not provable from the empty axiom system. So if

we had a decision procedure for $\mathcal{P}$ we could decide, $Q(n)$; since we cannot, there is no decision procedure for $\mathcal{P}$. □

Exercise 2.3.1. *Prove that a function is recursive if and only if its graph* $\{ (x, f(x)) : x \in \Sigma_0^* \}$ *is recursively enumerable.*

Exercise 2.3.2. *(a) Prove that a language is recursively enumerable if and only if it can be enumerated without repetition by some Turing machine.*
(b) Prove that a language is recursive if and only if it can be enumerated in increasing order *by some Turing machine.*

Exercise 2.3.3. *(a) Construct a language that is not recursively enumerable.*

(b) Construct a language that is neither recursive nor recursively enumerable.

In the exercises below, we will sometimes use the following notion. A function f defined on a subset of Σ_0^* is called *partial recursive* (abbreviated as p.r.) if there exists a Turing machine that for any input $x \in \Sigma_0^*$ will stop after finite time if and only if $f(x)$ is defined and in this case, it will have $f(x)$ written on its first tape.

Exercise 2.3.4. *Let us call two Turing machines equivalent if for all inputs, they give the same outputs. Let the function* $f : \Sigma_0^* \to \{0, 1\}$ *be 1 if* p, q *are codes of equivalent Turing machines and 0 otherwise. Prove that* f *is undecidable.*

Exercise 2.3.5. *(Inseparability Theorem.) Let* U *be a one-tape Turing machine simulating the universal two-tape Turing machine. Let* $u'(x)$ *be 0 if the first symbol of the value computed on input* x *is 0, and 1 if* U *halts but this first symbol is not 0. Then* u' *is a partial recursive function, defined for those* x *on which* U *halts. Prove that there is no computable total function which is an extension of the function* $u'(x)$*. In particular, the two disjoint r.e. sets defined by the conditions* $u' = 0$ *and* $u' = 1$ *cannot be enclosed into disjoint recursive sets.*

Exercise 2.3.6. *(Nonrecursive function with recursive graph.) Give a p.r. function* f *that is not extendable to a recursive function, and whose graph is recursive.*

[Hint: use the running time of the universal Turing machine.]

Exercise 2.3.7. *Construct an undecidable, recursively enumerable set* B *of pairs of natural numbers with the property that for all* x*, the set* $\{ y : (x, y) \in B \}$ *is decidable, and at the same time, for all* y*, the set* $\{ x : (x, y) \in B \}$ *is decidable.*

Exercise 2.3.8. *Construct an undecidable set* S *of natural numbers such that*

$$\lim_{n\to\infty} \frac{1}{n} |S \cap \{0, 1, \ldots, n\}| = 0.$$

Can you construct an undecidable set for which the same limit is 1*?*

Exercise 2.3.9. *A rooted tree is a set of "nodes" in which each node has some "children", the single "root" node has no parent and each other node has a unique parent. A path is a sequence of nodes in which each node is the parent of the next one. Suppose that each node has only finitely many children and the tree is infinite. Prove that then the tree has an infinite path.*

Exercise 2.3.10. *Consider a Turing machine* T *which we allow now to be used in the following nonstandard manner: in the initial configuration, it is not required that*

the number of nonblank symbols be finite. Suppose that T halts for all possible initial configurations of the tape. Prove that then there is an n such that for all initial configurations, on all tapes, the heads of T stay within distance n of the origin.

Exercise 2.3.11. *Let the partial function $f_T(n)$ be defined if T, started with the empty tape, will ever write a nonblank symbol in cell n; in this case, let it be the first such symbol. Prove that there is a T for which $f_T(n)$ cannot be extended to a recursive function.*

Exercise* 2.3.12. *Show that there is a kit of dominoes with the property that it tiles the plane but does not tile it recursively.*

[Hint: Take the Turing machine of Exercise 2.3.11. Use the kit assigned to it by the proof of the tiling problem. We will only consider "prototiles" associated with the lower half-plane. We turn each of these prototiles into several others by writing a second tape symbol on both the top edge and the bottom edge of each prototile P in the following way. If the tape symbol of both the top and the bottom of P is $*$ or both are different from $*$ then for all symbols h in Σ_0, we make a new prototile P_h by adding h to both the top and the bottom of P. If the bottom of P has $*$ and the top has a nonblank tape symbol h then we make a new prototile P' by adding h to both the top and the bottom. The new kit for the upper half-plane consists of all prototiles of the form P_h and P'.]

Exercise 2.3.13. *Let us consider the following modifications of the tiling problem.*

- *In P_1, tiles are allowed to be rotated 180 degrees.*
- *In P_2, flipping around the vertical axis is allowed.*
- *In P_3, flipping around the main diagonal axis is allowed.*

Prove that there is always a tiling for P_1, the problem P_2 is decidable and problem P_3 is undecidable.

Exercise 2.3.14. *Show that the following modification of the tiling problem is also undecidable. We use tiles marked on the corners instead of the sides and all tiles meeting in a corner must have the same mark.*

Exercise 2.3.15. *Our proof of Gödel's theorem does not seem to give a specific sentence φ_T undecidable for a given minimally adequate theory T. Show that such a sentence can be constructed, if the language L used in the definition of "minimally adequate" is obtained by any standard coding from the nonrecursive r.e. set constructed in the proof of the undecidablity of the halting problem.*

Chapter 3

Computation with resource bounds

The algorithmic solvability of some problems can be very far from their *practical* solvability. There are algorithmically solvable problems that cannot be solved, for an input of a given size, in fewer than exponentially or doubly exponentially many steps (see Theorem 3.3.3). Complexity theory, a major branch of the theory of algorithms, investigates the solvability of individual problems under certain resource restrictions. The most important resources are *time* and *space* (storage).

We define these notions in terms of the Turing machine model of computation. This definition is suitable for theoretical study; in describing algorithms, using the RAM is more convenient, and it also approximates reality better. It follows, however, from Theorem 1.3.1 and 1.3.2 that from the point of view of the most important types of resource restrictions (e.g. polynomial time and space) it does not matter, which machine model is used in the definition.

This leads us to the definition of various *complexity classes*: classes of problems solvable within given time bounds, depending on the size of the input. Every positive function of the input size defines such a class, but some of them are particularly important. The most central complexity class is *polynomial time*. Many algorithms important in practice run in polynomial time (in short, are polynomial). Polynomial algorithms are often very interesting mathematically, since they are built on deeper insight into the mathematical structure of the problems, and often use strong mathematical tools.

We restrict the computational tasks to yes-or-no problems; this is not too much of a restriction, and pays off in what we gain in simplicity of presentation. Note that the task of computing any output can be broken down to computing its bits in any reasonable binary representation.

Most of this chapter is spent on illustrating how certain computational tasks can be solved within given resource constraints. We start with the most important case, and show that most of the basic everyday computational tasks can be solved in polynomial time. These basic tasks include tasks in number theory (arithmetic operations, greatest common divisor, modular arithmetic) linear algebra (Gaussian elimination) and graph theory. (We cannot in any sense survey all the basic algorithms, especially in graph theory; we will restrict ourselves to a few that will be needed later.)

Polynomial space is a much more general class than polynomial time (i.e., a much less restrictive resource constraint). The most important computational problems solv-

able in polynomial space (but most probably not in polynomial time) are *games* like chess or Go. We give a detailed description of this connection. We end the chapter with a briefer discussion of other typical complexity classes.

Let us fix some finite alphabet Σ, including the blank symbol $*$ and let $\Sigma_0 = \Sigma \setminus \{*\}$. In this chapter, when a Turing machine is used for computation, we assume that it has an *input tape* that it can only read (it cannot change the symbols of the tape and the head cannot move outwards from the $*$'s delimiting the input) and *output tape* that it can only write and $k \geq 1$ work tapes. At start, there is a word in Σ_0^* written on the input tape.

The *time demand* of a Turing machine T is a function $\mathrm{time}_T(n)$ defined as the maximum of the number of steps taken by T over all possible inputs of length n. We assume $\mathrm{time}_T(n) \geq n$ (the machine must read the input; this is not necessarily so but we exclude only trivial cases with this assumption). It may happen that $\mathrm{time}_T(n) = \infty$.

Similarly, the function $\mathrm{space}_T(n)$ is defined as the maximum number, over all inputs of length n, of all cells on all but the input and output tapes to which the machine writes. Note that writing the same symbol which was read also counts as writing, so this quantity is the number of cells that are visited by the heads (except the ones on the input or output tape).

A Turing machine T is called *polynomial*, if there is a polynomial $f(n)$ such that $\mathrm{time}_T(n) = O(f(n))$. This is equivalent to saying that there is a constant c such that the time demand of T is $O(n^c)$. We say that an algorithm is polynomial if there is a polynomial Turing machine realizing it. We can define exponential Turing machines (resp. algorithms) similarly (for which the time demand is $O(2^{n^c})$ for some $c > 0$), and also Turing machines (resp. algorithms) working in polynomial and exponential space.

Now we consider a yes-or-no problem. This can be formalized as the task of deciding whether the input word x belongs to a fixed language $\mathcal{L} \in \Sigma_0^*$.

We say that a language $\mathcal{L} \in \Sigma_0^*$ has *time complexity at most* $f(n)$, if it can be decided by a Turing machine with time demand at most $f(n)$. We denote by $\mathrm{DTIME}(f(n))$ the class of languages whose time complexity is at most $f(n)$. (The letter "D" indicates that we consider here only deterministic algorithms; later, we will also consider algorithms that are "nondeterministic" or use randomness). We denote by PTIME, or simply by P, the class of all languages decidable by a polynomial Turing machine. We define similarly when a language has *space complexity at most* $f(n)$, and also the language classes $\mathrm{DSPACE}(f(n))$ and PSPACE (polynomial space).

Remarks. 1. It would be tempting to define the time complexity of a language $\mathcal{L}$ as the optimum time of a Turing machine that decides the language. Note that we were more careful above, and only defined when the time complexity is *at most* $f(n)$. The reason is that there may not be a best algorithm (Turing machine) solving a given problem: some algorithms may work better for smaller instances, some others on larger, some others on even larger etc.

2. When we say that the multiplication of two numbers of size n can be performed in time n^2 then we actually find an upper bound on the complexity of a *function* (multiplication of two numbers represented by the input strings) rather than a language. The classes $\mathrm{DTIME}(f(n))$, $\mathrm{DSPACE}(f(n))$, etc. are defined as classes of *languages*; corresponding classes of functions can also be defined.

Sometimes, it is easy to give a trivial lower bound on the complexity of a function.

Consider e.g., the function $x \cdot y$ where x and y are numbers in binary notation. Its computation requires at least $|x| + |y|$ steps, since this is the length of the output. Lower bounds on the complexity of languages are never this trivial, since the output of the computation deciding the language is a single bit.

How to define time on the RAM machine? The number of steps of the Random Access Machine is not the best measure of the "time it takes to work". One could (mis)use the fact that the instructions operate on natural numbers of arbitrary size, and develop computational tricks that work in this model but use such huge integers that to turn them into practical computations would be impossible. For example, we can simulate vector addition by the addition of two very large natural numbers.

Therefore, we prefer to characterize the running time of RAM algorithms by two numbers, and say that "the machine makes at most m steps on numbers with at most k bits". Similarly, the space requirement is best characterized by saying that "the machine stores at most m numbers with at most k bits".

If we want a single number to characterize the running time of a RAM computation, we can count as the time of a step not one unit but the number of bits of the integers involved in it (both register addresses and their contents). Since the number of bits of an integer is essentially base two logarithm of its absolute value, it is also usual to call this model *logarithmic cost* RAM.)

In arithmetical and algebraic algorithms, it is sometimes convenient to count the *arithmetical operations*; on a Random Access Machine, this corresponds to extending the set of basic operations of the programming language to include the subtraction, multiplication, division (with remainder) and comparison of integers, and counting the number of steps instead of the running time. If we perform only a polynomial number of operations (in terms of the length of the input) on numbers with at most a polynomial number of digits, then our algorithm will be polynomial in the logarithmic cost model.

3.1 Polynomial time

Arithmetic operations

All basic arithmetic operations are polynomial: addition, subtraction, multiplication and division of integers with remainder. (Recall that the length of an integer n as input is the number of its bits, i.e., $\log_2 n+O(1)$). We learn polynomial time algorithms for all these operations in elementary school (linear time algorithms in the case of addition and subtraction, quadratic time algorithms in the case of multiplication and division). We also count the comparison of two numbers as a trivial but basic arithmetic operation, and this can also be done in polynomial (linear) time.

A less trivial polynomial time arithmetic algorithm is the *Euclidean algorithm*, computing the greatest common divisor of two numbers.

Euclidean Algorithm. We are given two natural numbers, a and b. Select one that is not larger than the other, let this be a. If $a = 0$ then the greatest common divisor of a and b is $\gcd(a, b) = b$. If $a > 0$ then let us divide b by a, with remainder, and let r be the remainder. Then $\gcd(a, b) = \gcd(a, r)$, and it is enough, therefore, to determine

the greatest common divisor of a and r. Since $r < a$, this recurrence will terminate in a finite number of iterations and we get the greatest common divisor of a and b.

Notice that, strictly speaking, the algorithm given above is not a program for the Random Access Machine. It is a recursive program, and even as such it is given somewhat informally. But we know that such an informal program can be translated into a formal one, and a recursive program can be translated into a machine-language program (most compilers can do that).

Lemma 3.1.1. *The Euclidean algorithm takes polynomial time. More exactly, it carries out* $O(\log a + \log b)$ *arithmetical operations on numbers not exceeding* $\max(a, b)$ *on input* (a, b).

Proof. Since $0 \leq r < a \leq b$, the Euclidean algorithm will terminate sooner or later. Let us see that it terminates in polynomial time. Notice that $b \geq a + r > 2r$ and thus $r < b/2$. Hence $ar < ab/2$. Therefore after $\lceil \log(ab) \rceil$ iterations, the product of the two numbers will be smaller than 1, hence r will be 0, i.e., the algorithm terminates. Each iteration consist of elementary arithmetic operations, and can be carried out in polynomial time. □

It is an important feature of the Euclidean algorithm that it not only gives the value of the greatest common divisor, but also yields integers p, q such that $\gcd(a, b) = pa + qb$. For this, we simply maintain such a form for all numbers computed during the algorithm. If $a' = p_1 a + q_1 b$ and $b' = p_2 a + q_2 b$ and we divide, say, b' by a' with remainder: $b' = ha' + r'$ then

$$r' = (p_2 - hp_1)a + (q_2 - hp_2)b,$$

and thus we obtain the representation of the new number r' in the form $p'a + q'b$.

Exercise 3.1.1. *The Fibonacci numbers are defined by the following recurrence:* $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ *for* $k > 1$. *Let* $1 \leq a \leq b$ *and let* F_k *denote the greatest Fibonacci number not greater than* b. *Prove that the Euclidean algorithm, when applied to the pair* (a, b), *terminates in at most* k *steps. How many steps does the algorithm take when applied to* (F_k, F_{k-1})*?*

Remark. The Euclidean algorithm is sometimes given by the following iteration: if $a = 0$ then we are done. If $a > b$ then let us switch the numbers. If $0 < a \leq b$ then let $b := b - a$. Mathematically, essentially the same thing happens (Euclid's original algorithm was closer to this), this algorithm is *not polynomial*: even the computation of $\gcd(1, b)$ requires b iterations, which is exponentially large in terms of the number $\log b + O(1)$ of the digits of the input.

The operations of addition, subtraction, multiplication can be carried out in polynomial times also in the ring of remainder classes modulo an integer m. We represent the remainder classes by the smallest nonnegative remainder. We carry out the operation on these as on integers; at the end, another division by m, with remainder, is necessary.

If m is a prime number then we can also carry out the *division* in the field of the residue classes modulo m, in polynomial time. (This is different from division with remainder!) For general m we can also carry out division in polynomial time. Let a, b

and m be given integers, such that $0 \leq a, b \leq m-1$ and $gcd(b, m) = 1$. Then the result of a divided by $b \pmod{m}$ is an integer x with $0 \leq x < m$ such that

$$bx \equiv a \pmod{m}.$$

(Such an x is sometimes denoted by $a/b \pmod{m}$.)

The way to find x is to apply the Euclidean algorithm to compute the greatest common divisor of the numbers b and m. Of course, we know in advance that the result is 1. But as remarked, we also obtain integers p and q such that $bp + mq = 1$. In other words, $bp \equiv 1 \pmod{m}$, and thus $b(pa) \equiv a \pmod{m}$. So the quotient x we are looking for is the remainder of the product pa after dividing by m.

We mention yet another application of the Euclidean algorithm. Suppose that a certain integer x is unknown to us but we know its remainders $x_1, \ldots, x_k$ with respect to the moduli $m_1, \ldots, m_k$ which are all relatively prime to each other. The Chinese Remainder Theorem says that these remainders uniquely determine the remainder of x modulo the product $m = m_1 \cdots m_k$. But how can we compute this remainder?

It suffices to deal with the case $k = 2$ since for general k, the algorithm follows from this by mathematical induction. We are looking for an integer x such that $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$. We also want that $0 \leq x \leq m_1 m_2 - 1$, but this we can achieve by dividing with remainder at the end.

In other words, we are looking for integers x, q_1 and q_2 such that $x = x_1 + q_1 m_1$ and $x = x_2 + q_2 m_2$. Subtracting, we get $x_2 - x_1 = q_1 m_1 - q_2 m_2$. This equation does not determine the numbers q_1 and q_2 uniquely, but this is not important. We can find, using the Euclidean algorithm, numbers q_1 and q_2 such that

$$x_2 - x_1 = q_1 m_1 - q_2 m_2,$$

and compute $x = x_1 + q_1 m_1 = x_2 + q_2 m_2$. Then $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$, as desired.

Next, we discuss the operation of exponentiation. Since even to write down the number 2^n, we need an exponential number of digits (in terms of the length of the input, i.e., the number of binary digits of n), so this number is not computable in polynomial time. The situation changes, however, if we want to carry out the exponentiation modulo m: then the residue class of a^b modulo m can be represented by $\log m + O(1)$ bits. We will show that it can be not only represented polynomially but also computed in polynomial time.

Lemma 3.1.2. *Let a, b and m be three natural numbers. Then $a^b \pmod{m}$ can be computed in polynomial time, or more precisely, with $O(\log b)$ arithmetical operations, carried out on natural numbers with $O(\log m + \log a)$ digits.*

Algorithm 3.1.3. Let us write b in binary:

$$b = 2^{r_1} + \cdots + 2^{r_k},$$

where $0 \leq r_1 < \cdots < r_k$. It is obvious that $r_k \leq \log b$ and therefore $k \leq \log b$. Now, the numbers $a^{2^t} \pmod{m}$ for $0 \leq t \leq \log b$ are easily obtained by repeated squaring, and then we multiply those k together that make up a^b. Of course, we carry out all operations modulo m, i.e., after each multiplication, we also perform a division with remainder by m.

Remark. It is not known whether $a!$ mod m or $\binom{a}{b}$ mod m can be computed in polynomial time.

Algorithms in linear algebra

The basic operations of linear algebra are polynomial: addition and inner product of vectors, multiplication and inversion of matrices, and the computation of determinants. However, these facts are non-trivial in the last two cases, so we will deal with them in detail later in Chapter 9.

Let $A = (a_{ij})$ be an arbitrary $n \times n$ matrix consisting of integers.

Let us verify, first of all, that the polynomial computation of $\det(A)$ is not inherently impossible, in the sense that the result can be expressed with polynomially many bits. Let $K = \max |a_{ij}|$, then to write down the matrix A we need obviously at least $L = n^2 + \log K$ bits. On the other hand, the definition of determinants gives

$$|\det(A)| \le n! K^n,$$

hence $\det(A)$ can be written down using

$$\log(n!K^n) + O(1) \le n(\log n + \log K) + O(1)$$

bits. This is polynomial in L.

Linear algebra gives a formula for each element of $\det(A^{-1})$ as the quotient of two subdeterminants of A. This shows that A^{-1} can also be written down with polynomially many bits.

Exercise 3.1.2. *Show that if A is a square matrix consisting of integers, then to write down* $\det(A)$ *we need at most as many bits as to write up A. [Hint: If $a_1, \dots, a_n$ are the row vectors of A then $|\det(A)| \le |a_1| \cdots |a_n|$ (this so-called "Hadamard Inequality" is analogous to the statement that the area of a parallelogram is smaller than the product of the lengths of its sides).]*

The usual procedure to compute the determinant is *Gaussian elimination*. We can view this as the transformation of the matrix into a lower triangular matrix with column operations. These transformations do not change the determinant, and in the final triangular matrix, the computation of the determinant is trivial: we just multiply the diagonal elements to obtain it. It is also easy to obtain the inverse matrix from this form; we will not deal with this issue separately.

Gaussian elimination. Suppose that for all i such that $1 \le i \le t$, we have achieved already that in the i-th row, only the first i entries hold a nonzero element. Pick a nonzero element from the last $n - t$ columns (stop if there is no such element). Call this element the *pivot element* of this stage. Rearrange the rows and columns so that this element gets into position $(t + 1, t + 1)$. Subtract column $t + 1$, multiplied by $a_{t+1,i}/a_{t+1,t+1}$, from column i for all $i = t + 2, \dots, n$, in order to get 0's in the elements $(t+1, t+2), \dots, (t+1, n)$. These subtractions do not change the value of the determinant and the rearrangement changes at most the sign, which is easy to keep track of.

Since one iteration of the Gaussian elimination uses $O(n^2)$ arithmetic operations and n iterations must be performed, this procedure uses $O(n^3)$ arithmetic operations.

But the problem is that we must also divide, and not with remainder. This does not cause a problem over a finite field, but it does in the case of the rational field. We assumed that the elements of the original matrix are integers; but during the run of the algorithm, matrices also occur that consist of rational numbers. In what form should these matrix elements be stored? The natural answer is as pairs of integers (whose quotient is the rational number).

Do we require that the fractions be in simplified form, i.e., that their numerator and denominator be relatively prime to each other? We could do so; then we have to simplify each matrix element after each iteration, for which we would have to perform the Euclidean algorithm. This can be performed in polynomial time, but it is a lot of extra work, and it is desirable to avoid it. (Of course, we also have to show that in the simplified form, the occurring numerators and denominators have only polynomially many digits. This will follow from the discussions below.)

We could also choose not to require that the matrix elements be in simplified form. Then we define the sum and product of two rational numbers a/b and c/d by the following formulas: $(ad + bc)/(bd)$ and $(ac)/(bd)$. With this convention, the problem is that the numerators and denominators occurring in the course of the algorithm can become very large (have a nonpolynomial number of digits)!

Fortunately, we can give a procedure that stores the fractions in partially simplified form, and avoids both the simplification and the excessive growth of the number of digits. For this, let us analyze a little the matrices occurring during Gaussian elimination. We can assume that the pivot elements are, as they come, in positions $(1,1),\dots,(n,n)$, i.e., we do not have to permute the rows and columns. Let $(a_{ij}^{(k)})$ $(1 \le i,j \le n)$ be the matrix obtained after k iterations. Let us denote the elements in the main diagonal of the final matrix, for simplicity, by $d_1,\dots,d_n$ (thus, $d_i = a_{ii}^{(n)}$). Let $D^{(k)}$ denote the submatrix determined by the first k rows and columns of matrix A, and let $D_{ij}^{(k)}$, for $k+1 \le i,j \le n$, denote the submatrix determined by the first k rows and the i-th row and the first k columns and the j-th column. Let $d_{ij}^{(k)} = \det(D_{ij}^{(k)})$. Obviously, $\det(D^{(k)}) = d_{kk}^{(k-1)}$.

Lemma 3.1.4.

$$a_{ij}^{(k)} = \frac{d_{ij}^{(k)}}{\det(D^{(k)})}.$$

Proof. If we compute $\det(D_{ij}^{(k)})$ using Gaussian elimination, then in its main diagonal, we obtain the elements $d_1,\dots,d_k,a_{ij}^{(k)}$. Thus

$$d_{ij}^{(k)} = d_1 \cdots d_k \cdot a_{ij}^{(k)}.$$

Similarly,

$$\det(D^{(k)}) = d_1 \cdots d_k.$$

Dividing these two equations by each other, we obtain the lemma. □

By this lemma, every number occurring in the Gaussian elimination can be represented as a fraction both the numerator and the denominator of which is a determinant of some submatrix of the original A matrix. In this way, a polynomial number of digits is certainly enough to represent all the fractions obtained.

However, it is not necessary to compute the simplifications of all fractions obtained in the process. By the definition of Gaussian elimination we have that

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{i,k+1}^{(k)} a_{k+1,j}^{(k)}}{a_{k+1,k+1}^{(k)}}$$

and hence using Lemma 3.1.4

$$d_{ij}^{(k+1)} = \frac{d_{ij}^{(k)} d_{k+1,k+1}^{(k)} - d_{i,k+1}^{(k)} d_{k+1,j}^{(k)}}{d_{k,k}^{(k-1)}}.$$

This formula can be considered as a recurrence for computing the numbers $d_{ij}^{(k)}$. Since the left-hand side is an integer, the division can be carried out exactly. Using the above considerations, we find that the number of digits in the quotient is polynomial in terms of the size of the input.

Remarks. 1. There are at least two further possibilities to remedy the problem of the fractions occurring in Gaussian elimination.

We can approximate the numbers by binary "decimals" of limited accuracy (as it seems natural from the point of view of computer implementation), allowing, say, p bits after the binary "decimal point". Then the result is only an approximation, but since the determinant is an integer, it is enough to compute it with an error smaller than $1/2$. Using the methods of numerical analysis, it can be determined how large p must be chosen to make the error in the end result smaller than $1/2$. It turns out that a polynomial number of digits are enough, and this also leads to a polynomial algorithm.

The third possibility is based on the remark that if $m > |\det(A)|$ then it is enough to determine the value of $\det(A)$ modulo m. If m is a prime number then computing modulo m, we do not have to use fractions at all. Since we know that $|\det(A)| < n!K^n$ it is enough to choose for m a prime number greater than $n!K^n$.

It is, however, not that simple to select such a large prime (see Chapter 5). An easier method is to choose m as the product of different small primes: $m = 2 \cdot 3 \cdots p_k$ where for k we can choose, e.g., the total number of bits occurring in the representation of A. Then it is easy to compute the remainder of $\det(A)$ modulo p_i for all p_i, using Gaussian elimination in the field of residue classes modulo p_i. Then we can compute the remainder of $\det(A)$ modulo m using the Chinese Remainder Theorem. (Since k is small we can afford to find the first k primes simply by brute force. But the cost of this computation must be judged differently anyway since the same primes can then be used for the computation of arbitrarily many determinants.)

2. The modular method is successfully applicable in a number of other cases. One way to look at this method is to consider it as an encoding of the integers in a way different from the binary (or decimal) number system: we code the integer n by its remainder after division by the primes 2,3, etc. This is an infinite number of bits, but if we know in advance that no number occurring in the computation is larger than N then it is enough to consider the first k primes whose product is larger than N. In this encoding, the arithmetic operations can be performed very simply, and even in parallel for the different primes. Comparison by magnitude is, however, awkward.

Graph algorithms

The most important algorithms of graph theory are polynomial. It is impossible to survey this topic in this course; graph algorithms can fill several books. We restrict ourselves to a brief discussion of a few examples that provide particularly important insight into some basic techniques. We will group our examples around two algorithmic methods: searching a graph and augmenting paths.

How is a graph given?

Perhaps the most natural way to describe a graph is by its *adjacency* matrix. However, if a graph is *sparse*, i.e., it has much less edges than n^2, e.g., $O(n)$ or $O(n \log n)$, then it can be encoded much more efficiently by its *edge list*. In an edge list we simple enumerate for each vertex its neighbors in some order. The space requirement of this representation has the same order of magnitude as the number of edges. The following algorithms work for edge list representations much faster for sparse graphs.

Exercise 3.1.3. *Give a polynomial time algorithm that converts the adjacency matrix into an edge list and vice versa.*

Searching a graph

a) General search and connectivity testing. Perhaps the most fundamental question to ask about a graph is whether or not it is connected. If not, we often want to find its connected components.

These tasks are trivial to do in polynomial time, in a variety of ways. The reason why we describe a solution is because its elements will be useful later.

Let G be a graph. Select any node r. Build up a tree T as follows. At the beginning, T consists of just r. At any stage, we check if there is an edge between the nodes of T and the rest of the nodes. If there is such an edge, we add it to T, together with its endnode outside T. If not, then we know that G is disconnected and T is a spanning tree of a connected component. We can delete the nodes of T from the graph and proceed recursively.

It is customary to call the nodes of T *labeled*. The usual way to look for an edge between a labeled and an unlabeled node is to look at the labeled nodes and investigate the edges going out of them. A simple but important observation is that if at one stage we find that none of the edges of a given labeled node goes to an unlabeled node, then we don't need to investigate this node at any other later stage (since only an unlabeled node can become labeled, not vice versa). Therefore, we can mark this node as taken care of, or *scanned*. At any time, we only need to look at those nodes that are labeled but not scanned.

This general labeling-scanning procedure is called *searching the graph*.

b) Breadth-First-Search and shortest paths. Specializing the order in which we investigate edges going out of the labeled nodes, we get special search algorithms that are good for solving different kinds of graph problems. Perhaps the simplest such specialization is *breadth-first-search*. The problem is to find a shortest path in a graph from a distinguished node s (the source) to a distinguished node t (the sink). The solution of this problem is very simple once we embed it in a larger family of problems;

we ask for the shortest path from the source s to *every* other node of the graph. Proceed recursively: label the neighbors of the source s with a 1. Label with k those unlabeled vertices which are neighbors of vertices of label $k-1$. Then the label of each node is its distance from s. (This method is perhaps the simplest example of a technique called *dynamic programming.*)

This idea reappears in the more general problem of finding a shortest path from s to t in a graph with weighted edges. (The weight c_{ij} on the edge $\{i,j\}$ is interpreted as the length of the edge, so we do not allow negative weights.) Again we embed this problem in the larger family of problems which asks for a shortest (minimum weight) path from s to every other node of the graph.

Dijkstra's algorithm recognizes that a path between two vertices may be of minimum weight even if other paths have fewer edges. So if we begin at the source, among all neighbors j of s, find one such that the edge sj has minimum weight c_{sj}. We can confidently assert that the shortest path in the graph from s to this neighbor has length c_{sj}, but we are not sure about the other neighbors of s. So label this one neighbor c_{sj}. You may think of s as labeled with a 0. In the course of the algorithm we maintain a set T of vertices, each of which we have confidently labeled with the minimum weight of a path from s to it. Call this label $d(s,j)$, the distance from s to the node j. These minimum weight paths pass only through vertices already in T. At each step of the algorithm, we consider vertices on the frontier of T (vertices not in T which are neighbors of vertices in T). Consider edges ij between vertices i in T and vertices j in this frontier. An upper bound for the minimum weight of a path from s to j is the smallest of the numbers $d(s,i)+c_{ij}$ where i is in T. Find $i \in T$ and j which minimize $d(s,i)+c_{ij}$. We can confidently label such a node j with $d(s,j) = d(s,i)+c_{ij}$ and add j to T for the following reason: Any path from s to j must leave the set T at a node i' then pass through a node j' of the frontier. Hence this competing path from s to j has length at least $d(s,i') + c_{i'j'}$. By our choice of i and j, the path from s to i in T then immediately to j has length no greater $d(s,i') + c_{i'j'}$ (figure 3.1).

c) Depth-First-Search, 2-connected components, and planarity.

Another special search algorithm is *depth-first-search.* Here we start with any node (called root) and visit every vertex in its connectivity component by always moving to a yet unvisited neighbor of the actual vertex, if exists. If not, we go back to the vertex where we came from. (For disconnected graphs, one usually continues by picking a new starting node as root for each component.) This search builds up a spanning tree (or forest for disconnected graphs) that has several interesting properties, e.g., each edge of the graph is between two vertices that are on the same branch (starting from the root). We omit the detailed applications of this method, but it can be used to find efficiently the 2-connected components of a graph or to check whether a graph can be drawn in the plane without crossing edges.

Exercise 3.1.4. *Give a fast algorithm that determines the 2-connected components of a graph.*

Maximum bipartite matching and alternating paths

Let G be a graph; we want to find a maximum size matching in G (recall that a *matching* is a subset of the edges having mutually no endpoint in common). A matching is *perfect*

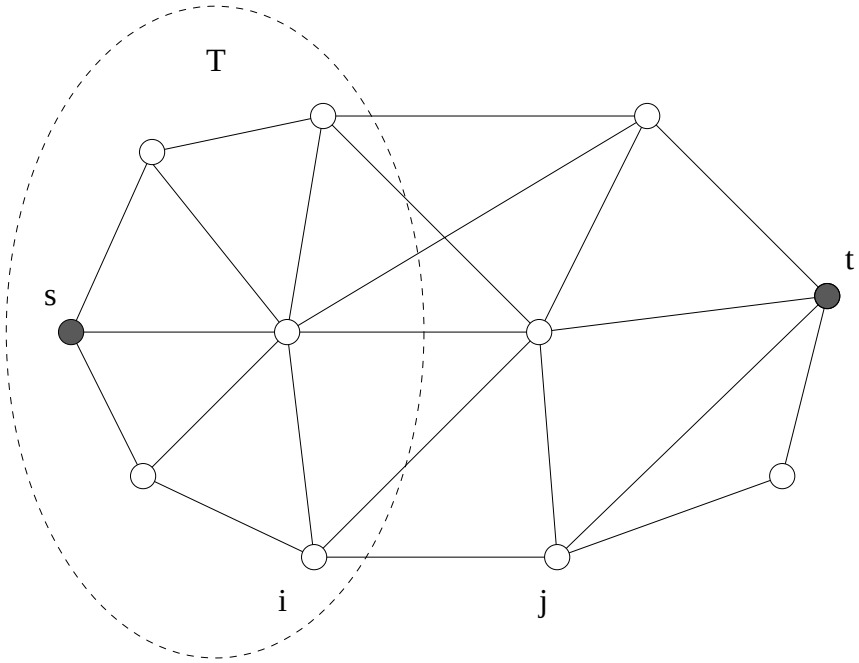

Figure 3.1: Dijkstra's shortest path algorithm.

if it meets every node. We describe a polynomial time algorithm that finds a maximum size matching in a bipartite graph.

Assume that we have a matching M; how can we see whether M is maximum? One answer to this question is provided by the following simple criterion due to Berge. An *alternating path* (with respect to M) is a path in which every other edge belongs to M (the path may start with an edge of M or with an edge not in M). An alternating path is *augmenting* if it starts and ends with an edge not in M, and moreover, its endpoints are not incident to any edge of M. (An augmenting path has necessarily odd length.)

Lemma 3.1.5. *A matching in G is maximum size if and only if there is no augmenting path with respect to it.*

Proof. It is obvious that if we have an augmenting path P with respect to M, then we can improve M; we can replace in M the edges of $P \cap M$ by the edges of $P \setminus M$, to get a larger matching.

Conversely, if M is not maximum, and there exists a larger matching M', then consider the connected components of $M \cup M'$. These are either alternating paths or alternating circuits with respect to M. At least one such component must contain more edges of M' than of M: this component is an augmenting path with respect to M. □

This fact is not very deep since it relates the existence of something (a larger matching) to the existence of something else (an augmenting path). But it does give a motivation for most of the known polynomial time matching algorithms: one can look for an augmenting path by building up a search tree.

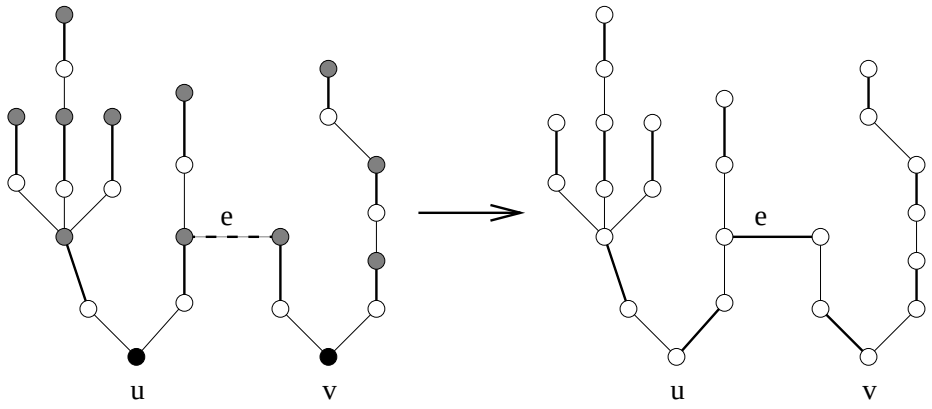

Figure 3.2: Extending a bipartite graph matching

Given a bipartite graph $G = (A, B)$, we want to find a maximal matching M. We use the notion of *alternating trees*. Given a matching M, we call a vertex of the graph *exposed* (by M) if it is not covered by M. An *alternating tree* is a subgraph of G which contains exactly one exposed node r; for which every node at odd distance from r has degree 2 in the tree; such that along any path from r every other edge belongs to the matching; and whose endpoints are all at even distance from r.

The vertices of the tree at even distance from r (including r itself) are called *outer* nodes, and the vertices at odd distance from r are called *inner* nodes.

An *alternating forest* is a forest such that every component is an alternating tree, and every exposed node of the graph is contained in the forest.

If we have a matching M and would like to extend it, we take an exposed vertex r and try to match it to a neighbour; but the neighbor may already be in the matching, so we leave this edge out; this creates a new exposed node which we try to match etc. This leads to the notion of alternating paths. Searching for alternating paths from an exposed node r leads to the construction of an alternating tree.

Now we describe the algorithm. We start with the empty matching and the alternating forest consisting of all nodes and no edges. At every stage, either we can increase the size of the matching (deleting some edges but adding more) or we can extend the forest already constructed (and keeping the matching unchanged), or we are stuck and the matching is optimal.

Suppose that we already have a candidate matching M and alternating forest F with respect to M.

Case 1. If there is an edge e connecting two outer nodes of the forest, then these nodes must belong to different trees since the graph is bipartite. Let u and v be the exposed nodes of the two trees. We have an alternating path from u to v, and switching the matching edges along the path we get a new matching M' with one more edge than M. The forest consisting of all exposed points of M' is alternating with respect to M'. (See figure 3.2).

Case 2. Otherwise, if there is an edge connecting an outer node p of the forest to a vertex u of $V(G) \setminus V(F)$, this vertex u cannot be exposed from the definition of an alternating forest. Hence it is the endpoint of an edge of the matching, say uv. Vertex v is also in $V(G) \setminus V(F)$ (otherwise u would have to be in F), and so we can extend F by adding the edges pu and uv. (See figure 3.3).

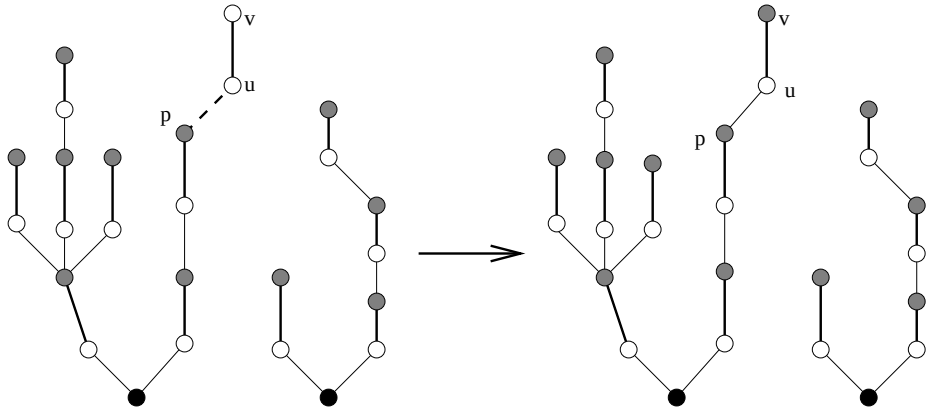

Figure 3.3: Extending an alternating forest

Case 3. If neither of the cases above applies, the outer points must be connected to inner points only. A simple but crucial counting argument will show that the matching must be optimal. (We will use this counting argument several times in this section.)

There is one exposed vertex in every component of F, and none outside F, so the total number of exposed points of the matching is equal to the number of components of F. Each outer point of the trees of F is matched to an inner point, which is its predecessor in the tree, except for the exposed vertex of each tree. Thus

$$\#\text{outer points} = \#\text{inner points} + \#\text{exposed points}.$$

Let D be the set of outer vertices and A be the set of inner vertices of F. The neighbor set of D in the graph is included in A, the number of exposed points of M is exactly $|D| - |A|$.

Let M' be any matching of the graph. M' can only match points of D to points of A, and must therefore leave at least $|D| - |A|$ points exposed. Hence M is a maximum matching.

Thus we have a valid algorithm for finding a maximum matching in a bipartite graph. At every step we must consider at most n^2 edges, and since the size of M or of F increases at each step, there are at most n^2 steps, hence a $O(n^4)$ algorithm.

Remark. Simple considerations enable to reduce the running time to $O(n^3)$. It is possible, but quite difficult, to reduce it further to $O(n^{2.5})$. It is also possible to modify the algorithm so that it works for general graphs (and not only for bipartite).

3.2 Other complexity classes

a) Exponential time. Checking "all cases" by "brute force" usually leads to exponential time algorithms, i.e., algorithms that run for somewhere between 2^{n^a} and 2^{n^b} many steps for some $a, b > 0$. For example, to decide whether a graph has a proper coloring with 3 colors, the trivial algorithm checks all possible colorings. This means checking all 3^n cases (here n denotes the number of vertices); each case takes $O(n^2)$ time. (Unfortunately although the 3^n can be reduced, for this problem we do not know

any non-exponential time algorithm, moreover, as we will see in Chapter 4, such an algorithm is not very likely to exist. We will show that graph coloring belongs to a special class of problems.)

b) Linear time. Several arithmetic operations (addition, comparison of two numbers) requires linear time.

Linear time algorithms are most important if we have to perform simple operations on very large inputs, thus most data handling algorithms have linear time. In the previous section we have seen linear time graph algorithms as well.

An algorithm is called *quasi-linear time* if it runs in $O(n(\log n)^c)$ steps where c is a constant. The most important problem that falls in this calls is sorting, for which several quasi-linear time algorithms are known. We can also find important quasi-linear time algorithms in geometric algoritms and image processing (e.g., to determine the convex hull of n planar points also requires quasi-linear time).

c) Polynomial space=PSPACE. Obviously, all polynomial time algorithms require only polynomial space, but polynomial space is significantly more general: many exponential time algorithms need only polynomial space. (It is clear that using polynomial space, we terminate in exponential time or never: if the same configuration (state of the machine, position of heads, contents of tape) is repeated, then we are in cycle).

For example, the space requirement of the trivial graph coloring algorithm discussed in a) is polynomial (even linear): if we survey all colorings in lexicographic order then it is sufficient to keep track of which coloring is currently checked and for which edges has it already been checked whether they connect points with the same color. In a similar way, we could carry out in polynomial space every brute force algorithm that tries to find in a graph a Hamiltonian cycle, the largest clique, etc.

The most typical example for a polynomial-space algorithm is finding the optimal step in a 2-person game like chess, by searching through all possible continuations.

Let us describe a general model for a "chess-like" game.

We assume that every given position of a game can be described by a word x of length n over some finite alphabet Σ (typically, telling the position of the pieces on the board, the name of the player whose turn it is and possibly some more information, e.g., in the case of chess whether the king has moved already, etc.). An initial configuration is distinguished. We assume that the two players take turns and have a way to tell, for two given positions whether it is legal to move from one into the other, by an algorithm taking polynomial time.[1] If there is no legal move in a position then it is a *terminal position* and a certain algorithm decides in polynomial time, who won. We assume that the game is always finished in n^c steps.

A position is *winning* for the player whose turn it is if there is a strategy that, starting from this position, leads to the victory of this player no matter what the other player does. A position is *losing* if no matter what move the player makes, his opponent has a winning strategy. Since the game is finite, every position is either a winning or losing position. Note that

- every legal move from a losing position leads to a winning position;

[1] It would be sufficient to assume polynomial space but it makes the game rather boring if it takes more than polynomial time to decide whether a move is legal or not.

- there is a move from any winning position that leads to a losing position.

We can visualize a game as a huge rooted tree. The root is labeled by the starting position. The children of the root are labeled by the positions that can be reached from the starting position by a legal move. Similarly, each node i of the tree is labeled by a position x_i in the game, and its children are labeled by the positions that can be reached from x_i by a legal move. The leaves of the tree are the terminal positions.

Note that the same position may occur more than once as a label, if there are different sequences of legal moves leading to that position. A node of the tree corresponds to a *subgame*, i.e., to a sequence $x_0, x_1, \ldots, x_k$ of positions such that a legal step leads from each position x_i into x_{i+1}. (We could identify nodes of the tree that correspond to the same position, and work on the resulting graph, which is not a tree any more. However, it is more transparent to have the whole tree around.)

We describe an algorithm that decides about each position in such a game whether it is a winning or losing position. The algorithm works in polynomial space but exponential time.

Roughly speaking, we search the game-tree in a depth-first-search manner, and label positions "winning" and "losing".

At a given step, the machine analyzes all possible continuations of some subgame $x_0, x_1, \ldots, x_k$. It will always be maintained that among all continuations of $x_0, x_1, \ldots, x_i$ $(0 \leq i \leq k)$ those that come before x_{i+1} (with respect to the lexicographical ordering of the words of length n) are "bad steps", i.e., they are either illegal moves or lead to a winning position (the position is winning for the other player, so to go there is a bad move!). In other words, if there are "good" moves at all, the algorithm finds the lexicographically first good move. The algorithm maintains the last legal continuation y of x_k it has studied (or that no such continuation was found yet). All the legal continuations that come before y lead to winning positions.

The algorithm looks at the words of length n following y in lexicographical order, to see whether they are legal continuations of x_k. If it finds one, this is x_{k+1} and goes on to examine the one longer partial game obtained this way. If it does not find such a move, and it did not find any legal moves from this position, then this position is terminal, and the algorithm marks it "winning" or "losing" according to the game rules. If there was at least one legal move from x_k, then we know that all legal moves from this position lead to winning positions, so this position is losing. The algorithm marks it so, and removes x_k from the partial game. It also marks x_{k-1} as winning (all you have to do to win is to move to x_k), and removes it from the partial game.

Eventually, the root gets labeled, and that tells who wins in this game.

Finally, most trivial counting algorithms require polynomial space but exponential time. e.g., to determine the number of proper 3 colorings of a graph, we can check all 3 colorings and add 1 to a counter for each coloring that is proper.

d) Linear space. This basic class is similar to polynomial space. It includes graph algorithms that can be described by operating with labels of vertices and edges, such as connectivity, hungarian method, shortest path, maximum flow. Most fixed precision numerical algorithms belong here. Nowadays in data mining due to the enormous size of the date almost only linear space algorithms are used. Breadth-first-search discussed in section 3.1 b) also belongs here.

e) Logarithmic space =LOGSPACE. This is another much investigated, important class. On one hand, it can be important when computing functions (remember that the input and the output tape do not count towards the space used!), on the other hand it is also important in case of decision problems (in which case we denote DSPACE($O(\log n)$) simply by L). In 2004 it was a groundbreaking result when Reingold has shown that the problem of deciding whether two given vertices of a graph are in the same connectivity component or not (known as USTCONN) is in L. (Interestingly about the similar problem for directed graph it is not known whether it is in L or not, thought most experts think it is not.)

Exercise 3.2.1. *Suppose that f and g are log-space computable functions. Show that their composition, $f \circ g$ is also log-space computable.*

3.3 General theorems on space and time complexity

If for a language L, there is a Turing machine deciding L for which for all large enough n the relation $\text{time}_T(n) \leq f(n)$ holds then there is also a Turing machine deciding L for which this inequality holds for all n. Indeed, for small values of n we assign the task of deciding the language to the control unit.

It can be expected that for the price of further complicating the machine, the time demands can be decreased. The next theorem shows the machine can indeed be accelerated by an arbitrary constant factor, at least if its time need is large enough (the time spent on reading the input cannot be saved).

Theorem 3.3.1 (Linear Speedup Theorem)**.** *For every Turing machine T and $c > 0$ there is a Turing machine S over the same alphabet which decides the same language and for which* $\text{time}_S(n) \leq c \cdot \text{time}_T(n) + n$.

Proof. For simplicity, let us assume that T has a single work tape (the proof would be similar for k tapes). We can assume that $c = 2/p$ where p is an integer.

Let the Turing machine S have an input-tape, $2p-1$ "starting" tapes and $2p-1$ further work tapes. Let us number these each from $1-p$ to $p-1$. Let the *index* of cell j of (start- or work) tape i be the number $j(2p-1)+i$. The start- or work cell with index t will correspond to cell t on the input resp. worktape of machine T. Let S also have an output tape.

Machine S begins its work by copying every letter of input x from its input tape to the cell with the corresponding index on its starting tapes, then moves every head back to cell 0. From then on, it ignores the "real" input tape.

Every further step of machine S will correspond to p consecutive steps of machine T. After pk steps of machine T, let the scanning head of the input tape and the work tape rest on cells t and s respectively. We will plan machine S in such a way that in this case, each cell of each start- resp. worktape of S holds the same symbol as the corresponding cell of the corresponding tape of T, and the heads rest on the starting-tape cells with indices $t-p+1, \ldots, t+p-1$ and the work-tape cells with indices $s-p+1, \ldots, s+p-1$. We assume that the control unit of machine S "knows" also

which head scans the cell corresponding to t resp. s. It knows further what is the state of the control unit of T.

Since the control unit of S sees not only what is read by T's control unit at the present moment on its input- and worktape but also the cells at a distance at most $p-1$ from these, it can compute where T's heads will step and what they will write in the next p steps. Say, after p steps, the heads of T will be in positions $t+i$ and $s+j$ (where, say, $i,j>0$). Obviously, $i,j<p$. Notice that in the meanwhile, the "work head" could change the symbols written on the work tape only in the interval $[s-p+1, s+p-1]$.

Let now the control unit of S do the following: compute and remember what will be the state of T's control unit p steps later. Remember which heads rest on the cells corresponding to the positions $(t+i)$ and $(s+j)$. Let it rewrite the symbols on the work tape according to the configuration p steps later (this is possible since there is a head on each work cell with indices in the interval $[s-p+1, s+p-1]$). Finally, move the start heads with indices in the interval $[t-p+1, t-p+i]$ and the work heads with indices in the interval $[s-p+1, s-p+j]$ one step right; in this way, the indices occupied by them will fill the interval $[t+p, t+p+i-1]$ resp. $[s+p, s+p+i-1]$ which, together with the heads that stayed in their place, gives interval $[t+i-p+1, t+i+p-1]$ resp. $[s+j-p+1, s+j+p-1]$.

If during the p steps under consideration, T writes on the output tape (either 0 or 1) and stops, then let S do this, too. Thus, we constructed a machine S that (apart from the initial copying that takes $n+n/p \le n+\mathrm{time}_T(n)/p$ steps) makes only a p-th of the number of steps of T and decides the same language. □

Exercise* **3.3.1.** *For every Turing machine T and $c>0$, one can find a Turing machine S with the same number of tapes that decides the same language and for which $\mathrm{time}_S(n) \le c\cdot \mathrm{time}_T(n)+n$ (here, we allow the extension of the alphabet).*

Exercise 3.3.2. *Formulate and prove the analogue of the above problem for storage in place of time.*

It is trivial that the storage demand of a k-tape Turing machine is at most k times its time demand (since in one step, at most k cells will be written). Therefore, if we have $\mathcal{L} \in \mathrm{DTIME}(f(n))$ for a language then there is a constant k (depending on the language) such that $\mathcal{L} \in \mathrm{DSPACE}(k\cdot f(n))$. (If extending the alphabet is allowed and $f(n)>n$ then $\mathrm{DSPACE}(k\cdot f(n)) = \mathrm{DSPACE}(f(n))$ and thus it follows that $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$.) On the other hand, the time demand is not greater than an exponential function of the space demand (since exactly the same memory configuration, taking into account also the positions of the heads and the state of the control unit, cannot occur more than once without getting into a cycle). Computing more precisely, the number of different memory configurations is at most $c\cdot n\cdot f(n)^k (2m)^{f(n)}$ where m is the size of the alphabet. (Here c is the possible states, n the position of the head on the input tape, $f(n)^k$ is the position of the other heads and $(2m)^{f(n)}$ is the description of the tape contents where the $2\cdot$ is needed to mark the end of the tapes.)

Since according to the above, the time complexity of a language does not depend on a constant factor, and in this upper bound the numbers c,k,m are constants, it follows that if $f(n) > \log n$ and $\mathcal{L} \in \mathrm{DSPACE}(f(n))$ then $\mathcal{L} \in \mathrm{DTIME}((m+1)^{f(n)})$.

So $\mathrm{DSPACE}(f(n)) \subseteq \bigcup_{c=1}^{\infty} \mathrm{DTIME}(c^{f(n)})$.

Theorem 3.3.2. *There is a function $f(n)$ such that every recursive language is an element of* $\mathrm{DTIME}(f(n))$.

Proof. There are only countably many Turing machines, so there are only countably many that halt for every input. Take an enumeration $T_1, T_2, \ldots$ of these machines. Define f as

$$f(n) := \max_{i \leq n} \mathrm{time}_{T_i}(n).$$

If $\mathcal{L}$ is recursive, there is a T_j that decides $\mathcal{L}$ in $\mathrm{time}_{T_j}(n)$ time. This is at most $f(n)$ if $n \geq j$, so (using the observation made in the first sentence of this section) there is a Turing machine T that decides $\mathcal{L}$ in $f(n)$ time for all n. □

A recursive language can have arbitrarily large time (and, due to the above inequality, also space-) complexity. More precisely:

Theorem 3.3.3. *For every recursive function $f(n)$ there is a recursive language $\mathcal{L}$ that is not an element of* $\mathrm{DTIME}(f(n))$.

Proof. The proof is similar to the proof of the fact that the halting problem is undecidable. We can assume $f(n) > n$. Let T be the 2-tape universal Turing machine constructed in Chapter 1, and let $\mathcal{L}$ consist of all words x for which it is true that having x as input on both of its tape, T halts in at most $f(|x|)^4$ steps. $\mathcal{L}$ is obviously recursive.

Let us now assume that $\mathcal{L} \in \mathrm{DTIME}(f(n))$. Then there is a Turing machine (with some $k > 0$ tapes) deciding $\mathcal{L}$ in time $f(n)$. From this we can construct a 1-tape Turing machine deciding $\mathcal{L}$ in time $cf(n)^2$ (e.g., in such a way that it stops and writes 0 or 1 as its decision on a certain cell). Since for large enough n we have $cf(n)^2 < f(n)^3$, and the words shorter than this can be recognized by the control unit directly, we can also make a 1-tape Turing machine that always stops in time $f(n)^3$. Let us modify this machine in such a way that if a word x is in $\mathcal{L}$ then it runs forever, while if $x \in \Sigma_0^* \setminus \mathcal{L}$ then it stops. This machine be S can be simulated on T by some program p in such a way that T halts with input (x, p) if and only if S halts with input x; moreover, it halts in these cases within $|p|f(|x|)^3$ steps.

There are two cases. If $p \in \mathcal{L}$ then—according to the definition of $\mathcal{L}$—starting with input p on both tapes, machine T will stop. Since the program simulates S it follows that S halts with input p. This is, however, impossible, since S does not halt at all for inputs from $\mathcal{L}$.

On the other hand, if $p \notin \mathcal{L}$ then—according to the construction of S—starting with p on its first tape, this machine halts in time $|p|f(|p|)^3 < f(|p|)^4$. Thus, T also halts in time $f(|p|)^4$. But then $p \in \mathcal{L}$ by the definition of the language $\mathcal{L}$.

This contradiction shows that $\mathcal{L} \notin \mathrm{DTIME}(f(n))$. □

There is also a different way to look at the above result. For some fixed universal two-tape Turing machine U and an arbitrary function $t(n) > 0$, the *t-bounded halting problem* asks, for n and all inputs p, x of maximum length n, whether the above machine

U halts in $t(n)$ steps. (Similar questions can be asked about storage.) This problem seems decidable in $t(n)$ steps, though this is true only with some qualification: for this, the function $t(n)$ must itself be computable in $t(n)$ steps (see the definition of "fully time-constructible" below). We can also expect a result similar to the undecidability of the halting problem, saying that the t-bounded halting problem cannot be decided in time "much less" than $t(n)$. How much less is "much less" here depends on some results on the complexity of simulation between Turing machines.

We call a function $f : \mathbf{Z}_+ \to \mathbf{Z}_+$ *fully time-constructible* if there is a multitape Turing machine that for each input of length n uses exactly $f(n)$ time steps. The meaning of this strange definition is that with fully time-constructable functions, it is easy to bound the running time of Turing machines: If there is a Turing machine making exactly $f(n)$ steps on each input of length n then we can build this into any other Turing machine as a clock: their tapes, except the input tape, are different, and the combined Turing machine carries out in each step the work of both machines.

Obviously, every fully time-constructible function is recursive. On the other hands, it is easy to see that n^2, 2^n, $n!$ and every "reasonable" function is fully time-constructible. The lemma below guarantees the existence of many completely time-constructable functions.

Let us call a function $f : \mathbf{Z}_+ \to \mathbf{Z}_+$ *well-computable* if there is a Turing machine computing $f(n)$ in time $O(f(n))$. (Here, we write n and $f(n)$ in unary notation: the number n is given by a sequence $1 \dots 1$ of length n and we want as output a sequence $1 \dots 1$ of length $f(n)$. The results would not be changed, however, if n and $f(n)$ were represented e.g., in binary notation.) Now the following lemma is easy to prove:

Lemma 3.3.4. (a) *To every well-computable function $f(n)$, there is a fully time-constructible function $g(n)$ such that $f(n) \le g(n) \le \text{constant} \cdot f(n)$.*

(b) *For every fully time-constructible function $g(n)$ there is a well-computable function $f(n)$ with $g(n) \le f(n) \le \text{constant} \cdot g(n)$.*

(c) *For every recursive function f there is a fully time-constructible function g with $f \le g$.*

This lemma allows us to use, in most cases, fully time-constructible and well-computable functions interchangeably. Following the custom, we will use the former.

Theorem 3.3.5 (Time Hierarchy Theorem)**.** *If $f(n)$ is fully time-constructible and $g(n) \log g(n) = o(f(n))$ then there is a language in* DTIME($f(n)$) *that does not belong to* DTIME($g(n)$)*.*

This says that the time complexities of recursive languages are "sufficiently dense". Analogous, but easier, results hold for storage complexities.

Exercise 3.3.3. *Prove the above theorem, and the following, closely related statement: Let $t'(n) \log t'(n) = o(t(n))$. Then the $t(n)$-bounded halting problem cannot be decided on a two-tape Turing machine in time $t'(n)$.*

Exercise 3.3.4. *Show that if $S(n)$ is any function and $S'(n) = o(S(n))$ then the $S(n)$ space-bounded halting problem cannot be solved using $S'(n)$ space.*

The full time-constructibility of the function f plays very important role in the last theorem. If we drop it then there can be an arbitrarily large "gap" below $f(n)$ which

contains the time-complexity of no language at all.

Theorem 3.3.6 (Gap Theorem)**.** *For every recursive function $\varphi(n) \geq n$ there is a recursive function $f(n)$ such that*

$$\mathrm{DTIME}(\varphi(f(n))) = \mathrm{DTIME}(f(n)).$$

Thus, there is a recursive function f with

$$\mathrm{DTIME}(f(n)^2) = \mathrm{DTIME}(f(n)),$$

moreover, there is even one with

$$\mathrm{DTIME}(2^{2^{f(n)}}) = \mathrm{DTIME}(f(n)).$$

Proof. Let us fix a 2-tape universal Turing machine. Denote $\tau(x, y)$ the time needed for T to compute from input x on the first tape and y on the second tape. (This can also be infinite.)

Claim 3.3.7. *There is a recursive function h such that for all $n > 0$ and all $x, y \in \Sigma_0^*$, if $|x|, |y| \leq n$ then either $\tau(x, y) \leq h(n)$ or $\tau(x, y) \geq (\varphi(h(n)))^3$.*

If the function

$$\psi(n) = \max\{\, \tau(x, y) : |x|, |y| \leq n,\ \tau(x, y) \text{ is finite} \,\}$$

was recursive this would satisfy the conditions trivially. This function is, however, not recursive (exercise: prove it!). We introduce therefore the following "constructive version": for a given n, let us start from the time bound $t = n + 1$. Let us arrange all pairs $(x, y) \in (\Sigma_0^*)^2$, $|x|, |y| \leq n$ in a queue. Take the first element (x, y) of the queue and run the machine with this input. If it stops within time t then throw out the pair (x, y). If it stops in s steps where $t < s \leq \varphi(t)^3$ then let $t := s$ and throw out the pair (x, y) again. (Here, we used that $\varphi(n)$ is recursive.) If the machine does not stop even after $\varphi(t)^3$ steps then stop it and place the pair (x, y) to the end of the queue. If we have passed the queue without throwing out any pair then let us stop, with $h(n) := t$. This function clearly has the property formulated in the Claim.

We will show that with the function $h(n)$ defined above,

$$\mathrm{DTIME}(h(n)) = \mathrm{DTIME}(\varphi(h(n))).$$

For this, consider an arbitrary language $\mathcal{L} \in \mathrm{DTIME}(\varphi(h(n)))$ (containment in the other direction is trivial). To this, a Turing machine can thus be given that decides $\mathcal{L}$ in time $\varphi(h(n))$. Therefore, a one-tape Turing machine can be given that decides $\mathcal{L}$ in time $\varphi(h(n))^2$. This latter Turing machine can be simulated on the given universal Turing machine T with some program p on its second tape, in time $|p| \cdot \varphi(h(n))^2$. Thus, if n is large enough then T works on all inputs (y, p) ($|y| \leq n$) for at most $\varphi(h(n))^3$ steps. But then, due to the definition of $h(n)$, it works on each such input at most $h(n)$ steps. Thus, this machine decides, with the given program (which we can also put into the control unit, if we want) the language $\mathcal{L}$ in time $h(n)$, i.e., $\mathcal{L} \in \mathrm{DTIME}(h(n))$. □

As a consequence of the theorem, we see that there is a recursive function $f(n)$ with

$$\mathrm{DTIME}((m+1)^{f(n)}) = \mathrm{DTIME}(f(n)),$$

and thus

$$\mathrm{DTIME}(f(n)) = \mathrm{DSPACE}(f(n)).$$

For a given problem, there is often no "best" algorithm: the following surprising theorem is true.

Theorem 3.3.8 (Speed-up Theorem)**.** *For every recursive function $g(n)$ there is a recursive language $\mathcal{L}$ such that for every Turing machine T deciding $\mathcal{L}$ there is a Turing machine S deciding $\mathcal{L}$ with $g(\mathrm{time}_S(n)) < \mathrm{time}_T(n)$.*

The Linear Speedup Theorem applies to every language; this theorem states only the *existence* of an arbitrarily "speedable" language. In general, for an arbitrary language, better than linear speed-up cannot be expected.

Proof. The essence of the proof is that as we allow more complicated machines we can "hard-wire" more information into the control unit. Thus, the machine needs to work only with longer inputs "on their own merit", and we want to construct the language in such a way that this should be easier and easier. It will not be enough, however, to hard-wire only the membership or non-membership of "short" words in $\mathcal{L}$, we will need more information about them.

Without loss of generality, we can assume that $g(n) > n$ and that g is a fully time-constructable function. Let us define a function h with the recursion

$$h(0) = 1,\ h(n) = (g(h(n-1)))^3.$$

It is easy to see that $h(n)$ is a monotonically increasing (in fact, very fast increasing), fully time-constructable function. Fix a universal Turing machine T_0 with, say, two tapes. Let $\tau(x, y)$ denote the time spent by T_0 working on input (x, y) (this can also be infinite). Let us call the pair (x, y) "fast" if $|y| \le |x|$ and $\tau(x, y) \le h(|x| - |y|)$.

Let $(x_1, x_2, \ldots)$ be an ordering of the words e.g., in increasing order; we will select a word y_i for certain indices i as follows. For each index $i = 1, 2, \ldots$ in turn, we check whether there is a word y not selected yet that makes (x_i, y) fast; if there are such words let y_i be a shortest one among these. Let $\mathcal{L}$ consist of all words x_i for which y_i exists and the Turing machine T_0 halts on input (x_i, y_i) with the word "0" on its first tape. (These are the words not accepted by T_0 with program y_i.)

First we convince ourselves that $\mathcal{L}$ is recursive, moreover, for all natural numbers k the question $x \in \mathcal{L}$ is decidable in $h(n-k)$ steps (where $n = |x|$) if n is large enough. We can decide the membership of x_i if we decide whether y_i exists, find y_i (if it exists), and run the Turing machine T_0 on input (x_i, y_i) for $h(|x_i| - |y_i|)$ time.

This last step itself is already too much if $|y_i| \le k$; therefore we make a list of all pairs (x_i, y_i) with $|y_i| \le k$ (this is a finite list), and put this into the control unit. This begins therefore by checking whether the given word x is in this list as the first element of a pair, and if it is, it accepts x (beyond the reading of x, this is only one step!).

Suppose that x_i is not in the list. Then y_i, if it exists, is longer than k. We can try all inputs (x, y) with $k < |y| \le |x|$ for "fastness" and this needs only $|\Sigma|^{2n+1} \cdot h(n-k-1)$

(including the computation of $h(|x| - |y|)$). The function $h(n)$ grows so fast that this is less than $h(n-k)$. Now we have y_i and also see whether T_0 accepts the pair (x_i, y_i).

Second, we show that if a program y decides the language $\mathcal{L}$ on the machine T_0 (i.e., stops for all Σ_0^* writing 1 or 0 on its first tape according to whether x is in the language $\mathcal{L}$) then y cannot be equal to any of the selected words y_i. This follows by the usual "diagonal" reasoning: if $y_i = y$ then let us see whether x_i is in the language $\mathcal{L}$. If yes then T_0 must give result "1" for the pair (x_i, y_i) (since $y = y_i$ decides $\mathcal{L}$). But then according to the definition of $\mathcal{L}$, we did not put x_i into it. Conversely, if $x_i \notin \mathcal{L}$ then it was left out since T_0 answers "1" on input (x_i, y_i); but then $x_i \in \mathcal{L}$ since the program $y = y_i$ decides $\mathcal{L}$. We get a contradiction in both cases.

Third, we convince ourselves that if program y decides $\mathcal{L}$ on the machine T_0 then (x, y) can be "fast" only for finitely many words x. Let namely (x, y) be "fast", where $x = x_i$. Since y was available at the selection of y_i (it was not selected earlier) therefore we would have had to choose some y_i for this i and the actually selected y_i could not be longer than y. Thus, if x differs from all words x_j with $|y_j| \leq |y|$ then (x, y) is not "fast".

Finally, consider an arbitrary Turing machine T deciding $\mathcal{L}$. To this, we can make a one-tape Turing machine T_1 which also decides $\mathcal{L}$ and has $\mathrm{time}_{T_1}(n) \leq (\mathrm{time}_T(n))^2$. Since the machine T_0 is universal, T_0 simulates T_1 by some program y in such a way that (let us be generous) $\tau(x, y) \leq (\mathrm{time}_T(|x|))^3$ for all sufficiently long words x. According to what was proved above, however, we have $\tau(x, y) \geq h(|x| - |y|)$ for all but finitely many x, and thus $\mathrm{time}_T(n) \geq (h(n - |y|))^{1/3}$.

Thus, for the above constructed Turing machine S deciding $\mathcal{L}$ in $h(n - |y| - 1)$ steps, we have

$$\mathrm{time}_T(n) \geq (h(n - |y|))^{1/3} \geq g(h(n - |y| - 1)) \geq g(\mathrm{time}_S(n)).$$

□

The most important conclusion to be drawn from the speed-up theorem is that even though it is convenient to talk about the computational complexity of a certain language $\mathcal{L}$, rigorous statements concerning complexity generally do not refer to a single function $t(n)$ as the complexity, but only give *upper bounds* $t'(n)$ (by constructing a Turing machine deciding the language in time $t'(n)$) or *lower bounds* $t''(n)$ (showing that no Turing machine can make the decision in time $t''(n)$ for all n).

Space versus time

Above, some general theorems were stated with respect to complexity measures. It was shown that there are languages requiring a large amount of time to decide them. Analogous theorems can be proved for the storage complexity. It is natural to ask about the relation of these two complexity measures. There are some very simple relations mentioned in the text before Theorem 3.3.2.

There is a variety of natural and interesting questions about the *trade-off* between storage and time. Let us first mention the well-know practical problem that the work of most computers can be sped up significantly by adding memory. The relation here is not really between the storage and time complexity of computations, only between slower

and faster memory. Possibly, between random-access memory versus the memory on disks, which is closer to the serial-access model of Turing machines.

There are some examples of real storage-time trade-off in practice. Suppose that during a computation, the values of a small but complex Boolean function will be used repeatedly. Then, on a random-access machine, it is worth computing these values once for all inputs and use table look-up later. Similarly, if a certain field of our records in a data base is often used for lookup then it is worth computing a table facilitating this kind of search. All these examples fall into the following category. We know some problem P and an algorithm A that solves it. Another algorithm A' is also known that solves P in less time and more storage than A. But generally, we don't have any proof that with the smaller amount of time really more storage is needed to solve P. Moreover, when a lower bound is known on the time complexity of some function, we have generally no better estimate of the storage complexity than the trivial one mentioned above (and vice versa).

Chapter 4

Non-deterministic algorithms

When an algorithm solves a problem then, implicitly, it also provides a proof that its answer is correct. This proof is, however, sometimes much simpler (shorter, easier to inspect and check) then following the whole algorithm. For example, checking whether a natural number a is a divisor of a natural number b is easier than finding a divisor of a. Here is another example. König's theorem says that in a bipartite graph, if the size of the maximum matching is k then there are k points such that every edge is incident to one of them (a minimum-size *representing set*). There are several methods for finding a maximum matching, e.g., the so-called Hungarian method, which, though polynomial, takes some time. This method also gives a representing set of the same size as the matching. The matching and the representing set together supply a proof already by themselves that the matching is maximal.

We can also reverse our point of view and can investigate the proofs without worrying about how they can be found. This point of view is profitable in several directions. First, if we know the kind of proof that the algorithm must provide this may help in constructing the algorithm. Second, if we know about a problem that even the proof of the correctness of the answer cannot be given, say, within a certain time (or storage) then we also obtained lower bound on the complexity of the algorithms solving the problem. Third (but not last), classifying the problems by the difficulty of the correctness proof of the answer, we get some very interesting and fundamental complexity classes.

These ideas, called *non-determinism* will be treated in the following sections.

4.1 Non-deterministic Turing machines

A non-deterministic Turing machine differs from a deterministic one only in that in every position, the state of the control unit and the symbols scanned by the heads permit more than one possible action. To each state $g \in \Gamma$ and symbols $h_1, \ldots, h_k$ a set of "legal actions" is given where a *legal action* is a $(2k+1)$-tuple consisting of a new state $g' \in \Gamma$, new symbols $h'_1, \ldots, h'_k$ and moves $j_1, \ldots, j_k \in \{-1, 0, 1\}$. More exactly, a non-deterministic Turing machine is an ordered 4-tuple $T = (k, \Sigma, \Gamma, \Phi)$ where $k \geq 1$ is a natural number, Σ and Γ are finite sets, $* \in \Sigma$, $\mathrm{START}, \mathrm{STOP} \in \Gamma$ (so far, everything

is as with a deterministic Turing machine) and

$$\Phi \subseteq (\Gamma \times \Sigma^k) \times (\Gamma \times \Sigma^k \times \{-1, 0, 1\}^k)$$

is an arbitrary relation. A *legal computation* of the machine is a sequence of steps where in each step (just as with the deterministic Turing machine) the control unit enters a new state, the heads write new letters on the tapes and move at most one step left or right. The steps must satisfy the following conditions: if the state of the control unit was $g \in \Gamma$ before the step and the heads read on the tapes the symbols $h_1, \ldots, h_k \in \Sigma$ then for the new state g', the newly written symbols $h'_1, \ldots, h'_k$ and the steps $\varepsilon_1, \ldots, \varepsilon_k \in \{-1, 0, 1\}$ we have

$$(g, h_1, \ldots, h_k, g', h'_1, \ldots, h'_k, \varepsilon_1, \ldots, \varepsilon_k) \in \Phi.$$

A non-deterministic Turing machine can have therefore several legal computations for the same input.

We say that the non-deterministic Turing machine T *accepts* word $x \in \Sigma_0^*$ in time t if whenever we write x on the first tape and the empty word on the other tapes, the machine has a legal computation on input x consisting of t steps, which at its halting has in position 0 of the first tape the symbol "1". (There may be other legal computations that last much longer or maybe do not even stop, or reject the word.)

We say that a non-deterministic Turing machine T *recognizes* a language $\mathcal{L}$ if $\mathcal{L}$ consists exactly of those words accepted by T (in arbitrarily long finite time). If, in addition to this, the machine accepts all words $x \in \mathcal{L}$ in time $f(|x|)$ (where $f : \mathbf{Z}_+ \to \mathbf{Z}_+$), then we say that the machine recognizes $\mathcal{L}$ in time $f(n)$ (recognizability in storage $f(n)$ is defined similarly). The class of languages recognizable by a non-deterministic Turing machine in time $f(n)$ is denoted by NTIME($f(n)$).

Unlike deterministic classes, the non-deterministic recognizability of a language $\mathcal{L}$ does not mean that the complementary language $\Sigma_0^* \setminus \mathcal{L}$ is recognizable (we will see below that each recursively enumerable but not recursive language is an example for this). Therefore, we introduce the classes co−NTIME($f(n)$): a language $\mathcal{L}$ belongs to a class co−NTIME($f(n)$) if and only if the complementary language $\Sigma_0^* \setminus \mathcal{L}$ belongs to NTIME($f(n)$).

The notion of acceptance in storage s, and the classes NSPACE($f(n)$) and co−NSPACE($f(n)$) are defined analogously.

Remarks. 1. The deterministic Turing machines can be considered, of course, as special non-deterministic Turing machines.

2. We do not wish to model any real computation device by non-deterministic Turing machines; we will see that they are used to *pose* and not to *solve* problems.

3. A non-deterministic Turing machine can make several kinds of step in a situation; we did not assume any probability distribution on these, we cannot therefore speak about the probability of some computation. If we did this then we would speak of *randomized*, or probabilistic, Turing machines, which are the subjects of Chapter 5. In contrast to non-deterministic Turing machines, randomized Turing machines model computing processes that are practically important.

4. We have mentioned that if a non-deterministic Turing machine accepts a given input in t steps, it still might have longer, even non-halting computations for the same

input. If t is a well-computable function of the input, then we can "build in a clock" to the machine which forces the computation to half after t steps. Therefore, for such functions we could modify the definition of acceptance such that all legal computations must stop after t steps and from these at least one has to accept.

Theorem 4.1.1. *Languages recognizable by non-deterministic Turing machines are exactly the recursively enumerable languages.*

Proof. Assume first that language $\mathcal{L}$ is recursively enumerable. Then, there is a Turing machine T that halts in finitely many steps on input x if and only if $x \in \mathcal{L}$. Let us modify T in such a way that when before stops it writes the symbol 1 onto field 0 of the first tape. Obviously, this modified T has a legal computation accepting x if and only if $x \in \mathcal{L}$.

Conversely, assume that $\mathcal{L}$ is recognizable by a non-deterministic Turing machine T; we show that $\mathcal{L}$ is recursively enumerable. We can assume that $\mathcal{L}$ is nonempty and let $a \in \mathcal{L}$. Let the set $\mathcal{L}^{\#}$ consist of all finite legal computations of the Turing machine T. Each element of $\mathcal{L}^{\#}$ is, in an appropriate encoding, the input and then a sequence of non-deterministic moves. Every member of this sequence describes the new state, the symbols written and which ways the heads move. The set $\mathcal{L}^{\#}$ is obviously recursive. Let S be a Turing machine that for an input y decides whether it is in $\mathcal{L}^{\#}$ and if yes then whether it describes a legal computation accepting some word x. If yes, it outputs x, if not it outputs a. The range of values of the recursive function defined by S is obviously just $\mathcal{L}$. □

4.2 Witnesses and the complexity of non-deterministic algorithms

Let us fix a finite alphabet Σ_0 and consider a language $\mathcal{L}$ over it. Let us investigate first, what it really means if $\mathcal{L}$ is recognizable within some time by a non-deterministic Turing machine. We will show that this is connected with how easy it is to "prove" for a word that it is in $\mathcal{L}$.

We say that the language $\mathcal{L}_0 \in \mathrm{DTIME}(g(n))$ is a *witness* (or *certificate*) for language $\mathcal{L}$ if we have $x \in \mathcal{L}$ if and only if there is a word $y \in \Sigma_0^*$ and $x\&y \in \mathcal{L}_0$. (Here, & is a new symbol serving the separation of the words x and y.)

Moreover, if f and g are two functions with $g(n) \geq n$, then say that the language $\mathcal{L}_0 \in \mathrm{DTIME}(g(n))$ is a *witness* of length $f(n)$ and time $g(n)$ for language $\mathcal{L}$ if it is a witness and we have $x \in \mathcal{L}$ if and only if there is a word $y \in \Sigma_0^*$ with $|y| \leq f(|x|)$ and $x\&y \in \mathcal{L}_0$. (Here, & is a new symbol serving the separation of the words x and y.)

Theorem 4.2.1. *Let $g(n) \geq n$.*

a) Every language $\mathcal{L} \in \mathrm{NTIME}(f(n))$ has a witness of length $O(f(n))$ and time $O(n)$.

b) If language $\mathcal{L}$ has a witness of length $f(n)$ and time $g(n)$ then $\mathcal{L}$ is in $\mathrm{NTIME}(g(f(n) + n + 1))$.

Proof. a) Let T be the nondeterministic Turing machine recognizing the language $\mathcal{L}$ in time $f(n)$ with, say, two tapes. Following the pattern of the proof of Theorem 4.1.1, let us assign to each word x in $\mathcal{L}$ the description of a legal computation of T accepting x in time $f(|x|)$. It is not difficult to make a Turing machine deciding about a string of length N in $O(N)$ steps whether it is the description of a legal computation and if yes then whether this computation accepts the word x. Thus, the witness is composed of the pairs $x\&y$ where y is a legal computation accepting x.

b) Let $\mathcal{L}_0$ be a witness of $\mathcal{L}$ with length $f(n)$ and time $g(n)$, and consider a deterministic Turing machine S deciding $\mathcal{L}_0$ in time $g(n)$. Let us construct a non-deterministic Turing machine T doing the following. If x is written on its first tape then it first writes the symbol & at the end of x and then writes a word y after word $x\&$ by picking each letter and the end of y non-deterministically. After this it makes a transition into state START2.

From state START2, on the first tape, the machine moves the head on the starting cell, and then proceeds to work as the Turing machine S.

This machine T has an accepting legal computation if and only if there is a word $y \in \Sigma_0^*$ for which S accepts word $x\&y$, i.e., if $x \in \mathcal{L}$ and in this case such a y exists of length at most $f(|x|)$. The running time of this computation for such a y is obviously at most $O(f(|x|)) + g(|x| + 1 + f(|x|)) = O(g(|x| + 1 + f(x)))$. □

Corollary 4.2.2. *For an arbitrary language $\mathcal{L} \subseteq \Sigma_0^*$, the following properties are equivalent:*

- *$\mathcal{L}$ is recognizable on a non-deterministic Turing machine in polynomial time.*
- *$\mathcal{L}$ has a witness of polynomial length and time.*

Remark. We mention it without proof (even without exact formulation) that these properties are also equivalent to the following: one can give a definition of $\mathcal{L}$ in the formal axiom system of set theory such that, for a word $x \in \mathcal{L}$, the statement "$x \in \mathcal{L}$" can be proved from the axioms of set theory in a number of steps polynomial in $|x|$.

We denote the class of languages having the property stated in Corollary 4.2.2 by NP. The languages $\mathcal{L}$ whose complement $\Sigma_0^* \setminus \mathcal{L}$ is in NP form the class co−NP. As we mentioned earlier, with these classes of languages, what is easy is not the solution of the recognition problem of the language, only the verification of the witnesses for the solution. We will see later that these classes are fundamental: they contain a large part of the algorithmic problems important from the point of view of practice.

Figure 4.1 illustrates the relationship between the classes P, NP and co-NP. There are, however, significant differences. First, we don't know whether or not P fills the whole intersection NP ∩ co−NP. Second, we don't know whether the difference NP \ co−NP is empty or nonempty (i.e., it could be that NP = co−NP). We will discuss this issue later.

Many important languages are given by their witnesses – more precisely, by the language $\mathcal{L}_0$ and function $f(n)$ in our definition of witnesses (we will see many examples for this later). In such cases we are asking whether a given word x is in $\mathcal{L}$ (i.e., whether there is a y with $|y| \leq f(n)$ and $x\&y \in \mathcal{L}_0$). Without danger of confusion, the word y itself will also be called the *witness word*, or simply witness, belonging to x in the

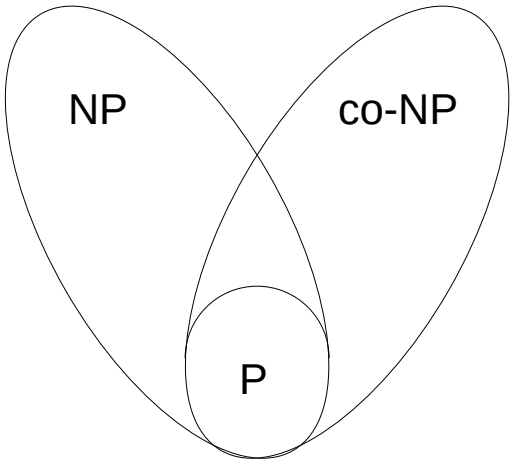

Figure 4.1: The classes of NP and co-NP

witness language $\mathcal{L}_0$. Very often, we are not only interested whether a witness word exists but would also like to produce one. This problem can be called the *search problem* belonging to the language $\mathcal{L}$. There can be, of course, several search problems belonging to a language. A search problem can make sense even if the corresponding decision problem is trivial. For example, every natural number has a prime decomposition but this is not easy to find.

Since every deterministic Turing machine can be considered non-deterministic it is obvious that

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n)).$$

In analogy with the fact that there is a recursively enumerable but not recursive language (i.e., without limits on time or storage, the non-deterministic Turing machines are "stronger"), we would expect that the above inclusion is strict. This is proved, however, only in very special cases (e.g., in case of linear functions f, by Paul, Pippenger, Trotter and Szemeredi). Later, we will treat the most important special case, the relation of the classes P and NP in detail.

The following simple relations connect the nondeterministic time- and space complexity classes:

Theorem 4.2.3. *Let f be a well-computable function. Then*

$$\mathrm{NTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathrm{DTIME}(2^{cf(n)}).$$

Proof. $\mathrm{NTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$: The essence of the construction is that all legal computations of a nondeterministic Turing machine can be tried out one after the other using only as much space as needed for one such legal computation; above this, we need some extra space to keep track of which computation we are trying out at the moment.

More exactly, this can be described as follows. Let T be a non-deterministic Turing machine recognizing language $\mathcal{L}$ in time $f(n)$. As mentioned in the first remark of this chapter, we can assume that all legal computations of T take at most $f(n)$ steps where

n is the length of the input. Let us modify the work of T in such a way that (for some input x) it will always choose first the lexicographically first action (we fix some ordering of Σ and Γ, this makes the actions lexicographically ordered). We give the new (deterministic) machine called S an extra "bookkeeping" tape on which it writes up which legal action it has chosen. If the present legal computation of T does not end with the acceptance of x then machine S must not stop but must look up, on its bookkeeping tape, the last action (say, this is the j-th one) which it can change to a lexicographically larger legal one. Let it perform a legal computation of T in such a way that up to the j-th step it performs the steps recorded on the bookkeeping tape, in the j-th step it performs the lexicographically next legal action, and after it, the lexicographically first one (and, of course, it rewrites the bookkeeping tape accordingly).

The modified, deterministic Turing machine S tries out all legal computations of the original machine T and uses only as much storage as the original machine (which is at most $f(n)$), plus the space used on the bookkeeping tape (which is again only $O(f(n))$).

$\mathrm{NSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathrm{DTIME}(2^{cf(n)})$: Let $T = \langle k, \Sigma, \Gamma, \Phi \rangle$ be a non-deterministic Turing machine recognizing $\mathcal{L}$ with storage $f(n)$. We can assume that T has only one tape. We want to try out all legal computations of T. Some care is needed since a legal computation of T can last as long as $2^{f(n)}$ steps, so there can even be $2^{2^{f(n)}}$ legal computations; we do not have time for checking this many computations.

To better organize the checking, we illustrate the situation by a graph as follows. Let us fix the length n of the input. By *configuration* of the machine, we understand a triple (g, p, h) where $g \in \Gamma$, $-f(n) \leq p \leq f(n)$ and $h \in \Sigma^{2f(n)+1}$. The state g is the state of the control unit at the given moment, the number p says where is the head and h specifies the symbols on the tape (since we are interested in computations whose storage need is at most $f(n)$ it is sufficient to consider $2f(n)+1$ cells). It can be seen that the number of configurations is at most $|\Gamma|(2f(n)+1)m^{2f(n)+1} = 2^{O(f(n))}$. Therefore, every configuration can be coded by a word of length $O(f(n))$ over Σ.

Prepare a directed graph G whose vertices are the configurations; we draw an edge from vertex u to vertex v if the machine has a legal action leading from configuration u to configuration v. Add a vertex v_0 and draw an edge to v_0 from every configuration in which the machine is in state STOP and has 1 on cell 0 of its tape. Denote u_x the starting configuration corresponding to input x. Word x is in $\mathcal{L}$ if and only if in this directed graph, a directed path leads from u_x to v_0.

On the RAM, we can construct the graph G in time $2^{O(f(n))}$ and (e.g., using breadth-first search) we can decide in time $O(|V(G)|) = 2^{O(f(n))}$ whether it contains a directed path from u_x to v_0. Since the RAM can be simulated by Turing machines in quadratic time, the time bound remains $2^{O(f(n))}$ also on the Turing machine. □

The following interesting theorem shows that the storage requirement is not essentially decreased if we allow non-deterministic Turing machines.

Theorem 4.2.4 (Savitch's Theorem)**.** *If $f(n)$ is a well-computable function and $f(n) \geq \log n$ then*

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{DSPACE}(f(n)^2).$$

Proof. Let $T = \langle 1, \Sigma, \Gamma, \Phi \rangle$ be a non-deterministic Turing machine recognizing $\mathcal{L}$ with storage $f(n)$. Let us fix the length n of inputs. Consider the graph G defined in the previous proof; we want to decide whether it contains a directed path leading from u_x to v_0. Now, of course, we do not want to construct this whole graph since it is very big. We will therefore view it as given by a certain "oracle". Here, this means that about any two vertices, we can decide in a single step whether they are connected by an edge. More exactly, this can be formulated as follows. Let us extend the definition of Turing machines. A *Turing machine with oracle for G* is a special kind of machine with three extra tapes reserved for the "oracle". The machine has a special state ORACLE. When it is in this state then in a single step, it writes onto the third oracle-tape a 1 or 0 depending on whether the words written onto the first and second oracle tapes are names of graph vertices (configurations) connected by an edge, and enters the state START. In every other state, it behaves like an ordinary Turing machine. When the machine enters the state ORACLE we say it *asks a question* from the oracle. The question is, of course, given by the pair of strings written onto the first two oracle tapes, and the answer comes on the third one.

Lemma 4.2.5. *Suppose that a directed graph G is given with an oracle on the set of words of length t. Then there is a Turing machine with an oracle for G which for given vertices u, v and natural number q decides, using $O(qt + q \log q)$ storage, whether there is a path of length at most 2^q from u to v.*

Proof. The Turing machine to be constructed will have two tapes besides the three oracle-tapes. At start, the first tape contains the pair (u, q), the second one the pair (v, q). The work of the machine proceeds in stages. At the beginning of some intermediate stage, both tapes will contain a few pairs (x, r) where x is the name of a vertex and $r \leq q$ is a natural number.

Let (x, r) and (y, s) be the last pair on the two tapes. In the present stage, the machine asks the question whether there is a path of length at most $\min\{2^r, 2^s\}$ from x to y. If $\min\{r, s\} = 0$ then the answer can be read off immediately from an oracle-tape. If $\min\{r, s\} \geq 1$ then let $m = \min\{r, s\} - 1$. We write the pair (w, m) to the end of the first tape and determine recursively whether there is a path of length at most 2^m from w to y, where w is the lexicographically first vertex initially. If there is one then we write (w, m) to the end of the second tape, erase it from the end of the first tape and determine whether there is a path of length at most 2^m from x to w. If there is one then we erase (w, m) from the end of the second tape: we know that there is a path of length at most $\min\{2^r, 2^s\}$ from x to y. If there is no path of length at most 2^m either between x and w or between w and y then we try the lexicographically next w. If we have tried all w's then we know that there is no path of length $\min\{2^r, 2^s\}$ between x and y.

It is easy to see that the second elements of the pairs are decreasing from left to right on both tapes, so at most q pairs will ever get on each tape. One pair requires $O(t + \log q)$ symbols. The storage thus used is only $O(q \log q + qt)$. This finishes the proof of the lemma. □

Returning to the proof of the theorem, note that the question whether there is an edge between two vertices of the graph G can be decided easily without the help of

additional storage; we might as well consider this decision as an oracle. The Lemma is therefore applicable with values $t, q = O(f(n))$, and we obtain that it can be decided with at most $tq + q \log q = O(f(n)^2)$ storage whether from a given vertex u_x there is a directed path into v_0, i.e., whether the word x is in $\mathcal{L}$. □

As we noted, the class PSPACE of languages decidable on a deterministic Turing machine in polynomial storage is very important. It seems natural to introduce the class NPSPACE which is the class of languages recognizable on a non-deterministic Turing machine with polynomial storage. But the following corollary of Savitch's theorem shows that this would not lead to any new notion:

Corollary 4.2.6. PSPACE = NPSPACE.

4.3 Examples of languages in NP

In this section, by a graph we mean a *simple graph*: an undirected graph without multiple edges or loops. n is always the number of nodes. Unless we say otherwise, we assume that the graph is described by its adjacency matrix, which we consider as a string in $\{0,1\}^{n^2}$. In this way, a graph property can be considered a language over $\{0,1\}$. We can thus ask whether a certain graph property is in NP. (Notice that describing a graph in one of the other usual ways, e.g., by giving a list of neighbors for each node, would not affect the membership of graph properties in NP. It is namely easy to compute these representations from each other in polynomial time.) The following graph properties are in NP.

a) Graph-connectivity. Witness: a set of $\binom{n}{2}$ paths, one for each pair of nodes.

b) Graph non-connectivity. Witness: a proper subset of the set of nodes that is not connected by any edge to the rest of the nodes.

c) Planarity. The natural witness is a concrete diagram, though some analysis is needed to see that in case such a diagram exists then one exists in which the coordinates of every node are integers whose number of bits is polynomial in n.

It is interesting to remark the fact known in graph theory that this latter diagram can be realized using single straight-line segments for the edges and thus, it is enough to specify the coordinates of the nodes. We must be careful, however, since the coordinates of the nodes used in the drawing may have too many bits, violating the requirement on the length of the witness. (It can be proved that every planar graph can be drawn in the plane in such a way that each edge is a straigh-line segment and the coordinates of every node are integers whose number of bits is polynomial in n.)

It is possible, however, to give a purely combinatorial way of drawing the graph. Let G be a graph with n nodes and m edges which we assume for simplicity to be connected. After drawing it in the plane, the edges partition the plane into domains which we call "countries" (the unbounded domain is also a country). We need the following fact, called Euler's formula:

Theorem 4.3.1. *A connected planar graph with n nodes and m edges has $n + m - 2$ countries.*

Thus to specify the drawing we give a set of $m - n + 2$ country names and for every country, we specify the sequence of edges forming its boundary (note that an edge can occur twice in a sequence). In this case, it is enough to check whether every edge appears in exactly two boundaries.

The fact that the existence of such a set of edge sequences is a necessary condition of planarity follows from Euler's formula. The sufficiency of this condition requires somewhat harder tools from topology; we will not go into these details. (Specifying a set of edge sequences as country boundaries amounts to defining a two-dimensional surface with the graph drawn onto it. A theorem of topology says that if a connected graph drawn on that surface satisfies Euler's formula then the surface is topologically equivalent (homeomorphic) to the plane.)

d) Non-planarity. Let us review the following facts.

1. Let K_5 be the complete graph on five vertices, i.e., the graph obtained by connecting five nodes in every possible way. Let $K_{3,3}$ be the complete balanced bipartite graph on six vertices, i.e., the 6-node bipartite graph containing two sets A, B of three nodes each, with every possible edge between A and B. This graph is also called "three houses, three wells" after a certain puzzle with a similar name. It is easy to see that K_5 and $K_{3,3}$ are nonplanar.

2. Given a graph G, we say that a graph G' is a *subdivision* of G if it is obtained from G by replacing each edge of G with arbitrarily long non-intersecting paths. It is easy to see that if G is nonplanar then any subdivision is nonplanar.

3. If a graph is nonplanar then, obviously, every graph containing it is also nonplanar.

The following fundamental theorem of graph theory says that the nonplanar graphs are just the ones obtained by the above operations:

Theorem 4.3.2 (Kuratowski's Theorem)**.** *A graph is nonplanar if and only if it contains a subgraph that is a subdivision of either K_5 or $K_{3,3}$.*

If the graph is nonplanar then the subgraph whose existence is stated by Kuratowski's Theorem can serve as a certificate for this.

e) Existence of perfect matching. Witness: the perfect matching itself.

f) Non-existence of a perfect matching. Witnesses for the non-existence in case of bipartite graphs are based on a fundamental theorem. Let G be a bipartite graph consisting of bipartition classes A and B. Recall the following theorem.

Theorem 4.3.3 (Frobenius's Theorem)**.** *A bipartite graph G has a perfect matching if and only if $|A| = |B|$ and for any k, any k nodes in B have at least k neighbors in A.*

Hence, if in some bipartite graph there is no perfect matching then this can be certified either by noting that A and B have different cardinality, or by a subset of A violating the conditions of the theorem.

Now let G be an arbitrary graph. If there is a perfect matching then it is easy to see that for any k, if we delete any k nodes, there remain at most k connected components of odd size. The following fundamental (and deeper) theorem says that this condition is not only necessary for the existence of a perfect matching but also sufficient.

Theorem 4.3.4 (Tutte's Theorem)**.** *A graph G has a perfect matching if and only if for any k, if we delete any k nodes, there remain at most k connected components of odd size.*

This way, if there is no perfect matching in the graph then this can be certified by a set of nodes whose deletion creates too many odd components.

A *Hamiltonian cycle* of a graph is a cycle going through each node exactly once.

g) Existence of a Hamiltonian cycle. Witness: the Hamitonian cycle itself.

A *coloring* of a graph is an assignment of some symbol called "color" to each node in such a way that neighboring nodes get different colors.

h) Colorability with three colors If a graph can be colored with three colors the coloring itself is a certificate. Of course, this is valid for any number of colors.

The properties listed above from a) to f) can be solved in polynomial time (i.e., they are in P). For the Hamiltonian cycle problem and the three-colorability problem, no polynomial algorithm is known (we return to this later).

To show that many fundamental problems in arithmetic and algebra also belong to the class NP, we recall that every natural number can be considered as a word in $\{0,1\}^*$ (representing the number in binary). We start with the problem of deciding whether a natural number is a prime.

i) Compositeness of an integer. Witness: a proper divisor.

j) Primality. It is significantly more difficult to find witnesses for primality. We use the following fundamental theorem of number theory:

Theorem 4.3.5. *An integer $n \geq 2$ is prime if and only if there is a natural number a such that $a^{n-1} \equiv 1 \pmod{n}$ but $a^m \not\equiv 1 \pmod{n}$ for any m such that $1 \leq m < n-1$.*

(This theorem says that there is a so-called "primitive root" a for n, whose powers run through all non-0 residues mod n.)

With this theorem in mind,we would like to use the number a to be the witness for the primality of n. Since, obviously, only the remainder of the number a after division by n is significant here, there will also be a witness a with $1 \leq a < n$. In this way, the restriction on the length of the witness is satisfied: a does not have more bits than n itself. Let k be the number of bits of n. As we have seen in Chapter 3, we can check the condition

$$a^{n-1} \equiv 1 \pmod{n} \tag{4.1}$$

in polynomial time. It is, however, a much harder question how to verify the further conditions:

$$a^m \not\equiv 1 \pmod{n} \qquad (1 \leq m < n-1). \tag{4.2}$$

We have seen that we can do this for each specific m, but it seems that we must do this $n-2$ times, i.e., exponentially many times in terms of k. We can use, however, the (easy) number-theoretical fact that if (4.1) holds then the smallest $m = m_0$ violating (4.2) (if there is any) is a divisor of $n-1$. It is also easy to see that then (4.2) is violated by every multiple of m_0 smaller than $n-1$. Thus, if the prime factor decomposition of

$n-1$ is $n-1 = p_1^{r_1} \cdots p_t^{r_t}$ then (4.2) is violated by some $m = (n-1)/p_i$. It is enough therefore to verify that for all i with $1 \leq i \leq t$,

$$a^{(n-1)/p_i} \not\equiv 1 \pmod{n}.$$

Now, it is obvious that $t \leq k$ and therefore we have to check (4.2) for at most k values which can be done in the way described before, in polynomial total time.

There is, however, another difficulty: how are we to compute the prime decomposition of $n-1$? This, in itself, is a harder problem than to decide whether n is a prime. We can, however, add the prime decomposition of $n-1$ to the "witness"; this consists therefore, besides the number a, of the numbers $p_1, r_1, \ldots, p_t, r_t$ (it is easy to see that this is at most $3k$ bits). The only problem that remains to be checked is whether this is indeed a prime decomposition, i.e., that $n-1 = p_1^{r_1} \cdots p_t^{r_t}$ (this is easy) and that $p_1, \ldots, p_t$ are indeed primes. We can do this recursively.

We still have to check that this recursive method gives witnesses of polynomial length and it can be decided in polynomial time that these are witnesses. Let $L(k)$ denote the maximum length of the witnesses in case of numbers n of k bits. Then, according to the above recursion,

$$L(k) \leq 3k + \sum_{i=1}^{t} L(k_i)$$

where k_i is the number of bits of the prime p_i. Since $p_1 \cdots p_t \leq n-1 < n$ it follows easily that

$$k_1 + \cdots + k_t \leq k.$$

Also obviously $k_i \leq k-1$. Using this, it follows from the above recursion that $L(k) \leq 3k^2$. This is obvious for $k = 1$ and if we know that it holds for all numbers less than k then

$$L(k) \leq 3k + \sum_{i=1}^{t} L(k_i) \leq 3k + \sum_{i=1}^{t} 3k_i^2$$

$$\leq 3k + 3(k-1) \sum_{i=1}^{t} k_i \leq 3k + 3(k-1) \cdot k \leq 3k^2.$$

We can prove similarly that it is decidable about a string in polynomial time whether it is a certificate defined in the above way.

Usually we are not satisfied with knowing whether a given number n is a prime or not, but if it is not a prime then we might also want to find one of its proper divisors. (If we can solve this problem then repeating it, we can find the complete prime decomposition.) This is not a decision problem, but it is not difficult to reformulate it as a decision problem:

k) Existence of a bounded divisor. Given two natural numbers n and k; does n have a proper divisor not greater than k?

It is clear that this problem is in NP: the certificate is the divisor.

The complementary language is also in NP: This is the set of all pairs (n, k) such that every proper divisor of n is greater than k. A certificate for this is the prime decomposition of n, together with a certificate of the primality of every prime factor.

It is not known whether the existence of a bounded divisor is in P, moreover not even a randomized polynomial algorithm was yet found. However, in 2002 primality was proved to be in P by Agrawal, Kayal and Saxena.

Now we turn to some basic questions in algebra. A notion analogous for primality of a positive integer is irreducibility of a polynomial (for simplicity, with a single variable, and with rational coefficients). A polynomial is *reducible* if it can be written as the product of two non-constant polynomials with rational coefficients.

l) Reducibility of a polynomial over the rational field. Witness: a proper divisor; but some remarks are in order.

Let f be the polynomial. To prove that this problem is in NP we must convince ourselves that the number of bits necessary for writing down a proper divisor can be bounded by a polynomial of the number of bits in the representation of f. (We omit the proof of this here.)

It can also be shown that this language is in P.

A system $Ax \leq b$ of linear inequalities (where A is an integer matrix with m rows and n columns and b is a column vector of m elements) can be considered a word over the alphabet consisting of the symbols "0", "1", "," and ";" when e.g., we represent its elements in the binary number system, write the matrix row after row, placing a comma after each number and a semicolon after each row. The following properties of systems of linear inequalities are in NP:

m) Existence of solution. The solution offers itself as an obvious witness of solvability but we must be careful: we need that if a system of linear equations with integer coefficients has a solution then it has a solution among rational numbers, moreover, even a solution in which the numerators and denominators have only a polynomial number of bits. These facts follow from the elements of the theory of linear programming.

n) Nonexistence of solution. Witnesses for the non-existence of solution can be found using the following fundamental theorem known from linear programming:

Theorem 4.3.6 (Farkas's Lemma)**.** *The system $Ax \leq b$ of inequalities is unsolvable if and only if the following system of inequalities is solvable: $y^T A = 0$, $y^T b = -1$, $y \geq 0$.*

In words, this lemma says that a system of linear inequalities is unsolvable if and only if a contradiction can be obtained by a linear combination of the inequalities with nonnegative coefficients.

Using this, a solution of the system of inequalities given in the lemma (the nonnegative coefficients) is a witness of the nonexistence of a solution for the original system.

o) Existence of an integer solution. The solution itself is a witness but we need some reasoning again to limit the size of witnesses, which is difficult here. The arguments needed are similar to the ones seen at the discussion of Gaussian elimination in Section 3.1 and are based on Cramer's rule.

It is not known whether the non-existence of an integer solution is in NP, i.e., if this fact can be certified by a polynomial length and polynomial time verifiable certificate.

It is important to note that the fundamental problem of linear programming, i.e., looking for the optimum of a linear object function under linear conditions, can be easily reduced to the problem of solvability of systems of linear inequalities. Similarly, the search for optimal integer solutions can be reduced to the decision of the existence of integer solutions.

For a long time, it was unknown whether the problem of solvability of systems of linear inequalities is in P (the well-known simplex method is not polynomial). The first polynomial algorithm for this problem was the ellipsoid method of L. G. Khachian (relying on work by Yudin and Nemirovskii).

The running time of this method led, however, to a very high-degree polynomial; it could not therefore compete in practice with the simplex method which, though is exponential in the worst case, is on average (in practice) much faster than the ellipsoid method.

Several polynomial time linear programming algorithms have been found since; among these, Karmarkar's method can compete with the simplex method even in practice.

No polynomial algorithm is known for solving systems of linear inequalities in integers; one cannot even hope to find such an algorithm (see the notion of NP-completeness below).

Reviewing the above list of examples, the following observations can be made.

- For many properties that are in NP, their negation (i.e., the complement of the corresponding language) is also in NP. This fact is, however, generally non-trivial; in various branches of mathematics, often the most fundamental theorems assert this for certain languages.

- It is often the case that if some property (language) turns out to be in NP$\cap$co$-$NP then sooner or later it also turns out to be in P. This happened, for example, with the existence of perfect matchings, planarity, the solution of systems of linear inequalities. Research is very intensive on prime testing. If NP is considered an analog of "recursively enumerable" and P an analog of "recursive" then we can expect that this is always the case. However, there is no proof for this; moreover, this cannot really be expected to be true in full generality.

- In case of other NP problems, their solution in polynomial time seems hopeless, they are very hard to handle (Hamiltonian cycle, graph coloring, and integer solution of a system of linear inequalities). We cannot prove that these are not in P (we don't know whether P = NP holds); but still, one can prove a certain property of these problems that shows that they are hard. We will return to this in a later section of this chapter.

- There are many problems in NP with a naturally corresponding search problem and with the property that if we can solve the decision problem then we can also solve (in a natural manner) the search problem. E.g., if we can decide in polynomial time whether there is a perfect matching in a certain graph then we can search for a perfect matching in polynomial time in the following way. We delete edges from the graph as long as a perfect matching still remains in it. When

we get stuck, the remaining graph must be a perfect matching. Using similar simple tricks, the search problem corresponding to the existence of Hamiltonian cycles, colorability with 3 colors, etc. can be reduced to the decision problem. This is, however, not always so. E.g., our ability to decide in polynomial time whether a number is a prime is not applicable to the problem of finding a proper divisor.

- A number of NP-problems have a related *optimization problem* which is more natural to state, even if it is not an NP-problem by its form. For example, instead of the general matching problem, it is more natural to determine the maximum size of a matching in the graph. In case of the coloring problem, we may want to look for the chromatic number, the smallest number of colors with which the graph is colorable. The solvability of a set of linear inequalities is intimately connected with the problem of finding a solution that maximizes a certain linear form: this is the problem of linear programming. Several other examples come later. If there is a polynomial algorithm solving the optimization problem then it automatically solves the associated NP problem. If there is a polynomial algorithm solving the NP-problem then, using binary search, it provides a polynomial algorithm to solve the associated optimization problem.

There are, of course, interesting problems (languages) also in other non-deterministic complexity classes. The *non-deterministic exponential time* (NEXPTIME) class can be defined as the union of the classes NTIME(2^{n^c}) for all $c > 0$. We can formulate an example in connection with Ramsey's Theorem. Let G be a graph; the *Ramsey number* $R(G)$ corresponding to G is the smallest $N > 0$ such that no matter how we color the edges of the N-node complete graph with two colors, some color contains a copy of G. (Ramsey's Theorem is the non-trivial fact that such a finite number exists.) Let $\mathcal{L}$ consist of the pairs (G, N) for which $R(G) > N$. The size of the input (G, N) (if G is described, say, by its adjacency matrix) is $O(|V(G)|^2 + \log N)$.

Now, $\mathcal{L}$ is in NEXPTIME since the fact $(G, N) \in \mathcal{L}$ is witnessed by a coloring of the complete graph on N nodes in which no homogenously colored copy of G; this property can be checked in time $O(N^{|V(G)|})$ which is exponential in the size of the input (but not worse). On the other hand, deterministically, we know no better algorithm to decide $(G, N) \in \mathcal{L}$ than a double exponential one. The trivial algorithm, which is, unfortunately, the best known, goes through all $2^{N(N-1)/2}$ colorings of the edges of the N-node complete graph.

4.4 NP-completeness

We say that a language $\mathcal{L}_1 \subseteq \Sigma_1^*$ is *polynomially reducible* to a language $\mathcal{L}_2 \subseteq \Sigma_2^*$ if there is a function $f : \Sigma_1^* \to \Sigma_2^*$ computable in polynomial time such that for all words $x \in \Sigma_1^*$ we have

$$x \in \mathcal{L}_1 \Leftrightarrow f(x) \in \mathcal{L}_2.$$

It is easy to verify from the definition that this relation is transitive:

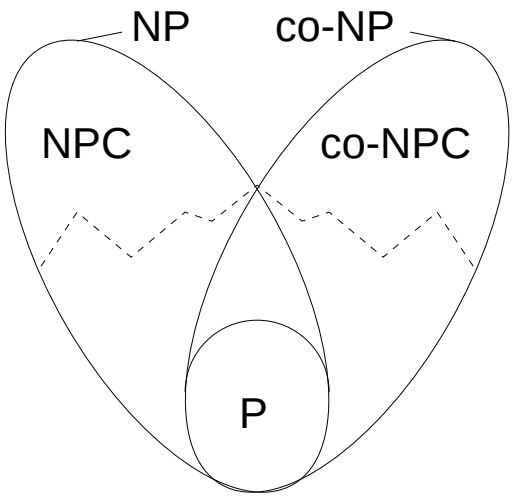

Figure 4.2: The classes of NP-complete (NPC) and co-NP-complete languages

Proposition 4.4.1. *If $\mathcal{L}_1$ is polynomially reducible to $\mathcal{L}_2$ and $\mathcal{L}_2$ is polynomially reducible to $\mathcal{L}_3$ then $\mathcal{L}_1$ is polynomially reducible to $\mathcal{L}_3$.*

The membership of a language in P can also be expressed by saying that it is polynomially reducible to the language $\{1\}$.

Proposition 4.4.2. *If a language is in* P *then every language is in* P *that is polynomially reducible to it. If a language is in* NP *then every language is in* NP *that it polynomially reducible to it.*

We call a language NP-*complete* if it belongs to NP and every language in NP is polynomially reducible to it. These are thus the "hardest" languages in NP. The class of NP-complete languages is denoted by NPC. Figure 4.2 adds the class of NP-complete languages to figure 4.1. We'll see that the position of the dotted line is not a proved fact: for example, if P = NP, then also NPC = P.

The word "completeness" suggests that such a problem contains all the complexity of the whole class: the solution of the decision problem of a complete language contains, in some sense, the solution to the decision problems of all other NP languages. If we could show about even a single NP-complete language that it is in P then P = NP would follow. The following observation is also obvious.

Proposition 4.4.3. *If an* NP-*complete language $\mathcal{L}_1$ is polynomially reducible to a language $\mathcal{L}_2$ in* NP *then $\mathcal{L}_2$ is also* NP-*complete.*

It is not obvious at all that NP-complete languages exist. Our first goal is to give an NP-complete language; later (by polynomial reduction, using Proposition 4.4.3) we will prove the NP-completeness of many other problems.

A Boolean polynomial is called *satisfiable* if the Boolean function defined by it is not identically 0. The *Satisfiability Problem* is to decide for a given Boolean polynomial f whether it is satisfiable. We consider the problem when the Boolean polynomial is a conjunctive normal form.

Exercise 4.4.1. *When is a disjunctive normal form satisfiable?*

Exercise 4.4.2. *Given a graph G and three colors, 1,2 and 3. Let us introduce, to each vertex v and color i a logical value $x[v,i]$. Give a conjunctive normal form B for the variables $x[v,i]$ which is satisfiable if and only if*

a) G can be colored with 3 colors such that $x[v,i]$ is true if and only if the color of v is i;

b) the coloring is also required to be proper.

Give a conjunctive normal form which is satisfiable if and only if G is 3-colorable.

We can consider each conjunctive normal form as a word over the alphabet consisting of the symbols "x", "0", "1", "$+$", "$\neg$" (or "$\overline{x}$"), "$\wedge$" and "$\vee$" (we write the indices of the variables in binary number system, e.g., $x_6 = x110$). Let SAT denote the language formed from the satisfiable conjunctive normal forms.

The following theorem is one of the central results in complexity theory.

Theorem 4.4.4 (Cook–Levin Theorem)**.** *The language* SAT *is* NP*-complete.*

Proof. Let $\mathcal{L}$ be an arbitrary language in NP. Then there is a non-deterministic Turing machine $T = \langle k, \Sigma, \Gamma, \Phi \rangle$ and there are integers $c, c_1 > 0$ such that T recognizes $\mathcal{L}$ in time $c_1 \cdot n^c$. We can assume $k = 1$. Let us consider an arbitrary word $h_1 \cdots h_n \in \Sigma^*$. Let $N = \lceil c_1 \cdot n^c \rceil$. Let us introduce the following variables:

$$\begin{aligned} x[t,g]&\ (0 \le t \le N,\ g \in \Gamma),\\ y[t,p]&\ (0 \le t \le N,\ -N \le p \le N),\\ z[t,p,h]&\ (0 \le t \le N,\ -N \le p \le N,\ h \in \Sigma). \end{aligned}$$

If a legal computation of the machine T is given then let us assign to these variables the following values: $x[t,g]$ is true if after the t-th step, the control unit is in state g; $y[t,p]$ is true if after the t-th step, the head is on the p-th tape cell; $z[t,p,h]$ is true if after the t-th step, the p-th tape cell contains symbol h. The variables x, y, z obviously determine the computation of the Turing machine.

However, not every possible system of values assigned to the variables will correspond to a computation of the Turing machine. One can easily write up logical relations among the variables that, when taken together, express the fact that this is a legal computation accepting $h_1 \cdots h_n$. We must require that the control unit be in some state in each step:

$$\bigvee_{g \in \Gamma} x[t,g] \qquad (0 \le t \le N);$$

and it should not be in two states:

$$\overline{x[t,g]} \vee \overline{x[t,g']} \qquad (g, g' \in \Gamma,\ 0 \le t \le N).$$

We can require, similarly, that the head should be only in one position in each step and there should be one and only one symbol in each tape cell. We write that initially

the machine is in state START and at the end of the computation, in state STOP, and the head starts from cell 0:

$$x[0, \mathrm{START}] = 1, \qquad x[N, \mathrm{STOP}] = 1, \qquad y[0, 0] = 1;$$

and, similarly, that the tape contains initially the input $h_1, \ldots, h_n$ and finally the symbol 1 on cell 0:

$$z[0, i-1, h_i] = 1 \ (1 \leq i \leq n)$$
$$z[0, i-1, *] = 1 \ (i < 0 \text{ or } i > n)$$
$$z[N, 0, 1] = 1.$$

We must further express the computation rules of the machine, i.e., that for all $g, g' \in \Gamma$, $h, h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$, $-N \leq p \leq N$ and $0 \leq t < N$ if $(g, h, g', h', \varepsilon) \notin \Phi$, then we have

$$\overline{x[t, g]} \vee \overline{y[t, p]} \vee \overline{z[t, p, h]} \vee \overline{x[t+1, g']} \vee \overline{y[t+1, p+\varepsilon]} \vee \overline{z[t+1, p, h']},$$

and to make sure that the head does not jump far, also

$$\overline{y[t, p]} \vee y[t+1, p-1] \vee y[t+1, p] \vee y[t+1, p+1].$$

We also need that where there is no head the tape content does not change:

$$y[t, p] \vee z[t, p, h] \vee \overline{z[t+1, p, h]}.$$

Joining all these relations by the sign "$\wedge$" we get a conjunctive normal form that is satisfiable if and only if the Turing machine T has a computation of at most N steps accepting $h_1, \ldots, h_n$. It easy to verify that for given $h_1, \ldots, h_n$, the described construction of a formula can be carried out in polynomial time. □

It will be useful to prove the NP-completeness of some special cases of the satisfiability problem. A conjunctive normal form is called a *k-form* if in each of its components, at most k literals occur. Let k-SAT denote the language made up by the satisfiable k-forms. Let further SAT-k denote the language consisting of those satisfiable conjunctive normal forms in which each variable occurs in at most k elementary disjunctions.

Theorem 4.4.5. *The language* 3-SAT *is* NP*-complete.*

Proof. We reduce SAT to $3-$SAT by introducing new variables. Given a B conjunctive normal form, take one of its disjunctions. This can be expressed as $E = z_1 \vee \ldots \vee z_k$ where each z_i is a literal. Let us introduce k new variables, $y_1^E, \ldots y_k^E$ and take the

$$y_1^E \vee \overline{z_1}, \Rightarrow \overline{y}_1^E \vee z_1$$

and

$$y_i^E \vee \overline{z}_i, \Rightarrow y_i^E \vee \overline{y}_{i-1}^E, \Rightarrow \overline{y}_i^E \vee y_{i-1}^E \vee z_i \Rightarrow (i = 2, \ldots k)$$

disjunctions (these express that $y_1^E = z_1$ and $y_i^E = y_{i-1}^E \vee z_i$, i.e., $y_i^E = z_1 \vee \ldots \vee z_i$). These and the single component disjunctions y_k^E for each E connected with $\wedge$ symbols we get a 3-form that is satisfiable if and only if B is satisfiable. □

It is natural to wonder at this point why have we considered only the 3-satisfiability problem. The problems 4-SAT, 5-SAT, etc. are harder than 3-SAT therefore these are, of course, also NP-complete. The theorem below shows, on the other hand, that the problem 2-SAT is already not NP-complete (at least if P $\neq$ NP). This illustrates the fact that often a little modification of the conditions of a problem leads from a polynomially solvable problem to an NP-complete one.

Theorem 4.4.6. *The language 2-*SAT *is in* P.

Proof. Let B be a 2-form on the variables $x_1, \ldots, x_n$. Let us use the convention that the variables x_i are also written as x_i^1 and the negated variables $\overline{x}_i$ are also written as new symbols x_i^0. Let us construct a directed graph G on the set $V(G) = \{x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n\}$ in the following way: we connect node x_i^ε to node x_j^δ if $x_i^{1-\varepsilon} \vee x_j^\delta$ is an elementary disjunction in B. (This disjunction is equivalent to $x_i^\varepsilon \Rightarrow x_j^\delta$.) Let us notice that then in this graph, there is also an edge from $x_j^{1-\delta}$ to $x_i^{1-\varepsilon}$. In this directed graph, let us consider the *strongly connected components*; these are the classes of nodes obtained when we group two nodes in one class whenever there is a directed path between them in both directions.

Lemma 4.4.7. *The formula B is satisfiable if and only if none of the strongly connected components of G contains both a variable and its negation.*

The theorem follows from this lemma since it is easy to find in polynomial time the strongly connected components of a directed graph. □

Proof of Lemma 4.4.7 Let us note first that if an assignment of values satisfies formula B and x_i^ε is "true" in this assignment then every x_j^δ is "true" to which an edge leads from x_i^ε: otherwise, the elementary disjunction $x_i^{1-\varepsilon} \vee x_j^\delta$ would not be satisfied. It follows from this that the nodes of a strongly connected component are either all "true" or none of them. But then, a variable and its negation cannot simultaneously be present in a component.

Conversely, let us assume that no strongly connected component contains both a variable and its negation. Consider a variable x_i. According to the condition, there cannot be directed paths in both directions between x_i^0 and x_i^1. Let us assume there is no such directed path in either direction. Let us then draw a new edge from x_i^1 to x_i^0. This will not violate our assumption that no connected component contains both a node and its negation. If namely such a connected components should arise then it would contain the new edge, but then both x_i^1 and x_i^0 would belong to this component and therefore there would be a path from x_i^0 to x_i^1. But then this path would also be in the original graph, which is impossible.

Repeating this procedure, we can draw in new edges (moreover, always from a variable to its negation) in such a way that in the obtained graph, between each variable and its negation, there will be a directed path in exactly one direction. Let now be $x_i = 1$ if a directed path leads from x_i^0 to x_i^1 and 0 if not. We claim that this assignment satisfies all disjunctions. Let us namely consider an elementary disjunction, say, $x_i \vee x_j$. If both of its members were false then—according to the definition—there were a directed path from x_i^1 to x_i^0 and from x_j^1 to x_j^0. Further, according to the definition of

the graph, there is an edge from x_i^0 to x_j^1 and from x_j^0 to x_i^1. But then, x_i^0 and x_i^1 are in a strongly connected component, which is a contradiction. □

Theorem 4.4.8. *The language* SAT-3 *is* NP*-complete.*

Proof. Let B be a Boolean formula of the variables $x_1, \ldots, x_n$. For each variable x_j, replace the i-th occurrence of x_j in B, with new variable y_j^i: let the new formula be B'. For each j, assuming there are m occurrences of x_j in B, form the conjunction

$$C_j = (y_j^1 \Rightarrow y_j^2) \wedge (y_j^2 \Rightarrow y_j^3) \wedge \cdots \wedge (y_j^m \Rightarrow y_j^1).$$

(Of course, $y_j^1 \Rightarrow y_j^2$ is equivalent to $\overline{y}_j^1 \vee y_j^2$, so the above can be rewritten into a conjunctive normal form.) The formula $B' \wedge C_1 \wedge \cdots \wedge C_n$ contains at most 3 occurrences of each variable, is a conjunctive normal form if B is, and is satisfiable obviously if and only if B is. □

Exercise 4.4.3. *Define the language 3-*SAT*-3 and show that it is* NP*-complete.*

Exercise 4.4.4. *Define the language* SAT*-2 and show that it is in* P*.*

4.5 Further NP-complete problems

One might think that NP-complete problems are of logical character. In what follows, we will show the NP-completeness of a number of important "everyday" combinatorial, algebraic, etc. problems. When we show that a problem is NP-complete, then it follows that it is not in P unless P = NP. Therefore, we can consider the NP-completeness of a language as a proof of its undecidability in polynomial time.

Let us formulate a fundamental combinatorial problem:

Problem 4.5.1 (Blocking Set Problem)**.** *Given a system $\{A_1, \ldots, A_m\}$ of finite sets and a natural number k, is there a set with at most k elements intersecting every A_i?*

We have met a special case of this problem, the Blocking Set Problem for the edges of a bipartite graph in Section 3.1. This special case was polynomial time solvable. In contrast to this, we prove:

Theorem 4.5.1. *The Blocking Set Problem is* NP*-complete.*

Proof. We reduce 3-SAT to this problem. For a given conjunctive 3-normal form B we construct a system of sets as follows: let the underlying set be the set $\{x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n\}$ of the variable symbols occurring in B and their negations. For each clause of B, let us take the set of literals occurring in it; let us further take the sets $\{x_i, \overline{x}_i\}$. The elements of this set system can be blocked with at most n nodes if and only if the normal form is satisfiable. □

The Blocking Set Problem remains NP-complete even if we impose various restrictions on the set system. It can be seen from the above construction that the Blocking Set Problem is NP-complete even for a system of sets with at most three elements. (We will see a little later that this holds even if the system contains only two-element

sets, i.e., the edges of a graph.) If we reduce the language SAT first to the language SAT-3 according to Theorem 4.4.8 and apply to this the above construction then we obtain a set system for which each element of the underlying set is in at most 4 sets.

With a little care, we can show that the Blocking Set Problem remains NP-complete even for set-systems in which each element is contained in at most 3 sets. Indeed, it is easy to reduce the Satisfiability Problem to the case when the input is a conjunctive normal form in which every variable occurs at least once negated and at least once unnegated; then the construction above gives such a set-system.

We cannot go further than this: if each element is in at most 2 sets then the Blocking Set Problem is solvable in polynomial time. In fact, it is easy to reduce this special case of the blocking set problem to the matching problem.

It is easy to see that the following problem is equivalent to the Blocking Set Problem (only the roles of "elements" and "subsets" must be interchanged):

Problem 4.5.2 (Covering problem)**.** *Given a system $\{A_1, \dots, A_m\}$ of subsets of a finite set S and a natural number k. Can k sets be selected in such a way that their union is the whole set S?*

According to the discussion above, this problem is NP-complete even when each of the given subsets has at most 3 elements but it is in P if the size of the subsets is at most 2.

For set systems, the following pair of problems is also important.

Problem 4.5.3 (k-partition problem)**.** *Given a system $\{A_1, \dots, A_m\}$ of subsets of a finite set V and a natural number k. Can a subsystem of k sets $\{A_{i_1}, \dots, A_{i_k}\}$ be selected that gives a* partition *of the underlying set (i.e., consists of disjoint sets whose union is the whole set V)?*

Problem 4.5.4 (Partition problem)**.** *Given a system $\{A_1, \dots, A_m\}$ of subsets of a finite set S. Can a subsystem (of any size) be selected that gives a partition of the underlying set?*

If all the A_i's are of the same size, then of course the number of sets in a partition is uniquely determined, and so the two problems are equivalent.

Theorem 4.5.2. *The k-partition problem and the partition problem are* NP*-complete.*

Proof. We reduce the Covering Problem with sets having at most 3 elements to the k-partition problem. Thus we are given a system of sets with at most 3 elements each and a natural number k. We want to decide whether k of these given sets can be selected in such a way that their union is the whole S. Let us expand the system by adding all subsets of the given sets (it is here that we exploit the fact that the given sets are bounded: from this, the number of sets grows at most $2^3 = 8$-fold). Obviously, if k sets of the original system cover S then k appropriate sets of the expanded system provide a partition of S, and vice versa. In this way, we have found that the k-partition problem is NP-complete.

Second, we reduce the k-partition problem to the partition problem. Let U be a k-element set disjoint from S. Let our new underlying set be $S \cup U$, and let our new set system contain all the sets of form $A_i \cup \{u\}$ where $u \in U$. Obviously, if from this new set system, some sets can be selected that form a partition of the underlying set

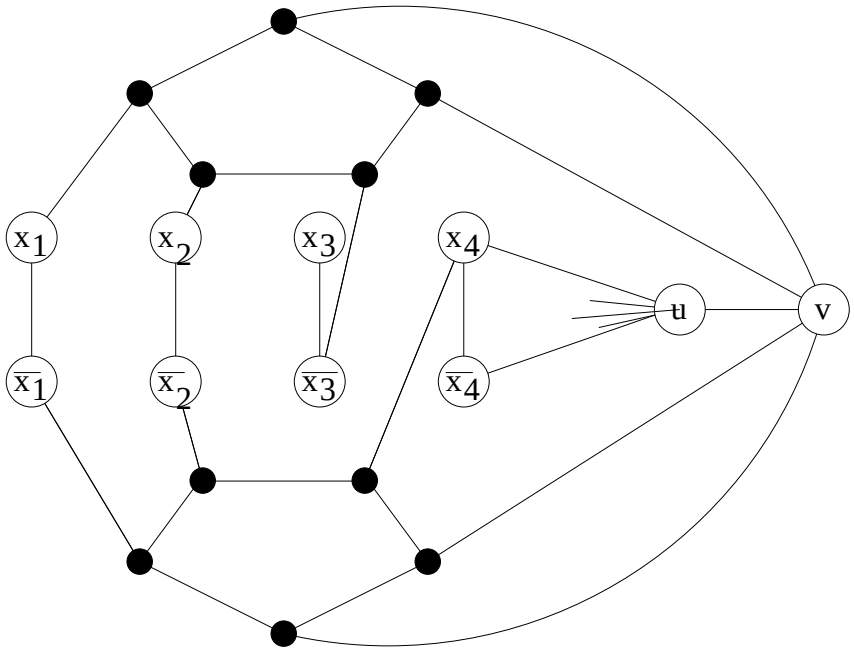

Figure 4.3: The graph whose 3-coloring is equivalent to satisfying the expression $(\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$

then the number of these is k and the parts falling in S give a partition of S into k sets. Conversely, every partition of S into k sets A_i provides a partition of the set $S \cup U$ into sets from the new set system. Thus, the partition problem is NP-complete. □

If the given sets have two elements then the Partition problem is just the perfect matching problem and can therefore be solved in polynomial time. On the other hand, the Partition problem for sets with at most 3 elements is NP-complete.

Next we treat a fundamental graph-theoretic problem, the coloring problem. We have seen that the problem of coloring with two colors is solvable in polynomial time. On the other hand:

Theorem 4.5.3. *The problem whether a graph can be colored with three colors is an* NP*-complete problem.*

Proof. Let a 3-form B be given; we construct a graph G for it that is colorable with three colors if and only if B is satisfiable.

For the nodes of the graph G, we first take the literals, and we connect each variable with its negation. We take two more nodes, u and v, and connect them with each other, further we connect u with all unnegated and negated variables. Finally, we take a pentagon for each elementary disjunction $z_{i_1} \vee z_{i_2} \vee z_{i_3}$; we connect two neighboring vertices of the pentagon with v, and its three other vertices with z_{i_1}, z_{i_2} and z_{i_3}. We claim that the graph G thus constructed is colorable with three colors if and only if B is satisfiable (Figure 4.3).

The following observation, which can be very easily verified, plays a key role in the proof: *if for some clause $z_{i_1} \vee z_{i_2} \vee z_{i_3}$, the nodes z_{i_1}, z_{i_2}, z_{i_3} and v are colored with three colors then this coloring can be extended to the pentagon as a legal coloring if and*

only if the colors of z_{i_1}, z_{i_2}, z_{i_3} and v are not identical.

Let us first assume that B is satisfiable, and let us consider the corresponding value assignment. Color red those (negated or unnegated) variables that are "true", and blue the others. Color u yellow and v blue. Since every elementary disjunction must contain a red node, this coloring can be legally extended to the nodes of the pentagons.

Conversely, let us assume that the graph G is colorable with three colors and let us consider a "legal" coloring with red, yellow and blue. We can assume that the node v is blue and the node u is yellow. Then the nodes corresponding to the variables can only be blue and red, and between each variable and its negation, one is red and the other one is blue. Then the fact that the pentagons are also colored implies that each elementary disjunction contains a red node. But this also means that taking the red nodes as "true", we get a value assignment satisfying B. □

It follows easily from the previous theorem that for every number $k \geq 3$ the k-colorability of graphs is NP-complete.

The following is another very basic graph theory problem. A set S of nodes of a graph is *independent*, if no edge connects any two of them.

Problem 4.5.5 (Independent node set problem)**.** *Given a graph G and a natural number k, is there an independent set of nodes of size k in G?*

Theorem 4.5.4. *The Independent node set problem is* NP*-complete.*

Proof. We reduce to this problem the problem of coloring with 3 colors. Let G be an arbitrary graph with n nodes and let us construct the graph H as follows: Take three disjoint copies G_1, G_2, G_3 of G and connect the corresponding nodes of the three copies. Let H be the graph obtained, this has thus $3n$ nodes.

We claim that there are n independent nodes in H if and only if G is colorable with three colors. Indeed, if G is colorable with three colors, say, with red, blue and yellow, then the nodes in G_1 corresponding to the red nodes, the nodes in G_2 corresponding to the blue nodes and the nodes in G_3 corresponding to the yellow nodes are independent even if taken together in H, and their number is n. The converse can be proved similarly. □

In the set system constructed in the proof of Theorem 4.5.1 there were sets of at most three elements, for the reason that we reduced the 3-SAT problem to the Blocking Set Problem. Since the 2-SAT problem is in P, we could expect that the Blocking Set Problem for two-element sets is in P. We note that this case is especially interesting since the issue here is the blocking of the edges of graphs. We can notice that the nodes outside a blocking set are independent (there is no edge among them). The converse is true in the following sense: if an independent set is maximal (no other node can be added to it while preserving independence) then its complement is a blocking set for the edges. Our search for a minimum Blocking set can therefore be replaced with a search for a maximum independent set, which is also a fundamental graph-theoretical problem.

Remark. The independent vertex set problem (and similarly, the Blocking set problem) is NP-complete only if k is part of the input. It is namely obvious that if we fix k (e.g., $k = 137$) then for a graph of n nodes it can be decided in polynomial time (in the

given example, in time $O(n^{137})$) whether it has k independent nodes. The situation is different with colorability, where already the colorability with 3 colors is NP-complete.

Exercise 4.5.1. *Prove that it is also* NP*-complete to decide whether in a given $2n$-vertex graph, there is an n-element independent set.*

Exercise 4.5.2. *Prove that it is also* NP*-complete to decide whether the chromatic number of a graph G (the smallest number of colors with which its vertices can be colored) is equal to the number of elements of its largest complete subgraph.*

Exercise 4.5.3. *Prove that the covering problem, if every set in the set system is restricted to have at most 2 elements, is reducible to the matching problem.*

Exercise 4.5.4. *Prove that for hypergraphs, already the problem of coloring with two colors is* NP*-complete: Given a system $\{A_1, \ldots, A_n\}$ of subsets of a finite set. Can the nodes of S be colored with two colors in such a way that each A_i contains nodes of both colors?*

From the NP-completeness of the Independent node set problem, we get the NP-completeness of two other basic graph-theory problems for free. First, notice that the complement of an independent set of nodes is a blocking set for the family of edges, and vice versa. Hence we get that the Blocking Set Problem for the family of edges of a graph is NP-complete. (Recall that in the special case when the graph is bipartite, then the minimum size of a blocking set is equal to the size of a maximum matching, and therefore it can be computed in polynomial time.)

Another easy transformation is to look at the complementary graph $\overline{G}$ of G (this is the graph on the same set of nodes, with "adjacent" and "non-adjacent" interchanged). An independent set in G corresponds to a clique (complete subgraph) in $\overline{G}$ and vice versa. Thus the problem of finding a k-element independent set is (trivially) reduced to the problem of finding a k-element clique, so we can conclude that the problem of deciding whether a graph has a clique of size k is also NP-complete.

Many other important combinatorial and graph-theoretical problems are NP-complete:

- Does a given graph have a Hamiltonian circuit?
- Can we cover the nodes with disjoint triangles? (For "2-angles", this is the matching problem!),
- Does there exist a family of k node-disjoint paths connecting k given pairs of nodes?

The book "Computers and Intractability" by Garey and Johnson (Freeman, 1979) lists NP-complete problems by the hundreds.

A number of NP-complete problems are known also outside combinatorics. The most important one among these is the following. In fact, the NP-completeness of this problem was observed (informally, without an exact definition or proof) by Edmonds several years before the Cook–Levin Theorem.

Problem 4.5.6 (Linear Diophantine Inequalities)**.** *Given a system $Ax \leq b$ of linear inequalities with integer coefficients, decide whether it has a solution in integers. (Recall*

that the epithet "Diophantine" indicates that we are looking for the solution among integers.)

Theorem 4.5.5. *The solvability of a Diophantine system of linear inequalities is an* NP*-complete problem.*

Here we only prove that the problem is NP-hard. It is a little more involved to prove that the problem is contained in NP, as we have already mentioned in Section 4.3 at **o)** Existence of an integer solution.

Proof. Let a 3-form B be given over the variables $x_1, \dots, x_n$. Let us take the following inequalities:

$$\begin{aligned} 0 \le x_i \le 1 \text{ for all } i, \\ x_{i_1} + x_{i_2} + x_{i_3} \ge 1 \text{ if } x_{i_1} \vee x_{i_2} \vee x_{i_3} \text{ is in } B, \\ x_{i_1} + x_{i_2} + (1 - x_{i_3}) \ge 1 \text{ if } x_{i_1} \vee x_{i_2} \vee \overline{x}_{i_3} \text{ is in } B, \\ x_{i_1} + (1 - x_{i_2}) + (1 - x_{i_3}) \ge 1 \text{ if } x_{i_1} \vee \overline{x}_{i_2} \vee \overline{x}_{i_3} \text{ is in } B, \\ (1 - x_{i_1}) + (1 - x_{i_2}) + (1 - x_{i_3}) \ge 1 \text{ if } \overline{x}_{i_1} \vee \overline{x}_{i_2} \vee \overline{x}_{i_3} \text{ is in } B. \end{aligned}$$

The solutions of this system of inequalities are obviously exactly the value assignments satisfying B, and so we have reduced the problem 3-SAT to the problem of solvability in integers of systems of linear inequalities. □

We mention that already a very special case of this problem is NP-complete:

Problem 4.5.7 (Subset sum problem)**.** *Given natural numbers $a_1, \dots, a_m$ and b. Does the set $\{a_1, \dots, a_m\}$ have a subset whose sum is b? (The empty sum is 0 by definition.)*

Theorem 4.5.6. *The subset sum problem is* NP*-complete.*

Proof. We reduce the partition problem to the subset sum problem. Let $\{A_1, \dots, A_m\}$ be a family of subsets of the set $S = \{0, \dots, n-1\}$, we want to decide whether it has a subfamily giving a partition of S. Let $q = m + 1$ and let us assign a number $a_i = \sum_{j \in A_i} q^j$ to each set A_i. Further, let $b = 1 + q + \cdots + q^{n-1}$. We claim that $A_{i_1} \cup \cdots \cup A_{i_k}$ is a partition of the set S if and only if

$$a_{i_1} + \cdots + a_{i_k} = b.$$

The "only if" is trivial. Conversely, assume $a_{i_1} + \cdots + a_{i_k} = b$. Let d_j be the number of those sets A_{i_r} that contain the element j $(0 \le j \le n-1)$. Then

$$a_{i_1} + \cdots + a_{i_k} = \sum_j d_j q^j.$$

Each d_j is at most $m = q - 1$, so this gives a representation of the integer b with respect to the number base q. Since $q > m$, this representation is unique, and it follows that $d_j = 1$, i.e., $A_{i_1} \cup \cdots \cup A_{i_k}$ is a partition of S. □

This last problem illustrates nicely that the way we encode numbers can significantly influence the complexity of a problem. Let us assume that each number a_i is encoded in such a way that it requires a_i bits (e.g., with a sequence $1 \cdots 1$ of length a_i). In

short, we say that we use the *unary* notation. The length of the input will increase this way, and therefore the number of steps an algorithm makes on it when measured as a function of the input, will become smaller.

Theorem 4.5.7. *In unary notation, the subset sum problem is polynomially solvable.*

(The general problem of solving linear inequalities in integers is NP-complete even under unary notation; this is shown by the proof of Theorem 4.5.5 where only coefficients with absolute value at most 2 are used.)

Proof. For every p with $1 \le p \le m$, we determine the set T_p of those natural numbers t that can be represented in the form $a_{i_1} + \cdots + a_{i_k}$, where $1 \le i_1 \le \cdots \le i_k \le p$. This can be done using the following trivial recursion:

$$T_0 = \{0\}, \qquad T_{p+1} = T_p \cup \{\, t + a_{p+1} : t \in T_p \,\}.$$

If T_m is found then we must only check whether $b \in T_m$ holds.

We must see yet that this simple algorithm is polynomial. This follows immediately from the observation that $T_p \subseteq \{0, \ldots, \sum_i a_i\}$ and thus the size of the sets T_p is polynomial in the size of the input, which is now $\sum_i a_i$. □

The method of this proof, that of keeping the results of recursive calls to avoid recomputation later, is called *dynamic programming*.

Remarks. 1. A function f is called NP-*hard* if every problem in NP can be reduced to it in the sense that if we add the computation of the value of the function f to the instructions of the Random Access Machine (and thus consider it a single step) then every problem in NP can be solved in polynomial time (the problem itself need not be in NP).

An NP-hard function may or may not be 01-valued (i.e., the characteristic function of a language). The characteristic function of every NP-complete language is NP-hard, but there are languages with NP-hard characteristic functions which are not in NP, and so are strictly harder than any problem in NP (e.g., to decide about a position of the GO game on an $n \times n$ board, who can win).

There are many important NP-hard functions whose values are not 0 or 1. If there is an optimization problem associated with an NP-problem, like in many important discrete optimization problems of operations research, then in case the problem is NP-complete the associated optimization problem is NP-hard. Some examples:

- the famous Traveling Salesman Problem: a non-negative "cost" is assigned to each edge of a graph, and we want to find a Hamiltonian cycle with minimum cost (the cost of a Hamiltonian cycle is the sum of the costs of its edges);
- the Steiner problem (find a connected subgraph of minimum cost (defined as previously, non-negative on each edge) containing a given set of vertices);
- the knapsack problem (the optimization problem associated with a more general version of the subset sum problem);
- a large fraction of scheduling problems.

Many enumeration problems are also NP-hard (e.g., to determine the number of all perfect matchings, Hamiltonian cycles or legal colorings).

2. Most NP problems occurring in practice turn out to be either NP-complete or in P. Nobody succeeded yet to put either into P or among the NP-complete ones the following problems:

BOUNDED DIVISOR. Does a given natural number n have a proper divisor not greater than k?

GRAPH ISOMORPHISM. Are two given graphs isomorphic?

For both problems it is expected that they are neither in P nor NP-complete.

3. When a problem turns out to be NP-complete we cannot hope to find for it such an efficient, polynomial algorithm such as e.g., for the matching problem. Since such problems can be very important in practice we cannot give them up because of such a negative result. Around an NP-complete problem, a mass of partial results of various types are born: special classes for which it is polynomially solvable; algorithms that are exponential in the worst case but are fairly well usable for not too large inputs, or for problems occurring in practice (whether or not we are able to describe the special structure of "real word" problems that make them easy); heuristics, approximation algorithms that do not give exact solution but (provably or in practice) give good approximation. It is, however, sometimes just the complexity of the problems that can be utilized: see Chapter 12.

Exercise 4.5.5. *Show that the Satisfiablity Problem can be reduced to the special case when each variable occurs at least once unnegated and at least once negated.*

Exercise 4.5.6. *In the* GRAPH EMBEDDING PROBLEM*, we are given a pair (G_1, G_2) of graphs. The question is whether G_2 has a subgraph isomorphic to G_1. Prove that this problem is* NP*-complete.*

Exercise 4.5.7. *Prove that if a system of sets is such that every element of the (finite) underlying set belongs to at most two sets of the system, then the Blocking Set Problem for this system is polynomial time solvable.*

[Hint: reduce it to the general matching problem.]

Exercise 4.5.8. *An instance of the problem of 0-1 Integer Programming is defined as follows. The input of the problem is arrays of integers a_{ij}, b_i for $i = 1, \ldots, m$, $j = 1, \ldots, n$. The task is to see if the set of equations*

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad (i = 1, \ldots, m)$$

is satisfiable with $x_j = 0, 1$. The Subset Sum Problem is a special case with $m = 1$.

Make a direct reduction of the 0-1 Integer Programming problem to the Subset Sum Problem.

Exercise 4.5.9. *The* SUM PARTITION PROBLEM *is the following. Given a set $A = \{a_1, \ldots, a_n\}$ of integers, decide whether there exists a set I such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. Prove that this problem is* NP*-complete. [Hint: use the* NP*-completeness of the subset sum problem.]*

Exercise 4.5.10. *The bounded tiling problem $\mathcal{B}$ is the following language. Its words have the form T&n&s. Here, the string T represents a set of tile types (kit) and n is a natural number. The string s represents a sequence of $4n-4$ tiles. The string T&n&s belongs to $\mathcal{B}$ if and only if there is a tiling of an $n \times n$ square with tiles whose type is in T in such a way that the tiles on the boundary of the square are given by s (starting, say, at the lower left corner and going counterclockwise). Prove that the language $\mathcal{B}$ is NP-complete.*

Exercise 4.5.11. *Consider the following tiling problem. We are given a fixed finite set of tile types with a distinguished initial tile among them. Our input is a number n in binary and we have to decide whether an $n \times n$ square can be tiled by tiles of these types, when all four corners must be the initial tile. Prove that there is a set of tiles for which this problem is NEXPTIME-complete.*

Chapter 5

Randomized algorithms

We cited Church's Thesis in Chapter 2: every "algorithm" (in the heuristic meaning of the word) is realizable on a Turing machine. It turned out that other models of computation were able to solve exactly the same class of problems.

But there is an extension of the notion of an algorithm that is more powerful than a Turing machine, and still realizable in the real world. This is the notion of a *randomized algorithm*: we permit "coin tossing", i.e., we have access to a random number generator. Such machines will be able to solve problems that the Turing machine cannot solve (we will formulate and prove this in an exact sense in a Chapter 6); furthermore, such machines can solve some problems more efficiently than Turing machines. We start with a discussion of such examples. The simplest example of such an application of randomization is checking an algebraic identity; the most important is quite certainly testing whether an integer is a prime.

Since in this way, we obtain a new, stronger mathematical notion of a machine, corresponding randomized complexity classes can also be introduced. Some of the most important ones will be treated at the end of the Chapter.

5.1 Verifying a polynomial identity

Let $f(x_1, \ldots, x_n)$ be a rational polynomial with n variables that has degree at most k in each of its variables. We would like to decide whether f is identically 0 (as a function of n variables). We know from classical algebra that a polynomial is identically 0 if and only if, after "opening its parentheses", all terms "cancel". This criterion is, however, not always useful. It is conceivable, e.g., that the polynomial is given in a parenthesized form and the opening of the parentheses leads to exponentially many terms as in

$$(x_1 + y_1)(x_2 + y_2) \cdots (x_n + y_n) + 1.$$

It would also be good to say something about polynomials in whose definition not only the basic algebraic operations occur but also some other ones, like the computation of a determinant (which is a polynomial itself but is often computed, as we have seen, in some special way).

The basic idea is that we write random numbers in place of the variables and compute the value of the polynomial. If this is not 0 then, naturally, the polynomial

cannot be identically 0. If the computed value is 0 then though it can happen that the polynomial is not identically 0, but "hitting" one of its roots has very small probability; therefore in this case we can conclude that the polynomial is identically 0; the probability that we make a mistake is small.

If we could give real values to the variables, chosen according to the uniform distribution e.g., in the interval $[0, 1]$, then the probability of error would be 0. We must in reality, however, compute with discrete values; therefore we assume that the values of the variables are chosen from among the integers of the interval $[0, N-1]$, independently and according to the uniform distribution. In this case, the probability of error will not be 0 but it will be "small" if N is large enough. This is the meaning of the following fundamental result.

Theorem 5.1.1 (Schwartz-Zippel Lemma)**.** *If f is a not identically 0 polynomial in n variables with degree at most k, and the integers ξ_i $(i = 1, \dots, n)$ are chosen in the interval $[0, N-1]$ independently of each other according to the uniform distribution then*

$$\mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0\} \leq \frac{k}{N}.$$

(The degree of a polynomial in several variables is defined as the largest degree of its terms (monomials); the degree of a monomial is the sum of the exponents of the variables in it.)

Proof. We prove the assertion by induction on n. The statement is true for $n = 1$ since a polynomial in one variable of degree at most k can have at most k roots. Let $n > 1$ and let us arrange f according to the powers of x_1:

$$f = f_0 + f_1 x_1 + f_2 x_1^2 + \cdots + f_t x_1^t,$$

where $f_0, \dots, f_t$ are polynomials of the variables $x_2, \dots, x_n$, the term f_t is not identically 0, and $t \leq k$. Now,

$$\begin{aligned}
&\mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0\} \\
&\leq \mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0 \mid f_t(\xi_2, \dots, \xi_n) = 0\} \cdot \mathsf{P}\{f_t(\xi_2, \dots, \xi_n) = 0\} \\
&+ \mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0 \mid f_t(\xi_2, \dots, \xi_n) \neq 0\} \cdot \mathsf{P}\{f_t(\xi_2, \dots, \xi_n) \neq 0\} \\
&\leq \mathsf{P}\{f_t(\xi_2, \dots, \xi_n) = 0\} + \mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0 \mid f_t(\xi_2, \dots, \xi_n) \neq 0\}.
\end{aligned}$$

Here we can estimate the first term by the induction hypothesis, using that the degree of f_t is at most $k - t$; thus the first term is at most $(k-t)/N$. The second term is at most t/N (since ξ_1 is independent of the variables $\xi_2, \dots, \xi_n$, therefore no matter how the latter are fixed in such a way that $f_t \neq 0$ (and therefore f as a polynomial of x_1 is not identically 0), the probability that ξ_1 is its root is at most t/N). Hence

$$\mathsf{P}\{f(\xi_1, \dots, \xi_n) = 0\} \leq \frac{k-t}{N} + \frac{t}{N} \leq \frac{k}{N}.$$

□

This suggests the following *randomized* algorithm to decide whether a polynomial f is identically 0:

Algorithm 5.1.2. We compute $f(\xi_1, \ldots, \xi_n)$ with integer values ξ_i chosen randomly and independently of each other according to the uniform distribution in the interval $[0, 2k]$. If the result is not 0, we stop; f is not identically 0. If we get a 0, we repeat the computation. If 0 is obtained 100 times we stop and declare that f is identically 0.

If f is identically 0 then this algorithm will say so. If f is not identically 0 then in every separate iteration—according to Schwartz-Zippel Lemma—the probability that the result is 0 is less than $1/2$. With 100 experiments repeated independently of each other, the probability that this occurs every time, i.e., that the algorithm asserts erroneously that f is identically 0, is less than 2^{-100}.

Two things are needed for us to be able to actually carry out this algorithm: on the one hand, we must be able to generate random numbers (here, we assume that this can be implemented, and even in time polynomial in the number of bits of the integers to be generated), on the other hand, we must be able to evaluate f in polynomial time (the size of the input is the length of the "definition" of f; this definition can be, e.g., an expression containing multiplications and additions with parentheses, but also something entirely different, e.g., a determinant form).

As a surprising example for the application of the method we present a matching algorithm. (We have already treated the matching problem in the earlier chapters.) Let G be a bipartite graph with the edge set $E(G)$ whose edges run between sets A and B, $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$. Let us assign to each edge $a_i b_j$ a variable x_{ij} and construct the $n \times n$ matrix M as follows:

$$m_{ij} = \begin{cases} x_{ij}, & \text{if } a_i b_j \in E(G), \\ 0, & \text{otherwise.} \end{cases}$$

The determinant of this graph is closely connected to the matchings of the graph G as Dénes Kőnig noticed while analyzing a work of Frobenius:

Theorem 5.1.3. *There is a perfect matching in the bipartite graph G if and only if* $\det(M)$ *is not identically 0.*

Proof. Consider a term in the expansion of the determinant:

$$\pm m_{1\pi(1)} m_{2\pi(2)} \cdots m_{n\pi(n)},$$

where π is a permutation of the numbers $1, \ldots, n$. For this not to be 0, we need that a_i and $b_{\pi(i)}$ be connected for all i; in other words, that $\{a_1 b_{\pi(1)}, \ldots, a_n b_{\pi(n)}\}$ be a perfect matching in G. In this way, if there is no perfect matching in G then the determinant is identically 0. If there are perfect matchings in G then to each one of them a nonzero expansion term corresponds. Since these terms do not cancel each other (any two of them contains at least two different variables), the determinant is not identically 0. □

Since $\det(M)$ is a polynomial of the elements of the matrix M that is computable in polynomial time (e.g., by Gaussian elimination) this theorem offers a polynomial time randomized algorithm for the matching problem in bipartite graphs. We mentioned it before that there is also a polynomial time deterministic algorithm for this problem (the "Hungarian method"). One advantage of the algorithm treated here is that it is very easy to program (determinant-computation can generally be found in the program

library). If we use "fast" matrix multiplication methods then this randomized algorithm is a little faster than the fastest known deterministic one: it can be completed in time $O(n^{2.4})$ instead of $O(n^{2.5})$. Its main advantage is, however, that it is well suitable to parallelization, as we will see in Chapter 10.

In non-bipartite graphs, it can also be decided by a similar but slightly more complicated method whether there is a perfect matching. Let $V = \{v_1, \ldots, v_n\}$ be the vertex set of the graph G. Assign again to each edge v_iv_j (where $i < j$) a variable x_{ij} and construct an asymmetric $n \times n$ matrix $T = (t_{ij})$ as follows:

$$t_{ij} = \begin{cases} x_{ij}, & \text{if } v_iv_j \in E(G), \text{ and } i < j, \\ -x_{ji}, & \text{if } v_iv_j \in E(G), \text{ and } i > j, \\ 0, & \text{otherwise.} \end{cases}$$

The following analogue of the above cited result of Frobenius and Kőnig (Theorem 5.1.3) comes from Tutte and we formulate it here without proof:

Theorem 5.1.4. *There is a perfect matching in the graph G if and only if* $\det(T)$ *is not identically 0.*

This theorem offers, similarly to the case of bipartite graphs, a randomized algorithm for deciding whether there is a perfect matching in G.

5.2 Primality testing

Let m be an odd natural number, we want to decide whether it is a prime. We have seen in the previous chapter that this problem is in NP $\cap$ co−NP. The witnesses described there did not lead, however, (at least for the time being) to a polynomial time prime test. We will therefore give first a new, more complicated NP description of compositeness.

Theorem 5.2.1 ("Little" Fermat Theorem)**.** *If m is a prime then $a^{m-1} - 1$ is divisible by m for all natural numbers $1 \leq a \leq m - 1$.*

If the integer $a^{m-1} - 1$ is divisible by m (for a given m) then we say that a *satisfies the Fermat condition.*

The Fermat condition, when required for all integers $1 \leq a \leq m - 1$, also characterizes primes:

Lemma 5.2.2. *An integer $m > 0$ is a prime if and only if all integers $1 \leq a \leq m - 1$ satisfy the Fermat condition.*

Indeed, if m is composite, then we can choose for a an integer not relatively prime to m, and then $a^{m-1} - 1$ is obviously not divisible by m.

A further nice feature of the Fermat condition is that it can be checked in polynomial time for given m and a. This was discussed in Chapter 4.

Of course, we cannot check the Fermat condition for every a: this would take exponential time. The question is therefore to which a's should we apply it?

We could just try $a = 2$. This is in fact not a bad test, and it reveals the non-primality of many (in a sense, most) composite numbers, but it may fail. For example,

$561 = 3 \cdot 11 \cdot 17$ is not a prime, but $561 \mid 2^{560} - 1$. Any other specific choice of a would have similar shortcoming.

The next idea is to select a random a and check the Fermat condition. If m is a prime, then a will of course satisfy it. Suppose that m is not prime, then at least those a's not relatively prime to m will violate it. Unfortunately, the number of such a's may be minuscule compared to the number of all choices for a, and so the probability that our random choice will pick one is negligible. (In fact, we can compute the greatest common divisor of a and m right away: if we can find an a not relatively prime to m, then this will yield a proper divisor of m.)

So we need to use a's relatively prime to m. Unfortunately, there are composite numbers m (the so-called pseudoprimes) for which the Fermat condition is satisfied for *all* a relatively prime to m; for such numbers, it will be especially difficult to find an integer a violating the condition. (Such a pseudoprime is e.g., $561 = 3 \cdot 11 \cdot 17$.)

But at least if m is not a pseudoprime, then the random choice for a works. This is guaranteed by the following lemma.

Lemma 5.2.3. *If m is not a prime and not a pseudoprime then at most half of the integers a, $1 \leq a \leq m-1$ relatively prime to m satisfies the Fermat condition.*

Note that none of the non-relatively-prime a's satisfy the Fermat condition.

Proof. Since m is not a pseudoprime, there is at least one b relatively prime to m such that $b^{m-1} - 1$ is not divisible by m. Now if a is "bad", i.e., $a^{m-1} - 1$ is divisible by m, then $ab \bmod m$ is "good":

$$(ab)^{m-1} - 1 = (a^{m-1} - 1)b^{m-1} + b^{m-1} - 1,$$

and here the first term is divisible by m but the second is not.

Hence for every a that is "bad", we find another a' (namely $ab \bmod m$) that is "good". It is easy to see that different a's yield different "good" numbers. Thus at least half of the numbers a must be good. □

Thus, if m is not a pseudoprime then the following randomized prime test works: check whether a randomly chosen integer $1 \leq a \leq m-1$ satisfies the Fermat condition. If not then we know that m is not a prime. If yes then repeat the procedure. If we found 100 times, independently of each other, that the Fermat condition is satisfied then we say that m is a prime. It can still happen that m is composite, but the probability that we picked an integer a satisfying the condition is less than $1/2$ at every step, and hence the probability that this occurs 100 times in a row is less than 2^{-100}.

Unfortunately, this method fails for pseudoprimes (it declares them prime with large probability). It turns out that one can modify the Fermat condition just a little to overcome this difficulty. Let us write the number $m-1$ in the form $2^k M$ where M is odd. We say that *a satisfies the Miller-Rabin condition* if at least one of the numbers

$$a^M - 1, a^M + 1, a^{2M} + 1, a^{4M} + 1, \ldots, a^{2^{k-1}M} + 1$$

is divisible by m. Note that the product of these numbers is $a^{m-1} - 1$. Hence every number satisfying the Miller-Rabin condition also satisfies the Fermat condition.

If m is a prime then it divides this product, and hence it divides one of these factors, i.e., every a satisfies the Miller-Rabin condition. If m is composite then, however, it

could happen that some a satisfies the Fermat condition but not the Miller-Rabin condition (m can be a divisor of a product without being the divisor of any of its factors).

Thus the Miller-Rabin condition provides a potentially stronger primality test than the Fermat condition. The question is: how much stronger?

We will need some fundamental facts about pseudoprimes.

Lemma 5.2.4. *Every pseudoprime m is*
(a) *Odd*
(b) *Squarefree (not divisible by any square).*

Proof. (a) If $m > 2$ is even then $a = m - 1$ will violate the Fermat condition, since $(m-1)^{m-1} \equiv -1 \not\equiv 1 \pmod{m}$.

(b) Assume that $p^2 \mid m$; let k be the largest exponent for which $p^k \mid m$. Then $a = m/p - 1$ violates the Fermat condition since the last two terms of the binomial expansion of $(m/p - 1)^{m-1}$ are $-(m-1)(m/p) + 1 \equiv m/p + 1 \not\equiv 1 \pmod{p^k}$ since all earlier terms are divisible by p^k. If an integer is not divisible by p^k then it is not divisible by m either, thus we are done. □

Lemma 5.2.5. *Let $m = p_1 p_2 \cdots p_t$ where the p_i's are different primes. The relation $a^{m-1} \equiv 1 \pmod{m}$ holds for all a relatively prime to m if and only if $p_i - 1$ divides $m - 1$ for all i with $1 \le i \le t$.*

Proof. If $p_i - 1$ divides $m - 1$ for all i with $1 \le i \le t$ then $a^{m-1} - 1$ is divisible by p_i according to the little Fermat Theorem and then it is also divisible by m. Conversely, suppose that $a^{m-1} \equiv 1 \pmod{m}$ for all a relatively prime to m. If e.g., $p_1 - 1$ would not divide $m - 1$ then let g be a primitive root modulo p_1 (the existence of primitive roots was stated in Theorem 4.3.5). According to the Chinese Remainder Theorem, there is a residue class h modulo m with $h \equiv g \pmod{p_1}$ and $h \equiv 1 \pmod{p_i}$ for all $i \ge 2$. Then $(h, m) = 1$ and $p_1 \not| h^{m-1} - 1$, so $m \not| h^{m-1} - 1$. □

Corollary 5.2.6. *The number m is a pseudoprime if and only if $m = p_1 p_2 \cdots p_t$ where the p_i's are different primes, $t \ge 2$, and $(p_i - 1)$ divides $(m - 1)$ for all i with $1 \le i \le t$.*

Remark. This is how one can show about the above example, 561, that it is a pseudoprime.

Now we can prove the main fact that makes the Miller-Rabin test better than the Fermat test.

Theorem 5.2.7. *If m is a composite number then at least half of the numbers $1, \ldots, m - 1$ violate the Miller-Rabin condition.*

Proof. Since we have already seen that the lemma holds for non-pseudoprimes, in what follows we can assume that m is a pseudoprime. Let $p_1 \cdots p_t$ ($t \ge 2$) be the prime decomposition of m. By the above, these primes are all odd and distinct, and we have $(p_i - 1) \mid (m - 1) = 2^k M$ for all i with $1 \le i \le t$.

Let l be the largest exponent with the property that none of the numbers $p_i - 1$ divides $2^l M$. Since the numbers $p_i - 1$ are even while M is odd, such an exponent exists (e.g., 0) and clearly $0 \le l < k$. Further, by the definition of l, there is a j for

which $p_j - 1$ divides $2^{l+1}M$. Therefore $p_j - 1$ divides $2^s M$ for all s with $l < s \leq k$, and hence p_j divides $a^{2^s M} - 1$ for all primitive residue classes a. Consequently p_j cannot divide $a^{2^s M} + 1$ which is larger by 2, and hence m does not divide $a^{2^s M} + 1$ either. If therefore a is a residue class that does not violate the Miller-Rabin condition then m must already be a divisor of one of the remainder classes $a^M - 1, a^M + 1, a^{2M} + 1, \ldots, a^{2^{l-1}M} + 1, a^{2^l M} + 1$. Hence for each such a, the number m divides either the product of the first $l + 1$, which is $(a^{2^l M} - 1)$, or the last one, $(a^{2^l M} + 1)$. Let us call the primitive residue class a modulo m an "accomplice of the first kind" if $a^{2^l M} \equiv 1 \pmod{m}$ and an "accomplice of the second kind" if $a^{2^l M} \equiv -1 \pmod{m}$.

Let us estimate first the number of accomplices of the first kind. Consider an index i with $1 \leq i \leq t$. Since $p_i - 1$ does not divide the exponent $2^l M$, Theorem 4.3.5 implies that there is a number c not divisible by p_i for which $c^{2^l M} - 1$ is not divisible by p_i. The reasoning of Lemma 5.2.3 shows that then at most half of the mod p_i residue classes will satisfy the Fermat condition belonging to the above exponent, i.e., such that $a^{2^l M} - 1$ is divisible by p_i. According to the Chinese Remainder Theorem, there is a one-to-one correspondence between the primitive residue classes with respect to the product $p_1 \cdots p_t$ as modulus and the t-tuples of primitive residue classes modulo the primes $p_1, \ldots, p_t$. Thus, modulo $p_1 \cdots p_t$, at most a 2^t-th fraction of the primitive residue classes is such that every p_i divides $(a^{2^l M} - 1)$. Therefore, at most a 2^t-th fraction of the mod m primitive residue classes are accomplices of the first kind.

It is easy to see that the product of two accomplices of the second kind is one of the first kind. Hence multiplying all accomplices of the second kind by a fixed one of the second kind, we obtain accomplices of the first kind, and thus the number of accomplices of the second kind is at least as large as the number of accomplices of the first kind. (If there is no accomplice of the second kind to multiply with then the situation is even better: zero is certainly not greater than the number of accomplices of the first kind.) Hence even the two kinds together make up at most a 2^{t-1}-th part of the primitive residue classes, and so (due to $t \geq 2$) at most a half. □

Lemma 5.2.8. *For a given m and a, it is decidable in polynomial time whether a satisfies the Miller-Rabin condition.*

For this, it is enough to recall from Chapter 3 that the remainder of a^b modulo c is computable in polynomial time. Based on these three lemmas, the following randomized algorithm, called the Miller–Rabin test, can be given for prime testing:

Algorithm 5.2.9. Choose a number between 1 and $m-1$ randomly and check whether it satisfies the Miller-Rabin condition. If it does not then m is composite. If it does then choose another a. If the Miller-Rabin condition is satisfied 100 times consecutively then we declare that m is a prime.

If m is a prime then the algorithm will certainly assert this. If m is composite then the number a chosen randomly violates the Miller-Rabin condition with probability at least $1/2$. After hundred independent experiments the probability will therefore be at most 2^{-100} that the Miller-Rabin condition is not violated even once, i.e., that the algorithm asserts that m is a prime.

Remarks. 1. If m is found composite by the algorithm then, interestingly enough, we see this not from finding a divisor but from the fact that one of the residues violates

the Miller-Rabin condition. If at the same time, the residue a does not violate the Fermat condition, then m cannot be relatively prime to each of the numbers $a^M - 1$, $a^M + 1$, $a^{2M} + 1$, $a^{4m} + 1$, $\cdots$, $a^{2^{k-1}M} + 1$, therefore computing its greatest common divisors with each one of them will be a proper divisor of m. No polynomial algorithm (either deterministic or randomized) is known for finding a factorization in the case when the Fermat condition is also violated. This problem appears to be significantly more difficult also in practice than the testing of primality. We will see in the Chapter 12 that this empirical fact has important applications.

2. For a given m, we can try to find an integer a violating the Miller-Rabin condition not by random choice but by trying out the numbers 1,2, etc. It is not known how small is the first such integer if m is composite. Using, however, a hundred year old conjecture of analytic number theory, the so-called Generalized Riemann Hypothesis, (which is too technical to be formulated here) one can show that it is not greater than $\log m$. Thus, this deterministic prime test works in polynomial time if the Generalized Riemann Hypothesis is true.

We can use the prime testing algorithm learned above to look for a prime number with n digits (say, in the binary number system). Choose, namely, a number k randomly from the interval $[2^{n-1}, 2^n - 1]$ and check whether it is a prime, say, with an error probability of at most $2^{-100}/n$. If it is, we stop. If it is not we choose a new number k. Now, it follows from the theory of prime numbers that in this interval, not only there is a prime number but the number of primes is rather large: asymptotically $(\log e)2^{n-1}/n$, i.e., a randomly chosen n-digit number will be a prime with probability about $1.44/n$. Repeating therefore this experiment $O(n)$ times we find a prime number with very large probability.

We can choose a random prime similarly from any sufficiently long interval, e.g., from the interval $[1, 2^n]$.

5.3 Randomized complexity classes

In the previous sections, we treated algorithms that used random numbers. Now we define a class of problems solvable by such algorithms.

First we define the corresponding machine. A *randomized Turing machine* is a deterministic Turing machine which has, besides the usual (input-, work- and result-) tapes, also a tape on whose every cell a bit (say, 0 or 1) is written that is selected randomly with probability 1/2. The bits written on the different cells are mutually independent. The machine itself works deterministically but its computation depends, of course, on chance (on the symbols written on the random tape).

Every legal computation of a randomized Turing machine has some probability. We say that a randomized Turing machine *weakly decides* (or, *decides in the Monte-Carlo sense*) a language $\mathcal{L}$ if for all inputs $x \in \Sigma^*$, it stops with probability at least 3/4 in such a way that in case of $x \in \mathcal{L}$ it writes 1 on the result tape, and in case of $x \notin \mathcal{L}$, it writes 0 on the result tape. Shortly: the probability that it gives a wrong answer is at most 1/4.

In our examples, we used randomized algorithms in a stronger sense: they could err

only in one direction. We say that a randomized Turing machine *accepts* a language $\mathcal{L}$ if for all inputs x, it always rejects the word x in case of $x \notin \mathcal{L}$, and if $x \in \mathcal{L}$ then the probability is at least $1/2$ that it accepts the word x.

We say that a randomized Turing machine *strongly decides* (or, *decides in the Las Vegas sense*) a language $\mathcal{L}$ if it gives a correct answer for each word $x \in \Sigma^*$ with probability 1. (Every single computation of finite length has positive probability and so the 0-probability exception cannot be that the machine stops with a wrong answer, only that it works for an infinite time.)

In case of a randomized Turing machine, for each input, we can distinguish the number of steps in the longest computation and the expected number of steps. The class of all languages that are weakly decidable on a randomized Turing machine in polynomial expected time is denoted by BPP (Bounded Probability Polynomial). The class of languages that can be accepted on a randomized Turing machine in polynomial expected time is denoted by RP (Random Polynomial). The class of all languages that can be strongly decided on a randomized Turing machine in polynomial expected time is denoted by ZPP. Obviously, BPP $\supseteq$ RP $\supseteq$ ZPP $\supseteq$ P.

The constant $3/4$ in the definition of weak decidability is arbitrary: we could say here any number smaller than 1 but greater than $1/2$ without changing the definition of the class BPP (it cannot be $1/2$: with this probability, we can give a correct answer by coin-tossing). If the machine gives a correct answer with probability $1/2 < c < 1$ then let us repeat the computation t times on input x and accept as answer the one given more often. It is easy to see from the Law of Large Numbers that the probability that this answer is wrong is less than c_1^t where c_1 is a constant smaller than 1 depending only on c. For sufficiently large t this can be made arbitrarily small and this changes the expected number of steps only by a constant factor.

It can be similarly seen that the constant $1/2$ in the definition of acceptance can be replaced with an arbitrary positive number smaller than 1.

Finally, we note that instead of the expected number of steps in the definition of the classes BPP and RP, we could also consider the largest number of steps; this would still not change the classes. Obviously, if the largest number of steps is polynomial, then so is the expected number of steps. Conversely, if the expected number of steps is polynomial, say, at most $|x|^d$, then according to Markov's Inequality, the probability that a computation lasts a longer time than $8|x|^d$ is at most $1/8$. We can therefore build in a counter that stops the machine after $8|x|^d$ steps, and writes 0 on the result tape. This increases the probability of error by at most $1/8$.

The same is, however, not known for the class ZPP: the restriction of the longest running time would lead here already to a deterministic algorithm, and it is not known whether ZPP is equal to P (moreover, this is rather expected not to be the case; there are examples for problems solvable by polynomial Las Vegas algorithms for which no polynomial deterministic algorithm is known).

Remark. We could also define a randomized Random Access Machine: this would have an extra cell w in which there is always a 0 or 1 with probability $1/2$. We have to add the instruction $y := w$ to the programming language. Every time this is executed a new random bit occurs in the cell w that is completely independent of the previous bits. Again, it is not difficult to see that this does not bring any significant difference.

It can be seen that every language in RP is also in NP. It is trivial that the classes BPP and ZPP are closed with respect to the taking of complement: they contain, together with every language $\mathcal{L}$ the language $\Sigma^* \setminus \mathcal{L}$. The definition of the class RP is not such and it is not known whether this class is closed with respect to complement. It is therefore worth defining the class co−RP: A language $\mathcal{L}$ is in co−RP if $\Sigma^* \setminus \mathcal{L}$ is in RP.

"Witnesses" provided a useful characterization of the class NP. An analogous theorem holds also for the class RP.

Theorem 5.3.1. *A language $\mathcal{L}$ is in* RP *if and only if there is a language $\mathcal{L}' \in$ P and a polynomial $f(n)$ such that*

(i) $\mathcal{L} = \{ x \in \Sigma^* : \exists y \in \Sigma^{f(|x|)} \; x\&y \in \mathcal{L}' \}$ *and*

(ii) if $x \in \mathcal{L}$ then at least half of the words y of length $f(|x|)$ are such that $x\&y \in \mathcal{L}'$.

Proof. Similar to the proof of the corresponding theorem on NP (Theorem 4.2.1). □

The connection of the classes RP and ΔRP is closer than it could be expected on the basis of the analogy to the classes NP and P:

Theorem 5.3.2. *The following properties are equivalent for a language $\mathcal{L}$:*

(i) $\mathcal{L} \in$ ZPP*;*

(ii) $\mathcal{L} \in RP \cap$ co−RP*;*

(iii) *There is a randomized Turing machine with polynomial (worst-case) running time that can write, besides the symbols "0" and "1", also the words "*I GIVE UP*"; the answers "0" and "1" are never wrong, i.e., in case of $x \in \mathcal{L}$ the result is "1" or "*I GIVE UP*", and in case of $x \notin \mathcal{L}$ it is "0" or "*I GIVE UP*". The probability of the answer "*I GIVE UP*" is at most* 1/2*.*

Proof. It is obvious that (i) implies (ii). It can also be easily seen that (ii) implies (iii). Let us submit x to a randomized Turing machine that accepts $\mathcal{L}$ in polynomial time and also to one that accepts $\Sigma^* \setminus \mathcal{L}$ in polynomial time. If the two give opposite answers then the answer of the first machine is correct. If they give identical answers then we "give it up". In this case, one of them made an error and therefore this has a probability at most 1/2.

Finally, to see that (iii) implies (i) we just have to modify the Turing machine T_0 given in (iii) in such a way that instead of the answer "I GIVE IT UP", it should start again. If on input x, the number of steps of T_0 is τ and the probability of giving it up is p then on this same input, the expected number of steps of the modified machine is

$$\sum_{t=1}^{\infty} p^{t-1}(1-p)t\tau = \frac{\tau}{1-p} \leq 2\tau.$$

□

For our example about polynomials not identically 0, it is only known that the decision problem is in RP and not known whether it belongs to ZPP or P. Among the algebraic (mainly group-theoretical) problems, there are many that are in RP or ZPP but no polynomial algorithm is known for their solution.

Remark. The algorithms that use randomization should not be confused with the algorithms whose performance (e.g., the expected value of their number of steps) is being examined for random inputs. Here we did not assume any probability distribution on the set of inputs, but considered the worst case. The investigation of the behavior of algorithms on random inputs coming from a certain distribution is an important but difficult area, still in its infancy, that we will not treat here.

Exercise 5.3.1. *Suppose that some experiment has some probability p of success. Prove that in n^3 experiments, it is possible to compute an approximation $\hat{p}$ of p such that the probability of $|p-\hat{p}| > \sqrt{p(1-p)}/n$ is at most $1/n$. [Hint: Use Chebychev's Inequality.]*

Exercise 5.3.2. *We want to compute a real quantity a. Suppose that we have a randomized algorithm that computes an approximation A (which is a random variable) such that the probability that $|A - a| > 1$ is at most $1/20$. Show that by calling the algorithm t times, you can compute an approximation B such that the probability that $|B - a| > 1$ is at most 2^{-t}.*

Exercise 5.3.3. *Suppose that somebody gives you three $n \times n$ matrices A, B, C (of integers of maximum length l) and claims $C = AB$. You are too busy to verify this claim exactly and instead do the following. You choose a random vector x of length n whose entries are integers chosen uniformly from some interval $[0, \ldots, N-1]$, and check $A(Bx) = Cx$. If this is true you accept the claim otherwise you reject it.*

- *How large must N be chosen to make the probability of false acceptance smaller than 0.01?*
- *Compare the time complexity the probabilistic algorithm to the one of the deterministic algorithm computing AB.*

Exercise 5.3.4. *Show that if m is a pseudoprime then the Miller–Rabin test not only discovers this with large probability but it can also be used to find a decomposition of m into two factors.*

Exercise 5.3.5. *Formulate what it means that a randomized RAM accepts a certain language in polynomial time and show that this is equivalent to the fact that some randomized Turing machine accepts it.*

Exercise 5.3.6. *Let us call a Boolean formula with n variables robust if it is either unsatisfiable or has at least $2^n/n^2$ satisfying assignments. Give a probabilistic polynomial algorithm to decide the satisfiability of robust formulas.*

Chapter 6

Information complexity: the complexity-theoretic notion of randomness

The mathematical foundation of probability theory appears among the already mentioned famous problems of Hilbert formulated in 1900. Von Mises made an important attempt in 1919 to define the randomness of a 0-1 sequence. His attempt can be sketched as follows. We require that the frequency of 0's and 1's be approximately the same. This is clearly not enough, but we can require the same to hold also if we select every second number of the sequence. More generally, we can require the same for all subsequences obtained by selecting indices from an arithmetic progression. This approach, however, did not prove sufficiently fruitful.

In 1931 Kolmogorov initiated another approach, using measure theory. His theory was very successful from the point of view of probability theory, and it is the basis of the rigorous development of probability theory in any textbook today.

However, this standard approach fails to capture some important aspects. For example, in probability theory based on measure theory, we cannot speak of the randomness of a single 0-1 sequence, only of the probability of a set of sequences. At the same time, in an everyday sense, it is "obvious" that the sequence "Head, Head, Head,..." cannot be the result of coin tossing. In the 1960's Kolmogorov and independently Chaitin revived the idea of von Mises, using complexity-theoretic tools. They defined the *information complexity (information content)* of a sequence; then (roughly speaking) random sequences are those whose information content is as large as possible. The importance of these results goes beyond the foundation of probability theory; it contributes to the clarification of the basic notions in several fields like data compression, information theory and statistics.

In this chapter we introduce the notion of *information complexity* first. Next we treat the notion of *self-delimiting information complexity*. While this may seem just a technical variant of the basic notion, it is important because (unlike in the case of, say, Turing machines, where such variants turn out to be equivalent) self-delimiting information complexity has quite different properties in many respects. Then we discuss the notion of an *informatically random sequence*, and show that such sequences behave like "usual" random sequences: they obey the Laws of Large Numbers. Finally, we

discuss the problem of *optimal encoding* of various structures.

6.1 Information complexity

Fix an alphabet Σ. Let $\Sigma_0 = \Sigma \setminus \{*\}$. It will be convenient to identify Σ_0 with the set $\{0, 1, \ldots, m-1\}$. Consider a 2-tape, universal Turing machine T over Σ. We say that the word (program) q over Σ_0 *prints* word x if writing q on the second tape (the program tape) of T and leaving the first tape empty, the machine stops in finitely many steps with the word x on its first tape (the data tape).

Let us note right away that every word is printable on T. There is namely a one-tape (perhaps large, but rather trivial) Turing machine S_x that, when started with the empty tape, writes the word x onto it and halts. This Turing machine can be simulated by a program q_x that, in this way, prints x.

The *information complexity* (also called Kolmogorov complexity) of a word $x \in \Sigma_0^*$ we mean the length of the shortest word (program) that makes T print the word x. We denote the complexity of the word x by $\mathbf{K}_T(x)$.

We can also consider the program printing x as a "code" of the word x where the Turing machine T performs the decoding. This kind of code will be called a *Kolmogorov code*. For the time being, we make no assumptions about how much time this decoding (or encoding, finding the appropriate program) can take.

We would like the complexity to be a characteristic property of the word x and to depend on the machine T as little as possible. It is, unfortunately, easy to make a Turing machine that is obviously "clumsy". For example, it uses only every second letter of each program and "skips" the intermediate letters. Such a machine can be universal, but every word will be defined twice as complex as on the machine without this strange behavior.

We show that if we impose some, rather simple, conditions on the machine T then it will no longer be essential which universal Turing machine is used for the definition of information complexity. Roughly speaking, it is enough to assume that every input of a computation performable on T can also be submitted as part of the program. To make this more exact, we assume that there is a word (say, DATA) for which the following holds:

(a) Every one-tape Turing machine can be simulated by a program that does not contain the word DATA as a subword;

(b) If T is started so that its program tape contains a word of the form $x\text{DATA}y$ where the word x does not contain the subword DATA, then the machine halts if and only if it halts when started with y written on the data tape and x on the program tape, and in fact with the same output on the data tape.

It is easy to see that every universal Turing machine can be modified to satisfy the assumptions (a) and (b). In what follows, we will always assume that our universal Turing machine has these properties.

Lemma 6.1.1. *There is a constant c_T (depending only on T) such that $\mathbf{K}_T(x) \leq |x| + c_T$.*

Proof. T is universal, therefore the (trivial) one-tape Turing machine that does nothing (stops immediately) can be simulated on it by a program p_0 (not containing the word DATA). But then, for every word $x \in \Sigma_0^*$, the program $p_0\text{DATA}x$ will print the word x and stop. Thus the constant $c_T = |p_0| + 4$ satisfies the conditions. □

Remark. We had to be a little careful since we did not want to restrict what symbols can occur in the word x. In the BASIC programming language, for example, the instruction `PRINT "`x`"` is not good for printing words x that contain the symbol ". We are interested in knowing how concisely the word x can be coded *in the given alphabet*, and we do not allow therefore the extension of the alphabet.

We prove now the basic theorem showing that the complexity (under the above conditions) does not depend too much on the given machine.

Theorem 6.1.2 (Invariance Theorem)**.** *Let T and S be universal Turing machines satisfying conditions $(a), (b)$. Then there is a constant c_{TS} such that for every word x we have $|\mathbf{K}_T(x) - \mathbf{K}_S(x)| \leq c_{TS}$.*

Proof. We can simulate the two-tape Turing machine S by a one-tape Turing machine S_0 in such a way that if on S, a program q prints a word x then writing q on the single tape of S_0, it also stops in finitely many steps, with x printed on its tape. Further, we can simulate the work of Turing machine S_0 on T by a program p_{S_0} that does not contain the subword DATA.

Now let x be an arbitrary word from Σ_0^* and let q_x be a shortest program printing x on S. Consider the program $p_{S_0}\text{DATA}q_x$ on T: this obviously prints x and has length only $|q_x| + |p_{S_0}| + 4$. The inequality in the other direction is obtained similarly. □

On the basis of this lemma, we will not restrict generality if we consider T fixed and do not indicate the index T. So, $K(x)$ is determined up to an additive constant.

Unfortunately, the following theorem shows that in general the optimal code cannot be found algorithmically.

Theorem 6.1.3. *The function $\mathbf{K}(x)$ is not recursive.*

Proof. The essence of the proof is a classical logical paradox, the so-called typewriter-paradox. (This can be formulated simply as follows: let n be the smallest number that cannot be defined with fewer than 100 symbols. We have just defined n with fewer than 100 symbols!)

Assume, by way of contradiction, that $\mathbf{K}(x)$ is computable. Let c be a natural number to be chosen appropriately. Arrange the elements of Σ_0^* in increasing order, and let $x(k)$ denote the k-th word according to this ordering. Let x_0 be the first word with $\mathbf{K}(x_0) \geq c$. Assuming that our language can be programmed in the programming language Pascal let us consider the following simple program.

```
var k : integer;
function x(k : integer) : integer;
⋮
function Kolm(k : integer) : integer;
⋮
begin
  k := 0;
  while Kolm(k) < c do k := k + 1;
  print(x(k));
end.
```

(The dotted parts stand for subroutines computing the given functions. The first is easy and could be explicitly included. The second is hypothetical, based on the assumption that $\mathbf{K}(x)$ is computable.)

This program obviously prints x_0. When determining its length we must take into account the subroutines for the computation of the functions $x(k)$ and $Kolm(k) = \mathbf{K}(x(k))$ (where $x(k)$ is the k-th string); but this is a constant (independent of c). Thus the total number of all these symbols is only $\log c + O(1)$. If we take c large enough, this program consists of fewer than c symbols and prints x_0, which is a contradiction. □

As a simple application of the theorem, we get a new proof for the undecidability of the halting problem. To this end, let's ask the following question: Why is it not possible to compute $\mathbf{K}(x)$ as follows? Take all words y in increasing order and check whether T prints x when started with y on its program tape. Return the first y for which this happens; its length is $\mathbf{K}(x)$.

We know that something must be wrong here, since $\mathbf{K}(x)$ is not computable. The only trouble with this algorithm is that T may never halt on some y. If the halting problem were decidable, we could "weed out" in advance the programs on which T would work forever, and not even try these. Thus we could compute $\mathbf{K}(x)$, therefore, the halting problem is not decidable.

Exercise 6.1.1. *Show that we cannot compute the function $\mathbf{K}(x)$ even approximately, in the following sense: If f is a recursive function then there is no algorithm that for every word x computes a natural number $\gamma(x)$ such that for all x*

$$\mathbf{K}(x) \leq \gamma(x) \leq f(\mathbf{K}(x)).$$

Exercise 6.1.2. *Show that there is no algorithm that for every given number n constructs a 0-1 sequence of length n with $\mathbf{K}(x) > 2 \log n$.*

Exercise 6.1.3. *If $f : \Sigma_0^* \to \mathbf{Z}_+$ is a recursive function such that $f(x) \leq \mathbf{K}(x)$ for all strings x, then f is bounded.*

In contrast to Theorem 6.1.3, we show that the complexity $\mathbf{K}(x)$ can be very well approximated *on the average*.

For this, we must first make it precise what we mean by "on the average". Assume that the input words come from some probability distribution; in other words, every

word $x \in \Sigma_0^*$ has a probability $p(x)$. Thus

$$p(x) \geq 0, \qquad \sum_{x \in \Sigma_0^*} p(x) = 1.$$

We assume that $p(x)$ is *computable*, i.e., each $p(x)$ is a rational number whose numerator and denominator are computable from x. A simple example of a computable probability distribution is $p(x_k) = 2^{-k}$ where x_k is the k-th word in increasing order, or $p(x) = (m+1)^{-|x|-1}$ where m is the alphabet size.

Remark. There is a more general notion of a computable probability distribution that does not restrict probabilities to rational numbers; for example, $\{e^{-1}, 1-e^{-1}\}$ could also be considered a computable probability distribution. Without going into details we remark that our theorems would also hold for this more general class.

Theorem 6.1.4. *For every computable probability distribution there is an algorithm computing a Kolmogorov code $f(x)$ for every word x such that the expectation of $|f(x)| - \mathbf{K}(x)$ is finite.*

Proof. For simplicity of presentation, assume that $p(x) > 0$ for every word x. Let $x_1, x_2, \ldots$ be an ordering of the words in Σ_0^* for which $p(x_1) \geq p(x_2) \geq \cdots$, and the words with equal probability are, say, in increasing order (since each word has positive probability, for every x there are only a finite number of words with probability at least $p(x)$, and hence this is indeed a single sequence).

Proposition 6.1.5. (a) *Given a word x, the index i for which $x = x_i$ is computable.*

(b) *Given a natural number i, the word x_i is computable.*

Proof. (a) Let $y_1, y_2, \ldots$ be all words arranged in increasing order. Given a word x, it is easy to find the index j for which $x = y_j$. Next, find the first $k \geq j$ for which

$$p(y_1) + \cdots + p(y_k) > 1 - p(y_j). \tag{6.1}$$

Since the left-hand side converges to 1 while the right-hand side is less than 1, this will occur sooner or later.

Clearly each of the remaining words $y_{k+1}, y_{k+2}, \ldots$ has probability less than $p(y_j)$, and hence to determine the index of $x = y_j$ it suffices to order the finite set $\{y_1, \ldots, y_k\}$ according to decreasing p, and find the index of y_j among them.

(b) Given an index i, we can compute the indices of $y_1, y_2, \ldots$ using (a) and wait until i shows up. □

Returning to the proof of the theorem, the program of the algorithm in the above proposition, together with the number i, provides a Kolmogorov code $f(x_i)$ for the word x_i. We show that this code satisfies the requirements of the theorem. Obviously, $|f(x)| \geq \mathbf{K}(x)$.

Furthermore, the expected value of $|f(x)| - \mathbf{K}(x)$ is

$$\sum_{i=1}^{\infty} p(x_i)(|f(x_i)| - \mathbf{K}(x_i)).$$

We want to show that this sum is finite. Since its terms are non-negative, it suffices to show that its partial sums remain bounded, i.e., that

$$\sum_{i=1}^{N} p(x_i)(|f(x_i)| - \mathbf{K}(x_i)) < C$$

for some C independent of N. We can express this sum as

$$\sum_{i=1}^{N} p(x_i)(|f(x_i)| - \log_m i) + \sum_{i=1}^{N} p(x_i)(\log_m i - \mathbf{K}(x_i)). \tag{6.2}$$

We claim that both sums are bounded. The difference $|f(x_i)| - \log_m i$ is just the length of the program computing x_i without the length of the parameter i, and hence it is an absolute constant C. Thus the first sum in (6.2) is at most C.

To estimate the second term in (6.2), we use the following simple but useful principle. Let $a_1 \geq a_2 \geq \cdots \geq a_m$ be a decreasing sequence and let $b_1, \ldots, b_m$ be an arbitrary sequence of real numbers. Let $b_1^* \geq \cdots \geq b_m^*$ be the sequence b ordered decreasingly, and let $b_1^{**} \leq \cdots \leq b_m^{**}$ be the sequence b ordered increasingly. Then

$$\sum_i a_i b_i^{**} \leq \sum_i a_i b_i \leq \sum_i a_i b_i^*.$$

Let $(z_1, z_2, \ldots,)$ be an ordering of the words so that $\mathbf{K}(z_1) \leq \mathbf{K}(z_2) \leq \ldots$ (we can't compute this ordering, but we don't have to compute it). Then by the above principle,

$$\sum_{i=1}^{N} p(x_i)\mathbf{K}(x_i) \geq \sum_{i=1}^{N} p(x_i)\mathbf{K}(z_i).$$

The number of words x with $\mathbf{K}(x) = k$ is at most m^k, and hence the number of words x with $\mathbf{K}(x) \leq k$ is at most $1 + m + \cdots + m^k < m^{k+1}$. This is the same as saying that

$$i \leq m^{\mathbf{K}(z_i)+1},$$

and hence

$$\mathbf{K}(z_i) \geq \log_m i - 1.$$

Thus

$$\sum_{i=1}^{N} p(x_i)(\log_m i - \mathbf{K}(x_i)) \leq \sum_{i=1}^{N} p(x_i)(\log_m i - \mathbf{K}(z_i)) \leq \sum_{i=1}^{N} p(x_i) = 1.$$

This proves the theorem. □

6.2 Self-delimiting information complexity

The Kolmogorov-code, strictly taken, uses an extra symbol besides the alphabet Σ_0: it recognizes the end of the program while reading the program tape by encountering the symbol "$*$". We can modify the concept in such a way that this should not be possible: the head reading the program should not run beyond program. We will call a word *self-delimiting* if, when it is written on the program tape of our two-tape universal Turing machine, the head does not even try to read any cell beyond it. The length of the shortest self-delimiting program printing x will be denototed by $\mathbf{H}_T(x)$. This modified information complexity notion was introduced by Levin and Chaitin. It is easy to see that the Invariance Theorem here also holds and therefore it is again justified to drop the subscript and use the notation $\mathbf{H}(x)$. The functions $\mathbf{K}$ and $\mathbf{H}$ do not differ too much, as it is shown by the following lemma:

Lemma 6.2.1.

$$\mathbf{K}(x) \le \mathbf{H}(x) \le \mathbf{K}(x) + 2\log_m(\mathbf{K}(x)) + O(1).$$

Proof. The first inequality is trivial. To prove the second inequality, let p be a program of length $\mathbf{K}(x)$ for printing x on some machine T. Let $n = |p|$, let $u_1 \cdots u_k$ be the form of the number n in the base m number system. Let $u = u_1 0 u_2 0 \cdots u_k 011$. Then the prefix u of the word up can be uniquely reconstructed, and from it, the length of the word can be determined without having to go beyond its end. Using this, it is easy to write a self-delimiting program of length $2k + n + O(1)$ that prints x. □

From the foregoing, it may seem that the function $\mathbf{H}$ is a slight technical variant of the Kolmogorov complexity. The next lemma shows a significant difference between them.

Lemma 6.2.2.

(a) $\sum_x m^{-\mathbf{K}(x)} = +\infty$.

(b) $\sum_x m^{-\mathbf{H}(x)} \le 1$.

Proof. Statement (a) follows easily from Lemma 6.1.1. In order to prove (b), consider an optimal code $f(x)$ for each word x. Due to the self-delimiting property, neither of these can be a prefix of another one; thus, (b) follows immediately from the simple but important information-theoretical lemma below. □

Lemma 6.2.3. *Let $\mathcal{L} \subseteq \Sigma_0^*$ be a language such that neither of its words is a prefix of another one. Let $m = |\Sigma_0|$. Then*

$$\sum_{y \in \mathcal{L}} m^{-|y|} \le 1.$$

Proof. Choose letters $a_1, a_2, \ldots$ independently, with uniform distribution from the alphabet Σ_0; stop if the obtained word is in $\mathcal{L}$. The probability that we obtained a word $y \in \mathcal{L}$ is exactly $m^{-|y|}$ (since according to the assumption, we did not stop on any prefix of y). Since these events are mutually exclusive, the statement of the lemma follows. □

Some interesting consequences of these lemmas are formulated in the following exercises.

Exercise 6.2.1. *Show that the following strengthening of Lemma 6.2.1 is not true:*

$$\mathbf{H}(x) \le \mathbf{K}(x) + \log_m \mathbf{K}(x) + O(1).$$

Exercise 6.2.2. *The function $\mathbf{H}(x)$ is not recursive.*

The next theorem shows that the function $\mathbf{H}(x)$ can be approximated well.

Theorem 6.2.4 (Levin's Coding Theorem)**.** *Let p be a computable probability distribution on Σ_0^*. Then for every word x we have*

$$\mathbf{H}(x) \le -\log_m p(x) + O(1).$$

Proof. Let us call *m-ary rational* those rational numbers that can be written with a numerator that is a power of m. The m-ary rational numbers of the interval $[0, 1)$ can be written in the form $0.a_1 \ldots a_k$ where $0 \le a_i \le m - 1$.

Subdivide the interval $[0, 1)$ into left-closed, right-open intervals $J(x_1), J(x_2), \ldots$ of lengths $p(x_1), p(x_2), \ldots$ respectively (where $x_1, x_2, \ldots$ is the increasing ordering of Σ_0^*). For every $x \in \Sigma_0^*$ with $p(x) > 0$, there will be an m-ary rational number $0.a_1 \ldots a_k$ with $0.a_1 \ldots a_k \in J(x)$ and $0.a_1 \ldots a_{k-1} \in J(x)$. We will call a shortest sequence $a_1 \ldots a_k$ with this property the *Shannon-Fano code* of x.

We claim that every word x can be computed easily from its Shannon-Fano code. Indeed, for the given sequence $a_1, \ldots, a_k$, for values $i = 1, 2, \ldots$, we check consecutively whether $0.a_1 \ldots a_k$ and $0.a_1 \ldots a_{k-1}$ belong to the same interval $J(x)$; if yes, we print x and stop. Notice that this program is self-delimiting: we need not know in advance the length of the code, and if $a_1 \ldots a_k$ is the Shannon-Fano code of a word x then we will never read beyond the end of the sequence $a_1 \ldots a_k$. Thus $\mathbf{H}(x)$ is not greater than the common length of the (constant-length) program of the above algorithm and the Shannon-Fano code of x; about this, it is easy to see that it is at most $\log_m p(x) + 1$. □

This theorem implies that the expected value of the difference between $\mathbf{H}(x)$ and $-\log_m p(x)$ is bounded (compare with Theorem 6.1.4).

Corollary 6.2.5. *With the conditions of Theorem 6.2.4*

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)| = O(1).$$

Proof.

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|$$
$$= \sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_+ + \sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_-.$$

Here, the first sum can be estimated, according to Theorem 6.2.4, as follows:

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_+ \leq \sum_x p(x)O(1) = O(1).$$

We estimate the second sum as follows:

$$|\mathbf{H}(x) + \log_m p(x)|_- \leq m^{-\mathbf{H}(x)-\log_m p(x)} = \frac{1}{p(x)} m^{-\mathbf{H}(x)},$$

and hence, according to Lemma 6.2.2,

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_- \leq \sum_x m^{-\mathbf{H}(x)} \leq 1.$$

□

Remark. The coding theorem can be further generalized, for this we introduce the following definitions.

Definition 6.2.1. *Let $f(x)$ be a function on strings, taking real number values. We say that $f(x)$ is* semicomputable *if there is computable function $g(x, n)$ taking rational values such that $g(x, n)$ is monotonically increasing in n and $\lim_{n\to\infty} g(x, n) = f(x)$.*

We say that $p(x)$ is a semimeasure *if $p(x) \geq 0$ and $\sum_x p(x) \leq 1$.*

Levin proved the coding theorem for the more general case when $p(x)$ is a semicomputable semimeasure. Lemma 6.2.2 shows that $m^{-\mathbf{H}(x)}$ is a semicomputable semimeasure. Therefore, Levin's theorem implies that $m^{-\mathbf{H}(x)}$ is maximal, to within a multiplicative constant, among all semicomputable semimeasures. This is a technically very useful characterization of $\mathbf{H}(x)$.

Let us show that the complexity $\mathbf{H}(x)$ can be well approximated for "almost all" strings, where the meaning of "almost all" is given by some probability distribution. But we will not consider arbitrary probability distributions, only such that can be approximated, at least from below, by a computable sequence.

Proof for the semicomputable case:

Proof. Here we prove only the case $m = 2$. If $p(x)$ is semicomputable then the set

$$\{\, (x, k) : k2^{-k} < p(x) \,\}$$

is recursively enumerable. Let $\{\, (z_t, k_t) : t = 1, 2, \dots \}$ be a recursive enumeration of this set without repetition. Then

$$\sum_t 2^{-k_t} = \sum_x \sum \{\, 2^{-k_t} : z_t = x \,\} \leq \sum_x 2p(x) < 2.$$

Let us cut off consecutive adjacent, disjoint intervals $I_1, I_2, \dots$, where I_t has length 2^{-k_t-1}, from the left side of the interval $[0, 1]$. For any binary string w consider the interval J_w delimited by the binary "decimals" $0.w$ and $0.w1$. We define a function $F(w)$ as follows. If J_w is the largest such binary subinterval of some I_t then $F(w) = z_t$. Otherwise $F(w)$ is undefined. It follows from the construction that for every x there

is a t with $z_t = x$ and $0.5p(x) < 2^{-k_t}$. Therefore, for every x there is a w such that $F(w) = x$ and $|w| \leq -\log p(x) + 4$.

It is easy to see that a program q can be written for our universal self-delimiting Turing machine such that for every w, the string qw is a self-delimiting program for $F(w)$. (For this, it is enough to see that if $F(v)$ and $F(w)$ are both defined then v is not a prefix of w.) □

6.3 The notion of a random sequence

In this section, we assume that $\Sigma_0 = \{0, 1\}$, i.e., we will consider only the complexity of 0-1 sequences. Roughly speaking, we want to consider a sequence random if there is no "regularity" in it. Here, we want to be as general as possible and consider any kind of regularity that would enable a more economical coding of the sequence (so that the complexity of the sequence would be small).

Remark. Note that this is not the only possible idea of regularity. One might consider a 0-1-sequence regular if the number of 0's in it is about the same as the number of 1's. This kind of regularity is compatible with (in fact implied by) randomness: we should really consider only regularities that are shared only by a small minority of the sequences.

Let us bound first the complexity of "average" 0-1 sequences.

Lemma 6.3.1. *The number of 0-1 sequences x of length n with $\mathbf{K}(x) \leq n - k$ is less than 2^{n-k+1}.*

Proof. The number of "codewords" of length at most $n-k$ is at most $1+2+\cdots+2^{n-k} < 2^{n-k+1}$, hence only fewer than 2^{n-k+1} strings x can have such a code. □

Corollary 6.3.2. *The complexity of 99% of the n-digit 0-1 sequences is at least $n-8$. If we choose a 0-1 sequence of length n randomly then $|\mathbf{K}(x)-n| \leq 100$ with probability $1 - 2^{100}$.*

Another corollary of this simple lemma is that it shows, in a certain sense, a "counterexample" to Church's Thesis, as we noted in the at the beginning of Chapter 5. Consider the following problem: For a given n, construct a 0-1 sequence of length n whose Kolmogorov complexity is greater than $n/2$. According to Exercise 6.1.2, this problem is algorithmically undecidable. On the other hand, the above lemma shows that with large probability, a randomly chosen sequence is appropriate.

According to Theorem 6.1.3, it is algorithmically impossible to find the best code. There are, however, some easily recognizable properties indicating about a word that it is codable more efficiently than its length. The next lemma shows such a property:

Lemma 6.3.3. *If the number of 1's in a 0-1 sequence x of length n is k then*

$$\mathbf{K}(x) \leq \log \binom{n}{k} + O(\log n).$$

Let $k = pn$ $(0 < p < 1)$, then this can be estimated as

$$\mathbf{K}(x) \leq (-p \log p - (1-p)\log(1-p))n + O(\log n).$$

In particular, if $k > (1/2 + \varepsilon)n$ or $k < (1/2 - \varepsilon)n$ then

$$\mathbf{K}(x) \leq cn + O(\log n)$$

where $c = -(1/2 + \varepsilon) \cdot \log(1/2 + \varepsilon) - (1/2 - \varepsilon) \cdot \log(1/2 - \varepsilon)$ is a positive constant smaller than 1 and depending only on ε.

Proof. x can be described as the "lexicographically t-th sequence among all sequences of length n containing exactly k 1's". Since the number of sequences of length n containing k 1's is $\binom{n}{k}$, the description of the numbers t, n and k needs only $\log \binom{n}{k} + 2\log n + 2\log k$ bits. (Here, the factor 2 is due to the need to separate the three pieces of information from each other; we leave it to the reader to find the trick.) The program choosing the appropriate sequence needs only a constant number of bits.

The estimate of the binomial coefficient is done by a method familiar from probability theory. □

On the basis of the above, one can consider either $|x| - \mathbf{K}(x)$ or $|x|/\mathbf{K}(x)$ as a measure of the randomness (or, rather, non-randomness) of the word x. The larger are these numbers, the smaller is $\mathbf{K}(x)$ relative to $|x|$, which means that x has more "regularity" and so it is less random.

In case of infinite sequences, a sharper difference can be made: we can define whether a given sequence is random. Several definitions are possible; we introduce here the simplest version. Let x be an infinite 0-1 sequence, and let x_n denote its starting segment formed by the first n elements. We call the sequence x *informatically (weakly) random* if $\mathbf{K}(x_n)/n \to 1$ when $n \to \infty$.

It can be shown that every informatically weakly random sequence satisfies the laws of large numbers. We consider here only the simplest such result. Let a_n denote the number of 1's in the string x_n, then the previous lemma immediately implies the following theorem:

Theorem 6.3.4. *If x is informatically random then $a_n/n \to 1/2$ $(n \to \infty)$.*

The question arises whether the definition of an informatically random sequence is not too strict, whether there are any informatically random infinite sequences at all. Let us show that not only there are such sequences but that almost all sequences have this property:

Theorem 6.3.5. *Let the elements of an infinite 0-1 sequence x be 0's or 1's, independently from each other, with probability $1/2$. Then x is informatically random with probability 1.*

Proof. For a fixed $\varepsilon > 0$, let S_n be the set of all those length n sequences y for which $\mathbf{K}(y) < (1 - \varepsilon)n$, and let A_n denote the event $x_n \in S_n$. Then by Lemma 6.3.1,

$$\mathsf{P}(A_n) \leq \sum_{y \in S_n} 2^{-n} < 2^{(1-\varepsilon)n+1} \cdot 2^{-n} = 2^{1-\varepsilon n},$$

and hence the sum $\sum_{n=1}^{\infty} \mathsf{P}(A_n)$ is convergent. But then, the Borel-Cantelli Lemma from probability theory implies that with probability 1, only finitely many of the events A_n occur, which means that $\mathbf{K}(x_n)/n \to \infty$. □

Remark. If the members of the sequence x are generated by an algorithm, then x_n can be computed from the program of the algorithm (constant length) and from the number n (can be given in $\log n$ bits). Therefore, for such a sequence $\mathbf{K}(x_n)$ grows very slowly.

6.4 Kolmogorov complexity, entropy and coding

Let $p = (p_1, p_2, \ldots)$ be a *discrete probability distribution*, i.e., a non-negative (finite or infinite) sequence with $\sum_i p_i = 1$. Its *entropy* is the quantity

$$H(p) = \sum_i -p_i \log p_i$$

(the term $p_i \log p_i$ is considered to be 0 if $p_i = 0$). Notice that in this sum, all terms are nonnegative, so $H(p) \geq 0$; equality holds if and only if the value of some p_i is 1 and the value of the rest is 0. It is easy to see that for fixed alphabet size m, the probability distribution with maximum entropy is $(1/m, \ldots, 1/m)$ and the entropy of this is $\log m$.

Entropy is a basic notion of information theory and we do not treat it in detail in these notes, we only point out its connection with Kolmogorov complexity. We have met with entropy for the case $m = 2$ in Lemma 6.3.3. This lemma is easy to generalize to arbitrary alphabets as

Lemma 6.4.1. *Let $x \in \Sigma_0^*$ with $|x| = n$ and let p_h denote the relative frequency of the letter h in the word x. Let $p = (p_h : h \in \Sigma_0)$. Then*

$$\mathbf{K}(x) \leq \frac{H(p)}{\log m} n + O(\frac{m \log n}{\log m}).$$

Proof. Let us give a different proof using Theorem 6.2.4. Consider the probability distribution over the strings of length n in which each symbol h is chosen with probability p_h. The probabilities p_h are fractions with denominator n, hence their description needs at most $O(m \log n)$ bits, what is $O(\frac{m \log n}{\log m})$ symbols of our alphabet. The distribution over the strings is therefore an enumerable probability distribution P whose program has length $O(m \log n)$. According to Theorem 6.2.4, we have

$$\mathbf{K}(x) \leq \mathbf{H}(x) \leq -\log_m P(x) + O(m \log n).$$

But $-\log_m P(x)$ is exactly $\frac{nH(p)}{\log m}$. □

Remark. We mention another interesting connection between entropy and complexity: the entropy of a computable probability distribution over all strings is close to the average complexity. This reformulation of Corollary 6.2.5 can be stated as

$$\left| H(p) - \sum_x p(x)\mathbf{H}(x) \right| = O(1),$$

for any computable probability distribution p over the set Σ_0^*.

Let $\mathcal{L} \subseteq \Sigma_0^*$ be a recursive language and suppose that we want to find a short program, "code", only for the words in $\mathcal{L}$. For each word x in $\mathcal{L}$, we are thus looking for a program $f(x) \in \{0,1\}^*$ printing it. We call the function $f : \mathcal{L} \to \Sigma^*$ a *Kolmogorov code* of $\mathcal{L}$. The *conciseness* of the code is the function

$$\eta(n) = \max\{\, |f(x)| : x \in \mathcal{L},\ |x| \leq n \,\}.$$

We can easily get a lower bound on the conciseness of any Kolmogorov code of any language. Let $\mathcal{L}_n$ denote the set of words of $\mathcal{L}$ of length at most n. Then obviously,

$$\eta(n) \geq \log |\mathcal{L}_n|.$$

We call this estimate the *information theoretical lower bound.*

This lower bound is sharp (to within an additive constant). We can code every word x in $\mathcal{L}$ simply by telling its serial number in the increasing ordering. If the word x of length n is the t-th element then this requires $\log t \leq \log |\mathcal{L}_n|$ bits, plus a constant number of additional bits (the program for taking the elements of Σ^* in lexicographic order, checking their membership in $\mathcal{L}$ and printing the t-th one).

We arrive at more interesting questions if we stipulate that the code from the word and, conversely, the word from the code should be polynomially computable. In other words: we are looking for a language $\mathcal{L}'$ and two polynomially computable functions:

$$f : \mathcal{L} \to \mathcal{L}', \qquad g : \mathcal{L}' \to \mathcal{L}$$

with $g \circ f = \mathrm{id}_{\mathcal{L}}$ for which, for every x in $\mathcal{L}$ the code $|f(x)|$ is "short" compared to $|x|$. Such a pair of functions is called a *polynomial time code.* (Instead of the polynomial time bound we could, of course, consider other complexity restrictions.)

We present some examples when a polynomial time code approaches the information-theoretical bound.

Example 6.4.1. In the proof of Lemma 6.3.3, for the coding of the 0-1 sequences of length n with exactly m 1's, we used the simple coding in which the code of a sequence is the number giving its place in the lexicographic ordering. We will show that this coding is polynomial.

Let us view each 0-1 sequence as the obvious code of a subset of the n-element set $\{n-1, n-2, \ldots, 0\}$. Each such set can be written as $\{a_1, \ldots, a_m\}$ with $a_1 > a_2 > \cdots > a_m$. Then the set $\{b_1, \ldots, b_m\}$ precedes the set $\{a_1, \ldots, a_m\}$ lexicographically if and only if there is an i such that $b_i < a_i$ while $a_j = b_j$ holds for all $j < i$. Let $\{a_1, \ldots, a_m\}$ be the lexicographically t-th set. Then the number of subsets $\{b_1, \ldots, b_m\}$ with the above property for a given i is exactly $\binom{a_i}{m-i+1}$. Summing this for all i we find that

$$t = 1 + \binom{a_1}{m} + \binom{a_2}{m-1} + \cdots + \binom{a_m}{1}. \tag{6.3}$$

So, given $a_1, \ldots, a_m$, the value of t is easily computable in time polynomial in n. Conversely, if $t < \binom{n}{m}$ is given then t is easy to write in the above form: first we find, using binary search, the greatest natural number a_1 with $\binom{a_1}{m} \leq t - 1$, then the

greatest number a_2 with $\binom{a_2}{m-1} \leq t-1-\binom{a_1}{m}$, etc. We do this for m steps. The numbers obtained this way satisfy $a_1 > a_2 > \cdots$; indeed, according to the definition of a_1 we have $\binom{a_1+1}{m} = \binom{a_1}{m} + \binom{a_1}{m-1} > t-1$ and therefore $\binom{a_1}{m-1} > t-1-\binom{a_1}{m}$ implying $a_1 > a_2$. It follows similarly that $a_2 > a_3 > \cdots > a_m \geq 0$ and that there is no "remainder" after m steps, i.e., that 6.3 holds. It can therefore be determined in polynomial time which subset is lexicographically the t-th.

Example 6.4.2. Consider trees, given by their adjacency matrices (but any other "reasonable" representation would also do). In such representations, the vertices of the tree have a given order, which we can also express saying that the vertices of the tree are labeled by numbers from 0 to $(n-1)$. We consider two trees equal if whenever the nodes i, j are connected in the first one they are also connected in the second one and vice versa (so, if we renumber the nodes of the tree then we may arrive at a different tree). Such trees are called *labeled trees*. Let us first see what does the information-theoretical lower bound give us, i.e., how many trees are there. The following classical result, called Cayley's Theorem, applies here:

Theorem 6.4.2 (Cayley's Theorem)**.** *The number of n-node labeled trees is n^{n-2}.*

Consequently, by the information-theoretical lower bound, for any encoding of trees some n-node tree needs a code with length at least $\lceil \log(n^{n-2}) \rceil = \lceil (n-2)\log n \rceil$. But can this lower bound be achieved by a polynomial time computable code?

(a) Coding trees by their adjacency matrices takes n^2 bits. (It is easy to see that $\binom{n}{2}$ bits are enough).

(b) We fare better if we specify each tree by enumerating its edges. Then we must give a "name" to each vertex; since there are n vertices we can give to each one a 0-1 sequence of length $\lceil \log n \rceil$ as its name. We specify each edge by its two endnodes. In this way, the enumeration of the edges takes cca. $2(n-1)\log n$ bits.

(c) We can save a factor of 2 in (b) if we distinguish a root in the tree, say the node 0, and we specify the tree by the sequence $(\alpha(1), \ldots, \alpha(n-1))$ in which $\alpha(i)$ is the first interior node on the path from node i to the root (the "father" of i). This is $(n-1)\lceil \log n \rceil$ bits, which is already nearly optimal.

(d) There is, however, a procedure, the so-called *Prüfer code*, that sets up a bijection between the n-node labeled trees and the sequences of length $n-2$ of the numbers $0, \ldots, n-1$. (Thereby it also proves Cayley's theorem). Each such sequence can be considered the expression of a natural number in the base n number system; in this way, we order a "serial number" between 0 and $n^{n-2}-1$ to the n-node labeled trees. Expressing these serial numbers in the base two number system, we get a coding in which the code of each number has length at most $\lceil (n-2)\log n \rceil$.

The Prüfer code can be considered as a refinement of procedure (c). The idea is that we order the edges $[i, \alpha(i)]$ not by the value of i but a little differently. Let us define the permutation $(i_1, \ldots, i_n)$ as follows: let i_1 be the smallest endnode (leaf) of the tree; if $i_1, \ldots, i_k$ are already defined then let i_{k+1} be the smallest endnode of the graph remaining after deleting the nodes $i_1, \ldots, i_k$. (We do not consider the root 0

an endnode.) Let $i_n = 0$. With the i_k's thus defined, let us consider the sequence $(\alpha(i_1), \ldots, \alpha(i_{n-1}))$. The last element of this is 0 (the "father" of the node i_{n-1} can namely be only i_n), it is therefore not interesting. We call the remaining sequence $(\alpha(i_1), \ldots, \alpha(i_{n-2}))$ the *Prüfer code* of the tree.

Claim 6.4.3. *The Prüfer code of a tree determines the tree.*

For this, it is enough to see that the Prüfer code determines the sequence $i_1, \ldots, i_n$; then we know all the edges of the tree (the pairs $[i, \alpha(i)]$).

The node i_1 is the smallest endnode of the tree; hence to determine i_1, it is enough to figure out the endnodes from the Prüfer code. But this is obvious: the endnodes are exactly those that are not the "fathers" of other nodes, i.e., the ones that do not occur among the numbers $\alpha(i_1), \ldots, \alpha(i_{n-2}), 0$. The node i_1 is therefore uniquely determined.

Assume that we know already that the Prüfer code uniquely determines $i_1, \ldots, i_{k-1}$. It follows similarly to the above that i_k is the smallest number not occurring neither among $i_1, \ldots, i_{k-1}$ nor among $\alpha(i_k), \ldots, \alpha(i_{n-2}), 0$. So, i_k is also uniquely determined.

Claim 6.4.4. *Every sequence $(b_1, \ldots, b_{n-2})$, where $0 \le b_i \le n-1$, occurs as the Prüfer code of some tree.*

Using the idea of the proof above, let $b_{n-1} = 0$ and let us define the permutation $i_1, \ldots, i_n$ by the recursion that i_k is the smallest number not occurring neither among $i_1, \ldots, i_{k-1}$ nor among $b_k, \ldots, b_{n-1}$, where $(1 \le k \le n-1)$; and let $i_n = 0$. Connect i_k with b_k for all $1 \le k \le n-1$ and let $\gamma(i_k) = b_k$. In this way, we obtain a graph G with $n-1$ edges on the nodes $0, \ldots, n-1$. This graph is connected, since for every i the $\gamma(i)$ comes later in the sequence $i_1, \ldots, i_n$ than i and therefore the sequence $i, \gamma(i), \gamma(\gamma(i)), \ldots$ is a path connecting i to the node 0. But then G is a connected graph with $n-1$ edges, therefore it is a tree. That the sequence $(b_1, \ldots, b_{n-2})$ is the Prüfer code of G is obvious from the construction.

Remark. An exact correspondence like the Prüfer code has other advantages besides optimal Kolmogorov coding. Suppose that our task is to write a program for a randomized Turing machine that outputs a random labeled tree of size n in such a way that all trees occur with the same probability. The Prüfer code gives an efficient algorithm for this. We just have to generate randomly a sequence $b_1, \ldots, b_{n-2}$, which is easy, and then decode from it the tree by the above algorithm.

Example 6.4.3. Consider now the unlabeled trees. These can be defined as the equivalence classes of labeled trees where two labeled trees are considered equivalent if they are *isomorphic*, i.e., by a suitable relabeling, they become the same labeled tree. We assume that we represent each equivalence class by one of its elements, i.e., by a labeled tree (it is not interesting now, by which one). Since each labeled tree can be labeled in at most $n!$ ways (its labelings are not necessarily all different as labeled trees!) therefore the number of unlabeled trees is at least $n^{n-2}/n! > 2^{n-2}$ (if $n \ge 25$). The information-theoretical lower bound is therefore at least $n-2$. (According to a difficult result of George Pólya, the number of n-node unlabeled trees is asymptotically $c_1 c_2^n n^{3/2}$ where c_1 and c_2 are constants defined in a certain complicated way.)

On the other hand, we can use the following coding procedure. Consider an n-node tree F. Walk through F by the "depth-first search" rule: Let x_0 be the node labeled 0 and define the nodes $x_1, x_2, \ldots$ as follows: if x_i has a neighbor that does not occur yet

in the sequence then let x_{i+1} be the smallest one among these. If it does not have such a neighbor and $x_i \neq x_0$ then let x_{i+1} be the neighbor of x_i on the path leading from x_i to x_0. Finally, if $x_i = x_0$ and every neighbor of x_0 occured already in the sequence then we stop.

It is easy to see that for the sequence thus defined, every edge occurs among the pairs $[x_i, x_{i+1}]$, moreover, it occurs once in both directions. It follows that the length of the sequence is exactly $2n-1$. Let now $\varepsilon_i = 1$ if x_{i+1} is farther from the root than x_i and $\varepsilon_i = 0$ otherwise. It is easy to understand that the sequence $\varepsilon_0\varepsilon_1\cdots\varepsilon_{2n-3}$ determines the tree uniquely; passing trough the sequence, we can draw the graph and construct the sequence $x_1, \ldots, x_i$ of nodes step-for-step. In step $(i+1)$, if $\varepsilon_i = 1$ then we take a new node (this will be x_{i+1}) and connect it with x_i; if $\varepsilon_i = 0$ then let x_{i+1} be the neighbor of x_i in the "direction" of x_0.

Remarks. 1. With this coding, the code assigned to a tree depends on the labeling but it does not determine it uniquely (it only determines the unlabeled tree uniquely).

2. The coding is not bijective: not every 0-1 sequence will be the code of an unlabeled tree. We can notice that

(a) There are as many 1's as 0's in each tree;

(b) In every starting segment of every code, there are at least as many 1's as 0's

(the difference between the number of 1's and the number of 0's among the first i numbers gives the distance of the node x_i from the node 0). It is easy to see that for each 0-1 sequence having the properties $(a)-(b)$, there is a labeled tree whose code it is. It is not sure, however, that this tree, as an unlabeled tree, is given with just this labeling (this depends on which unlabeled trees are represented by which of their labelings). Therefore, the code does not even use all the words with properties $(a)-(b)$.

3. The number of 0-1 sequences having properties $(a)-(b)$ is, according to a well-known combinatorial theorem, $\frac{1}{n}\binom{2n-2}{n-1}$ (the so-called *catalan number*). We can formulate a tree notion to which the sequences with properties $(a)-(b)$ correspond exactly: these are the *rooted planar trees*, which are drawn without intersection into the plane in such a way that their distinguished vertex—their root—is on the left edge of the page. This drawing defines an ordering among the "sons" (neighbors farther from the root) "from the top to the bottom"; the drawing is characterized by these orderings. The above described coding can also be done in rooted planar trees and creates a bijection between them and the sequences with the properties $(a)-(b)$.

Exercise 6.4.1. *(a) Let x be a 0-1 sequence that does not contain 3 consecutive 0's. Show that $\mathbf{K}(x) < .99|x| + O(1)$.*

(b) Find the best constant in place of .99. [Hint: you have to find approximately the number of such sequences. Let $A(n)$ and $B(n)$ be the number of such sequences ending with 0 and 1, respectively. Find recurrence relations for A and B.]

(c) Give a polynomial time coding-decoding procedure for such sequence that compresses each of them by at least 1 percent.

Exercise 6.4.2. *(a) Prove that for any two strings $x, y \in \Sigma_0^*$,*

$$\mathbf{K}(xy) \leq 2\mathbf{K}(x) + \mathbf{K}(y) + c,$$

where c depends only on the universal Turing machine in the definition of infromation complexity.

(b) Prove that the stronger and more natural looking inequality

$$\mathbf{K}(xy) \leq \mathbf{K}(x) + \mathbf{K}(y) + c$$

is false.

Exercise 6.4.3. *Suppose that the universal Turing machine used in the definition of $\mathbf{K}(x)$ uses programs written in a two-letter alphabet and outputs strings in an s-letter alphabet.*

(a) *Prove that $\mathbf{K}(x) \leq |x| \log s + O(1)$.*

(b) *Prove that, moreover, there are polynomial time functions f, g mapping strings x of length n to binary strings of length $n \log s + O(1)$ and vice versa with $g(f(x)) = x$.*

Exercise 6.4.4.

(a) *Give an upper bound on the Kolmogorov complexity of Boolean functions of n variables.*

(b) *Give a lower bound on the complexity of the most complex Boolean function of n variables.*

(c) *Use the above result to find a number $L(n)$ such that there is a Boolean function with n variables which needs a Boolean circuit of size at least $L(n)$ to compute it.*

Exercise 6.4.5. *Call an infinite 0-1 sequence x* (informatically) strongly random *if $n - \mathbf{H}(x_n)$ is bounded from above. Prove that every informatically strongly random sequence is also weakly random.*

Exercise 6.4.6. *Prove that almost all infinite 0-1 sequences are strongly random.*

Chapter 7

Pseudorandom numbers

We have seen that various important algorithms use random numbers (or, equivalently, independent random bits). But how do we get such bits?

One possible source is from outside the computer. We could obtain "real" random sequences, say, from radioactive decay. In most cases, however, this would not work: our computers are very fast and we have no physical device giving the equivalent of unbiased coin-tosses at this rate.

Thus we have to resort to generating our random bits by the computer. However, a long sequence generated by a short program is never random, according to the notion of randomness introduced in Chapter 6 using information complexity. Thus we are forced to use algorithms that generate random-looking sequences; but, as Von Neumann (one of the first mathematicians to propose the use of these) put it, everybody using them is inevitably "in the state of sin". In this chapter, we will understand the kind of protection we can get against the graver consequences of this sin.

There are other reasons besides practical ones to study pseudorandom number generators. We often want to repeat some computation for various reasons, including error checking. In this case, if our source of random numbers was really random, then the only way to use the same random numbers again is to store them, using a lot of space. With pseudorandom numbers, this is not the case: we only have to store the "seed", which is much shorter. Another, and more important, reason is that there are applications where what we want is only that the sequence should "look random" to somebody who does not know how it was generated. The collection of these applications called cryptography is to be treated in Chapter 12.

The way a *pseudorandom bit generator* works is that it turns a short random string called the "seed" into a longer pseudorandom string. We require that it works in polynomial time. The resulting string has to "look" random: and the important fact is that this can be defined exactly. Roughly speaking, there should be no polynomial time algorithm that distinguishes it from a truly random sequence. Another feature, often easier to verify, is that no algorithm can predict any of its bits from the previous bits. We prove the equivalence of these two conditions.

But how do we design such a generator? Various ad hoc methods that produce random-looking sequences (like taking the bits in the binary representation of a root of a given equation) turn out to produce strings that do not pass the strict criteria we impose. A general method to obtain such sequences is based on *one-way functions*:

functions that are easy to evaluate but difficult to invert. While the existence of such functions is not proved (it would imply that P is different from NP), there are several candidates, that are secure at least against current techniques.

7.1 Classical methods

There are several classical methods that generate a "random-looking" sequence of bits. None of these meets the strict standards to be formulated in the next section; but due to their simplicity and efficiency, they (especially linear congruential generators, example 7.1.2 below) can be used well in practice. There is a large amount of practical information about the best choice of the parameters; we don't go into this here, but refer to Volume 2 of Knuth's book.

Example 7.1.1. *Shift registers* are defined as follows. Let $f : \{0,1\}^n \to \{0,1\}$ be a function that is easy to compute. Starting with a *seed* of n bits $a_0, a_1, \ldots, a_{n-1}$, we compute bits $a_n, a_{n+1}, a_{n+2}, \ldots$ recursively, by

$$a_k = f(a_{k-1}, a_{k-2}, \ldots, a_{k-n}).$$

The name shift register comes from the fact that we only need to store $n+1$ bits: after storing $f(a_0, \ldots, a_{n-1})$ in a_n, we don't need a_0 any more, and we can shift a_1 to a_0, a_2 to a_1, etc. The most important special case is when f is a linear function over the 2-element field, and we'll restrict ourselves to this case.

Looking at particular instances, the bits generated by a linear shift register look random, at least for a while. Of course, the sequence $a_0.a_1, \ldots,$ will eventually have some n-tuple repeated, and then it will be periodic from then on; but this need not happen sooner than a_{2^n}, and indeed one can select the (linear) function f so that the period of the sequence is as large as 2^n.

The problem is that the sequence has more hidden structure than just periodicity. Indeed, let

$$f(x_0, \ldots, x_{n-1}) = b_0x_0 + b_1x_1 + \ldots b_{n-1}x_{n-1}$$

(where $b_i \in \{0,1\}$). Assume that we do not know the coefficients $b_0, \ldots, b_{n-1}$, but observe the first n bits $a_n, \ldots, a_{2n-1}$ of the output sequence. Then we have the following system of linear equations to determine the b_i:

$$\begin{aligned} b_0a_0 + b_1a_1 + \ldots b_{n-1}a_{n-1} &= a_n \\ b_0a_1 + b_1a_2 + \ldots b_{n-1}a_n &= a_{n+1} \\ &\vdots \\ b_0a_{n-1} + b_1a_n + \ldots b_{n-1}a_{2n-2} &= a_{2n-1} \end{aligned}$$

Here are n equations to determine the n unknowns (the equations are over the 2-element field). Once we have the b_i, we can predict all the remaining elements of the sequence $a_{2n}, a_{2n+1}, \ldots$

It may happen, of course, that this system is not uniquely solvable, because the equations are dependent. For example, we might start with the seed $00\ldots0$, in which

case the equations are meaningless. But it can be shown that for a random choice of the seed, the equations determine the coefficients b_i with positive probability. So after seeing the first $2n$ elements of the sequence, the rest "does not look random" for an observer who is willing to perform a relatively simple (polynomial time) computation.

Example 7.1.2. The most important pseudorandom number generators in practice are *linear congruential generators*. Such a generator is given by three parameters a, b and m, which are positive integers. Starting with a seed X_0, which is an integer in the range $0 \leq X_0 \leq m-1$, the generator recursively computes integers $X_1, X_2, \ldots$ by

$$X_i = aX_{i-1} + b \pmod{m}.$$

One might use all the X_i or extract, say, the middle bit of each, and output this sequence.

It turns out that the output of these generators can also be predicted by a polynomial time computation, after observing a polynomial number of output bits. The algorithms to do so are much more involved, however, and due to their fastness and simplicity, linear congruential generators can be used for most practical applications.

Example 7.1.3. As a third example, let us look at the binary expansion of, say, $\sqrt{5}$:

$$\sqrt{5} = 10.001111000110111\ldots$$

This sequence looks rather random. Of course, we cannot use the same number all the time; but we can pick, say, an n-bit integer a as our "seed", and output the bits of $\sqrt{a} - \lfloor\sqrt{a}\rfloor$. Unfortunately, this method turns out to be "breakable" by rather advanced (but polynomial time) methods from algorithmic number theory.

7.2 The notion of a pseudorandom number generator

In general, a pseudorandom bit generator transforms a short, truly random sequence s (the "seed") into a longer sequence $g(s)$ that still "looks" random. The success of using $g(s)$ in place of a random sequence depends on how severely the randomness of $g(s)$ is tested by the application. If the application has the ability to test all possible seeds that might have generated $g(s)$ then it finds the true seed and not much randomness remains. For this, however, the application may have to run too long. We would like to call g a pseudorandom bit generator if no applications running only in polynomial time can distinguish $g(s)$ from truly random strings.

To make the definition precise, we need some preparation. We say that a function $f : \mathbb{Z}_+ \to \mathbb{R}$ is *negligible*, if $n^k f(n) \to 0$ as $n \to \infty$ for each fixed k. In other words, f tends to 0 faster than the reciprocal of any polynomial. It will be convenient to denote this (analogously to the "big-O" notation), by

$$f(n) = \mathrm{NEGL}(n).$$

Note that a polynomial multiple of a negligible function is still negligible; thus

$$n^r \mathrm{NEGL}(n) = \mathrm{NEGL}(n)$$

for each fixed r.

Consider a polynomial time computable function $G : \{0,1\}^* \to \{0,1\}^*$, where we assume that the length $|G(x)|$ depends only on the length $|x|$ of x, and $|x| < |G(x)| < c|x|^c$ for some constant c. (So G stretches string but not too much.) We call such a function a *generator*. Let $\mathcal{A}$ be a randomized polynomial time algorithm (Turing machine) that accepts any 0-1 string x as input and computes a bit $\mathcal{A}(x)$ from it. (We will interpret $\mathcal{A}(x) = 0$ as "not random", and $\mathcal{A}(x) = 1$ as "random".) Fix an $n \geq 1$. Let x be chosen uniformly from $\{0,1\}^n$ and let y be chosen uniformly from $\{0,1\}^N$, where $N = |G(x)|$. We flip a coin, and depending on its result, we either feed $G(x)$ or y to $\mathcal{A}$. We say that $\mathcal{A}$ is *successful* if either $G(x)$ was fed to $\mathcal{A}$ and it output 0 or y was fed and the output is 1.

The generator G is called a *(safe) random number generator* if for every randomized polynomial time algorithm $\mathcal{A}$ that takes a 0-1 string x as input and computes a bit $\mathcal{A}(x)$ from it, the probability that $\mathcal{A}$ is successful is at most $1/2 + \mathrm{NEGL}(n)$.

This definition says that $G(x)$ passes every "reasonable" test (any test computable in randomized polynomial time) in the sense that the probability that such a test recognizes that $G(x)$ is not truly random is only negligibly larger than $1/2$ (which can of course be achieved by guessing randomly). The probability in (b) is taken over the random choice of x and y, over the coin flip determining which one is fed to $\mathcal{A}$, and over the internal coin flips of $\mathcal{A}$.

This requirement is so strong that it is unknown whether safe random number generators exist at all (if they do, then P $\neq$ NP; see the Exercise 7.4.3). But we will see in the next section that they exist under some complexity-theoretic assumptions.

Our definition of a safe random number generator is very general and the condition is difficult to verify. The following theorem of Yao provides a way to establish that a function is a safe random number generator that is often more convenient. What it says is that every bit of $G(x)$ is highly unpredictable from the previous bits, as long as the prediction algorithm does not use too much time.

We say that a generator g is *unpredictable* if the following holds. Let $n \geq 1$ and let x be a random string chosen uniformly from $\{0,1\}^n$. Let $g(x) = G_1 G_2 \ldots G_N$. Every G_i is a random bit, but these bits are in general dependent. Let i be a random integer chosen uniformly from $\{1, \ldots, N\}$. Then for every randomized polynomial time algorithm $\mathcal{B}$ that accepts the number n and a string $x \in \{0,1\}^i$ as input, and computes a bit from it,

$$\Pr\Big(\mathcal{B}(n; G_1 \ldots G_i) = G_{i+1}\Big) = \frac{1}{2} + \mathrm{NEGL}(n). \tag{7.1}$$

Informally: we try to use $\mathcal{B}$ to predict a bit of $G_1 \ldots G_N$ from the previous bits. Then the probability that we succeed is only negligibly larger than $1/2$ (again, we could achieve a success rate of $1/2$ just by guessing randomly).

Theorem 7.2.1 (A. Yao)**.** *A generator g is a safe random number generator if and only if it is unpredictable.*

Before giving the proof, let us point out an interesting consequence of this theorem. Obviously, if we reverse the output of a safe random number generator, we get another safe random number generator—any algorithm that could distinguish it from a truly random sequence could easily start with reversing it. But this implies that if a generator

is unpredictable, then also this reverse is unpredictable—and there is no easy way to see this.

Proof. I. Suppose that g is not unpredictable. This means that there is a randomized polynomial time algorithm $\mathcal{B}$, a constant $k > 0$, and infinitely many values n, such that for an i chosen randomly from $\{1, \dots, N\}$, we have

$$\Pr\Big(\mathcal{B}(n; G_1 \dots G_i) = G_{i+1}\Big) > \frac{1}{2} + \frac{1}{n^k}$$

(where $x \in \{0,1\}^n$ is a uniformly chosen random string and $g(x) = G_1 \dots G_N$).

Using this, we can perform the following randomness test $\mathcal{A}$ on strings $y = y_1 \dots y_N$: Choose a random $i \in \{1, \dots, N\}$. If $\mathcal{B}(n; y_1 \dots y_i) = y_{i+1}$, then declare y "non-random", else declare it "random".

Let us argue that this test works. Suppose that we give $\mathcal{A}$ either a truly random string $R_1 \dots R_N$ or the string $g(x)$ (each with probability 1/2). Then the probability of success is

$$\begin{aligned}&\frac{1}{2}\Pr\Big(\mathcal{B}(n; R_1 \dots R_i) \neq R_{i+1}\Big) + \frac{1}{2}\Pr\Big(\mathcal{B}(n; G_1 \dots G_i) = G_{i+1}\Big)\\ &\quad \geq \frac{1}{2}\cdot\frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + \frac{1}{n^k}\right) = \frac{1}{2} + \frac{1}{2n^k}.\end{aligned}$$

Since this is non-negligibly larger than 1/2, the generator is not safe.

II. Assume that there exists a randomized polynomial time algorithm $\mathcal{A}$ that, for infinitely many values of n, distinguishes the pseudorandom sequence $g(x) = G_1 \dots G_N$ from a truly random sequence $r = R_1 \dots R_N$ with a success probability that is at least $1/2 + n^{-k}$ for some constant $k > 0$. We want to show that in this case we can predict the i-th bit of $G_1 \dots G_N$ for a random i.

For a fixed choice of i, the success probability of $\mathcal{A}$ is

$$\frac{1}{2}\Pr(\mathcal{A}(r) = 1) + \frac{1}{2}\Pr(\mathcal{A}(g(x)) = 0).$$

(Success means that $\mathcal{A}$ accepts r, but rejects $g(x)$.) After a simple transformation this success probability is

$$\frac{1}{2} + \frac{1}{2}\Big(\Pr(\mathcal{A}(r) = 1) - \Pr(\mathcal{A}(g(x)) = 1)\Big).$$

So it follows that

$$\Pr(\mathcal{A}(r) = 1) - \Pr(\mathcal{A}(g(x)) = 1) \geq \frac{2}{n^k}. \tag{7.2}$$

The trick of the prediction algorithm $\mathcal{B}$ is to consider the mixed sequences

$$y^i = G_1 \dots G_i R_{i+1} \dots R_N$$

and subject them to the test $\mathcal{A}$. We have $y^0 = r$ and $y^N = g(x)$. We also need the sequences

$$z^i = G_1 \dots G_{i-1} \overline{G}_i R_{i+1} \dots R_N$$

where $\overline{G}_i$ denotes $1 - G_i$.

Suppose that we have seen $G_1, \ldots, G_{i-1}$. Let us flip a coin several times to get independent random bits $R_i, \ldots, R_N$. We run algorithm $\mathcal{A}$ on y^{i-1}, and predict

$$\mathcal{B}(n; G_1 \ldots G_{i-1}) = \begin{cases} R_i, & \text{if } \mathcal{A}(y^{i-1}) = 0, \\ \overline{R}_i, & \text{otherwise.} \end{cases}$$

The probability that this prediction is successful is

$$\begin{aligned} &\Pr(G_i = R_i, \mathcal{A}(y^{i-1}) = 0) + \Pr(G_i = \overline{R}_i, \mathcal{A}(y^{i-1}) = 1) \\ &\quad = \Pr(\mathcal{A}(y^{i-1}) = 0 \mid G_i = R_i) \Pr(G_i = R_i) + \\ &\Pr(\mathcal{A}(y^{i-1}) = 1 \mid G_i = \overline{R}_i) \Pr(G_i = \overline{R}_i) \\ &\quad = \frac{1}{2} \Pr(\mathcal{A}(y^i) = 0 \mid G_i = R_i) + \frac{1}{2} \Pr(\mathcal{A}(z^i) = 1 \mid G_i = \\ &\quad = \frac{1}{2} \Pr(\mathcal{A}(y^i) = 0) + \frac{1}{2} \Pr(\mathcal{A}(z^i) = 1), \end{aligned}$$

since if $G_i = R_i$, then $y^{i-1} = y^i$, while if $G_i = \overline{R}_i$, then $y^{i-1} = z^i$, and the events $\mathcal{A}(y^i) = 0$ and $\mathcal{A}(z^i) = 1$ are independent of the event $G_i = R_i$. Let us use also that

$$\begin{aligned} \Pr(\mathcal{A}(y^{i-1}) = 0) &= \frac{1}{2} \Pr(\mathcal{A}(y^i) = 0) + \frac{1}{2} \Pr(\mathcal{A}(z^i) = 0) \\ &= \frac{1}{2} + \frac{1}{2} \Pr(\mathcal{A}(y^i) = 0) - \frac{1}{2} \Pr(\mathcal{A}(z^i) = 1). \end{aligned}$$

Expressing the last term from this equation and substituting in the previous, we get

$$\Pr(\mathcal{B} \text{ is successful}) = \frac{1}{2} + \Pr(\mathcal{A}(y^{i-1}) = 1) - \Pr(\mathcal{A}(y^i) = 1).$$

This is valid for every fixed i. Choosing i at random, we have to average the right hand sides, and get

$$\begin{aligned} \Pr(\mathcal{B} \text{ is successful}) &= \frac{1}{2} + \frac{1}{N} \sum_{i=1}^{N} \Big(\Pr(\mathcal{A}(y^{i-1}) = 1) - \Pr(\mathcal{A}(y^i) = 1) \Big) \\ &= \frac{1}{2} + \frac{1}{N} \Big(\Pr(\mathcal{A}(y^0) = 1) - \Pr(\mathcal{A}(y^N) = 1) \Big) > \frac{1}{2} + \frac{1}{n^k N}, \end{aligned}$$

which is non-negligibly larger than 1/2. □

So we have defined in an exact way what a pseudorandom number generator is, and have proved some basic properties. But do such generators exist? It turns out that such generators can be constructed using some (unproved) complexity-theoretic assumptions (which are nevertheless rather plausible). This complexity-theoretic background is discussed in the next section.

7.3 One-way functions

A one-way function is a function that is "easy to compute but difficult to invert". The exact definition can be given as follows.

Definition 7.3.1. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is called a one-way function if*

- *there is a constant $c \geq 1$ such that $|x|^{1/c} < |f(x)| < |x|^c$;*
- *$f(x)$ is polynomial time computable;*
- *for every randomized polynomial time algorithm $\mathcal{A}$ that computes $0-1$ strings from $0-1$ strings, and for a string y randomly and uniformly chosen from $\{0,1\}^n$,*

$$\Pr\big(f(\mathcal{A}(f(y))) = f(y)\big) = NEG(n). \tag{7.3}$$

The third condition means that if we pick a random string y of length n, compute $f(y)$, and then try to use $\mathcal{A}$ to compute a pre-image of $f(y)$, the probability that we succeed is negligible. Note that we don't assume that f is invertible, so we cannot simply write $\mathcal{A}(f(y)) = y$.

But why don't we write simply

$$\Pr\big(f(\mathcal{A}(z)) = z\big) = NEG(n) \tag{7.4}$$

for a uniformly chosen z? The point is that since f may not be onto, it could be the case that most strings z are not in the range of f. then the above probability would be small, even if in the cases when z is in the range, a pre-image could always be easily computed. Thus (7.3) concentrates on the cases when a pre-image exists, and stipulates that even these are hard.

A *one-way permutation* is a one-way function that is one-to-one and satisfies $|f(x)| = |x|$ for every x. It is clear that under this assumption, (7.4) is equivalent to (7.3).

The main conclusion of this section is that, informally, "safe random number generators are equivalent to one-way functions". In one direction, the connection is easy to state and relatively easy to prove.

Theorem 7.3.1. *Let g be a safe random number generator, and assume that $N = |g(x)| \geq 2n$ where $n = |x|$. Then g is one-way.*

Proof. Suppose that g is not one-way, then there exists a constant $k > 0$, a randomized polynomial time algorithm $\mathcal{A}$, and infinitely many values n such that for a string y randomly and uniformly chosen from $\{0,1\}^n$,

$$\Pr\big(g(\mathcal{A}(g(y))) = g(y)\big) > \frac{1}{n^k}.$$

Now consider the following randomness test $\mathcal{B}$: we declare a string $z \in \{0,1\}^N$ "non-random" if $g(\mathcal{A}(z)) = z$, and "random" otherwise. If we give $\mathcal{B}$ either a truly random string $r = R_1 \dots R_N$ or $g(x)$ (each with probability 1/2), the probability of success is

$$\frac{1}{2}\Pr\big(g(\mathcal{A}(r)) \neq r\big) + \frac{1}{2}\Pr\big(g(\mathcal{A}(g(x))) = g(x)\big)$$

The first term is very close to 1/2; indeed, the total number of strings in the range of g is 2^n, so the probability that r is one of these of $2^{n-N} < 2^{-n}$ (and of course even if r is in the range of g, it may not be equal to $g(\mathcal{A}(r))$, which would help us but we don't have to use it). The second term is at least $1/n^k$ by assumption. Thus the probability of success is at least

$$\frac{1}{2}\left(1 - \frac{1}{2^n}\right) + \frac{1}{2}\frac{1}{n^k} > \frac{1}{2} + \frac{1}{4n^k},$$

which is non-negligibly larger than 1/2. □

In the reverse direction we describe how to use a one-way permutation f to construct a safe random number generator. (This construction is due to Goldreich and Levin.) For two 0-1 strings $u = u_1 \dots u_n$ and $v = v_1 \dots v_n$, define $u \cdot v = u_1 v_1 \oplus \dots \oplus u_n v_n$.

We describe a random number generator. We start with a function $f : \{0,1\}^n \to \{0,1\}^n$. The seed of the generator is a pair (x, p) of random sequences $x = (x_1, \dots, x_n)$ and $p = (p_1, \dots, p_n)$ (so the seed consists of $2n$ bits), and we stretch this to a pseudo-random sequence of length N as follows. Compute the strings

$$y^t = f^t(x)$$

for $t = 1, \dots, N$ (here f^t is the function f iterated t times), and let

$$G_t = p \cdot y^t, \qquad G(x,p) = G_1 \dots G_N.$$

Theorem 7.3.2. *If f is a one-way permutation, then g is a safe random number generator.*

Proof. Using Theorem 7.2.1, and the remark after it, it suffices to show that for every $1 \leq i \leq N$, and for every randomized polynomial time algorithm $\mathcal{B}$ computing a bit from $G_N \dots G_{i+1}$, the probability that this bit is G_i is only negligibly larger than 1/2. We can be generous and give the algorithm not only these bits, but all the strings $f^t(x)$ for $t \geq i+1$ and also the string p (from which $G_N \dots G_{i+1}$ is easily computed). But then, we do not have to give $f^{i+2}(x) \dots, f^N(x)$; these are easily computed from $f^{i+1}(x)$.

Since f is a permutation of $\{0,1\}^n$, the vector $f^t(x)$ is also uniformly distributed over $\{0,1\}^n$ for every t. We consider the two special vectors $y = f^i(x)$ and $z = f^{i+1}(x) = f(y)$. From the assumption that f is one-way, we know that no polynomial time algorithm can compute y from z with non-negligible success probability.

Thus the algorithm $\mathcal{B}$ gets $z = f^{i+1}(x)$ and p, and guesses $G_i = p \cdot f^{-1}(z)$. We denote this guess by $\mathcal{B}(p,z)$ or simply by $\mathcal{B}(p)$ (the dependence on z will not be essential). We show that any polynomial time algorithm that can do this with non-negligible success

probability over 1/2 can be used to compute $y = f^{-1}(z)$ with non-negligible success probability.

To warm up, let us assume that we have a polynomial time algorithm $\mathcal{B}$ that *always* gets $p \cdot y$ right. Then it is easy to compute y: its i-th bit is exactly $y_i = e_i \cdot y = \mathcal{B}(e_i)$, where $e_i = 0^{i-1}10^{n-i}$ (the string with exactly one 1 in the i-th position).

Unfortunately, we face a more difficult task: we can't be sure that $\mathcal{B}$ computes $p \cdot y$ correctly; all we know is that it computes a guess of this bit that is correct a little more than half of the time (when we average over p, z and the coin-flips that $\mathcal{B}$ may use). In particular, there is no guarantee that the algorithm gives the right result for the very special choice $p = e_i$.

The next idea is to use that

$$y_i = e_i \cdot y = (p \oplus e_i) \cdot y \oplus p \cdot y$$

for any p. This suggests that we can try to use

$$\mathcal{B}(p \oplus e_i) \oplus \mathcal{B}(p)$$

as our guess for y_i. Choosing p at random here (uniformly over $\{0,1\}^n$), we know that that we have a little, but non-negligible chance over 1/2 to get $p \cdot y$ right; and since along with p, the vector $p \oplus e_i$ is also uniformly distributed over $\{0,1\}^n$, the same holds for the first term on the right hand side. Unfortunately, this implies only a very bad bound on the chance of getting *both* of them right.

The main trick is to consider the values

$$g(p) = \mathcal{B}(p \oplus e_i) \oplus p \cdot y. \tag{7.5}$$

If for a p, $\mathcal{B}(p \oplus e_i)$ guesses right, then this bit is y_i; if it guesses wrong, then it is $\neg y_i$. Since on the average, $\mathcal{B}$ guesses more often right than wrong, we get that on the average, the number of vectors v for which this is y_i is at least $(1 + n^{-c})$ times larger than the number of terms for which it is $\neg y_i$. So it suffices to determine what is the majority of the bits $g(p)$.

There are two troubles with this approach: first, we cannot evaluate $g(p)$, since in (7.5), y is unknown! Second, it would take too long to consider all the values $g(p)$ to determine whether 0 or 1 is the majority.

Even though the first problem seems to kill the whole approach, we start with addressing the second, and - surprisingly - this will also suggest a way to treat the first one.

We can try to determine y_i by sampling: choose randomly and independently a sufficiently large number of vectors $p_1, \ldots, p_k$, and output the majority of $g(p_1), \ldots, g(p_k)$ as the guess for y_i. Probability theory (the Law of Large Numbers) tells us that the majority of the samples will be the same as the majority of all values, with large probability.

The exact computation is standard probability theory, but we go through it for later reference. Suppose (say) that $x_i = 0$, so that the majority of $g(p)$, over all vectors p, is 0, and in fact the number of vectors p with $g(p) = 1$ is $M \leq (1/2 - n^{-c})2^n$. In the sample, we expect to see $g(p_j) = 1$ about $kM2^{-n} \leq (1/2 - n^{-c})k$ times. If sampling

gives the wrong conclusion, then

$$\sum_{j=1}^{k} g(p_j) \geq \frac{k}{2},$$

and hence

$$\left(\sum_{j=1}^{k}(g(p_j) - M2^{-n})\right)^2 \geq n^{-2c}k^2. \tag{7.6}$$

Set $Z_j = g(p_j) - M2^{-n}$, then we have $\mathsf{E}(Z_j) = 0$, $\mathsf{E}(Z_jZ_l) = \mathsf{E}(Z_i)\mathsf{E}(Z_l) = 0$ if $j \neq l$ (since Z_i and Z_j are independent) and $\mathsf{E}(Z_j^2) = M2^{-n} - M^2 2^{-2n} < 1$. From this it is easy to compute the expectation of the left hand side of (7.6):

$$\mathsf{E}\left(\left(\sum_j Z_j\right)^2\right) = \sum_j \mathsf{E}(Z_j^2) - 2\sum_{1\leq j<l\leq n} \mathsf{E}(Z_jZ_l) < k.$$

Thus the probability that (7.6) occurs is, by Markov's inequality, less than $k/(n^{-2c}k^2) = n^{2c}/k$.

An important point to make is that to reach this conclusion we don't need independent samples: it suffices to assume that the vectors $p_1, \ldots, p_k$ are pairwise independent. This will be significant, because to generate pairwise independent samples, we need "less randomness". In fact, let $k = 2^r - 1$, and pick only r vectors $p_1, \ldots, p_r$ uniformly and independently from $\{0,1\}^n$, let $p_{r+1}, \ldots, p_k$ be all non-trivial linear combinations of them over GF(2) (say $p_{r+1} = p_1 \oplus p_2$, $p_{r+2} = p_1 \oplus p_2 \oplus p_3$ etc.; it does not matter in which order we go through these linear combinations). Then it is easy to check that the vectors $p_1, \ldots, p_k$ are pairwise independent, and hence can be used as sample vectors.

It may be nice to save on coin flips, but this way of generating $p_1, \ldots, p_k$ has a further, much more substantial advantage: it provides a way out from the trouble that we don't know y in (7.5). Indeed, only need to know the values $p_1 \cdot y, \ldots, p_k \cdot y$; and for this, it suffices to know the values $p_1 \cdot y, \ldots, p_r \cdot y$ (since then we have $p_{r+1} \cdot y = p_1 \cdot y \oplus p_2 \cdot y$ etc.).

So we need only r bits of information about y; this is much less than n, but how are we going to get it? The answer is, that we don't. We just try all possible r-tuples of 0 and 1. This is only $2^j = k + 1 = O(n^{2c})$ cases to consider. For each such trial, we try to reconstruct y as described above. We'll know when we succeed, since then we find $f(y) = z$, and we are done. And this happens with non-negligible probability. □

7.4 Candidates for one-way functions

Number theory provides several candidates of one-way functions. The length of inputs and outputs will not be exactly n, only polynomial in n.

Problem 7.4.1 (The factoring problem)**.** *Let x represent a pair of primes of length n (say, along with a proof of their primality). Let $f(n, x)$ be their product. Many special cases of this problem are solvable in polynomial time but still, a large fraction of the instances remains difficult.*

Problem 7.4.2 (The discrete logarithm problem)**.** *Given a prime number p, a primitive root g for p and a positive integer $i < p$, we output p, g and $y = g^i \bmod p$. The inversion problem for this is called the discrete logarithm problem since given p, g, y, what we are looking for is i which is also known as the index, or discrete logarithm, of y with respect to p.*

Problem 7.4.3 (The discrete square root problem)**.** *Given positive integers m and $x < m$, the function outputs m and $y = x^2 \bmod m$. The inversion problem is to find a number x with $x^2 \equiv y \pmod m$. This is solvable in polynomial time by a probabilistic algorithm if m is a prime but is considered difficult in the general case.*

7.4.1 Discrete square roots

In this section we discuss a number theoretic algorithm to extract square roots.

We call the integers $0, 1, \ldots, p-1$ *residues* (modulo p). Let p be an odd prime. We say that y is a *square root* of x (modulo p), if

$$y^2 \equiv x \pmod p.$$

If x has a square root then it is called a *quadratic residue*.

Obviously, 0 has only one square root modulo p: if $y^2 \equiv 0 \pmod p$, then $p|y^2$, and since p is a prime, this implies that $p|y$. For every other residue x, if y is a square root of x, then so is $p - y = -y \pmod p$. There are no further square roots: indeed, if $z^2 \equiv x$ for some residue z, then $p|y^2 - z^2 = (y - z)(y + z)$ and so either $p|y - z$ or $p|y + z$. Thus $z \equiv y$ or $z \equiv -y$ as claimed.

This implies that not every integer has a square root modulo p: squaring maps the non-zero residues onto a subset of size $(p-1)/2$, and the other $(p-1)/2$ have no square root.

The following lemma provides an easy way to decide if a residue has a square root.

Lemma 7.4.1. *A residue x has a square root if and only if*

$$x^{(p-1)/2} \equiv 1 \pmod p. \tag{7.7}$$

Proof. The "only if" part is easy: if x has a square root y, then

$$x^{(p-1)/2} \equiv y^{p-1} \equiv 1 \pmod p$$

by Fermat's Little Theorem. Conversely, the polynomial $x^{(p-1)/2} - 1$ has degree $(p - 1)/2$, and hence it has at most $(p - 1)/2$ roots modulo p (this can be proved just like the well-know theorem that a polynomial of degree n has at most n real roots). Since all quadratic residues are roots of $x^{(p-1)/2} - 1$, none of the quadratic non-residues can be. □

But how to find a square root? For some primes, this is easy.

Lemma 7.4.2. *Assume that $p \equiv 3 \pmod 4$. Then for every quadratic residue x, $x^{(p+1)/4}$ is a square root of x.*

Proof.

$$\left(x^{(p+1)/4}\right)^2 = x^{(p+1)/2} = x \cdot x^{(p-1)/2} \equiv x \pmod p.$$

□

The case when $p \equiv 1 \pmod 4$ is more difficult, and the only known polynomial time algorithms use randomization. In fact, randomization is only needed in the following auxiliary algorithm:

Lemma 7.4.3. *Let p be an odd prime. Then we can find a quadratic non-residue modulo p in randomized polynomial time.*

This can be done by selecting a random residue $z \neq 0$, and then testing (using lemma 7.4.1 whether it is a quadratic residue. If not, we try another z. Since the chance of hitting one is $1/2$, we find one in an average of two trials.

Remark. One could, of course, try to avoid randomization by testing the residues $2, 3, 5, 7, \ldots$ to see if they have a square root. Sooner or later we will find a quadratic non-residue. However, it is not known whether the smallest quadratic non-residue will be found in polynomial time this way. It is conjectured that one never has to try more than $O(\log^2 p)$ numbers this way.

Now let us return to the problem of finding the square root of a residue x, in the case when p is a prime satisfying $p \equiv 1 \pmod 4$. We can write $p - 1 = 2^k q$, where q is odd and $k \geq 2$.

We start with finding a quadratic non-residue z. The trick is to find an even power z^{2t} such that $x^q z^{2t} \equiv 1 \pmod p$. Then we can take $y = x^{(q+1)/2} z^t \pmod p$. Indeed,

$$y^2 \equiv x^{q+1} z^{2t} \equiv x \pmod p.$$

To construct such a power of z, we consruct for all $j \leq k - 1$ an integer $t_j > 0$ such that

$$x^{2^j q} z^{2^{j+1} t_j} \equiv 1 \pmod p. \tag{7.8}$$

For $j = 0$, this is just what we need. For $j = k - 1$, we can take $t_{k-1} = q$:

$$x^{2^{k-1} q} z^{2^k q} = x^{(p-1)/2} z^{p-1} \equiv 1 \pmod p,$$

since x is a quadratic residue and $z^{p-1} \equiv 1 \pmod p$ by Fermat's "Little" theorem. This suggests that we construct the number t_j "backwards" for $j = k - 2, k - 3, \ldots$

Suppose that we have t_j, $j > 0$, and we want to construct t_{j-1}. We know that

$$p \left| x^{2^j q} z^{2^{j+1} t_j} - 1 = \left(x^{2^{j-1} q} z^{2^j t_j} - 1\right)\left(x^{2^{j-1} q} z^{2^j t_j} + 1\right)\right.$$

We test which of the two factors is a multiple of p. If it is the first, we can simply take $t_{j-1} = t_j$. So suppose that it is the second. Now take

$$t_{j-1} = t_j + 2^{k-j-1} q.$$

Then

$$x^{2^{j-1}q} z^{2^j t_{j-1}} = x^{2^{j-1}q} z^{2^j t_j + 2^{k-1}q} = x^{2^{j-1}q} z^{2^j t_j} z^{(p-1)/2} \equiv (-1)(-1) = 1,$$

since z is a quadratic non-residue.

This completes the description of the algorithm.

Exercise 7.4.1. *Show that squaring an integer is not a safe random number generator.*

Exercise 7.4.2. *For a string x, let $rev(x)$ denote the reverse string. Show that if $g(s)$ is a safe random number generator, then so is $rev(g(s))$.*

Exercise 7.4.3. *If P=NP then no one-way function exists.*

Exercise 7.4.4. *Somebody proposes the following random number generator: it takes an integer x with n bits as the seed, and outputs $\lfloor x^3/2^n \rfloor$. Show that this random number generator is not safe.*

Chapter 8

Decision trees

The logical framework of many algorithms can be described by a tree: we start from the root and in every internal node, the result of a certain "test" determines which way we continue. E.g., most sorting algorithms make comparisons between certain pairs of elements and continue the work according to the result of the comparison. We assume that the tests performed in such computations contain all the necessary information about the input, i.e., when we arrive at a leaf of the tree, all that is left is to read off the output from the leaf. The complexity of the tree gives some information about the complexity of the algorithm; for example, the depth of the tree (the number of edges in the longest path leaving the root) tells us how many tests must be performed in the worst case during the computation. We can describe, of course, every algorithm by a trivial tree of depth 1 (the test performed in the root is the computation of the end result). This algorithmic scheme makes sense only if we restrict the kind of tests allowed in the nodes.

We will see that decision trees not only give a graphical representation of the structure of some algorithms but are also suitable for proving lower bounds on their depth. Such a lower bound can be interpreted as saying that the problem cannot be solved (for the worst input) in fewer steps than some given number, if we assume that information on the input is available only by the permissible tests (for example, in sorting we can only compare the given numbers with each other and cannot perform e.g., arithmetic operations on them).

8.1 Algorithms using decision trees

Consider some simple examples.

0) Binary search.

Perhaps the simplest situation in which decision trees are used is *binary search.* We want to compute an integer a, about which at the beginning we only konw that it lies, say, in the interval $[1, N]$. We have an algorithm that, given any integer m, $1 \leq m \leq N$, can decide whether $a \leq m$ is true. Then by calling this algorithm $\lceil \log N \rceil$ times, we can determine a. We have used this method when we showed that factoring an integer can be reduced to the problem of finding a bounded divisor.

We can describe this algorithm by a rooted binary tree: every node will correspond to an interval $[u, v] \subseteq [1, N]$. The root corresponds to the interval $[1, N]$, and each node corresponds to the interval of integers that are still possible values for a if we arrive at the node. The leaves correspond to one-element intervals, i.e., the possible values of a. For an internal node corresponding to the interval $[u, v]$, we select $w = \lfloor (u + v)/2 \rfloor$ and test if $a \leq w$. Depending on the outcome of this test, we proceed to one of the children of the node, which correspond to the intervals $[u, w]$ and $[w + 1, v]$.

a) Finding a false coin with a one-armed scale.

We are given n coins looking identical from the outside. We know that each must weigh 1 unit; but we also know that there is a false one among them that is lighter than the rest. We have a one-armed scale; we can measure with it the weight of an arbitrary subset of the coins. How many measurements are enough to decide which coin is false?

The solution is simple: with one measurement, we can decide about an arbitrary set of coins whether the false one is among them. If we put $\lceil n/2 \rceil$ coins on the scale, then after one measurement, we have to find the false coin only among at most $\lceil n/2 \rceil$ ones. This recursion ends in $\lceil \log_2 n \rceil$ steps.

We can characterize the algorithm by a rooted binary tree. Every vertex v corresponds to a set X_v of coins; arriving into this vertex we already know that the false coin is to be found in this set. (The root corresponds to the original set, and the leaves to the 1-element sets.) For every internal node v, we divide the set X_v into two parts, with $\lceil |X_v|/2 \rceil$ and $\lfloor |X_v|/2 \rfloor$ elements, respectively. These correspond to the children of v. Measuring the first one we learn which one contains the false coin.

b) Finding a false coin with a two-armed scale.

Again, we are given n identically looking coins. We know that there is a false one among them that is lighter than the rest. This time we have a two-armed scale but without weights. On this, we can find out which one of two (disjoint) sets of coins is lighter, or whether they are equal. How many measurements suffice to decide which coin is false?

Here is a solution. One measurement consists of putting the same number of coins into each pan. If one side is lighter then the false coin is in that pan. If the two sides have equal weight then the false coin is among the ones left out. It is most practical to put $\lceil n/3 \rceil$ coins into both pans; then after one measurement, the false coin must be found only among at most $\lceil n/3 \rceil$ coins. This recursion terminates in $\lceil \log_3 n \rceil$ steps.

Since one measurement has 3 possible outcomes, the algorithm can be characterized by a rooted tree in which each internal node has 3 children. Every node v corresponds to a set X_v of coins; arriving into this node we already know that the false coin is in this set. (As above, the root corresponds to the original set and the leaves to the one-element sets.) For each internal node v, we divide the set X_v into three parts, with $\lceil |X_v|/3 \rceil$, $\lceil |X_v|/3 \rceil$ and $|X_v| - 2\lceil |X_v|/3 \rceil$ elements. These correspond to the children of v. Comparing the two first ones we can find out which one of the three sets contains the false coin.

Exercise 8.1.1. *Prove that fewer measurements do not suffice in either problem a) or b).*

c) Sorting.

Given are n elements that are ordered in some way (unknown to us). We know a procedure to decide the order of two elements; this is called a *comparison* and considered an elementary step. We would like to determine the complete ordering using as few comparisons as possible. Many algorithms are know for this basic problem of data processing; we treat this question only to the depth necessary for the illustration of decision trees.

Obviously, $\binom{n}{2}$ comparisons are enough: with these, we can learn about every pair of elements, which one in the pair is greater, and this determines the complete order. These comparisons are not, however, independent: often, we can infer the order of certain pairs using transitivity. Indeed, it is enough to make $\sum_{k=1}^{n} \lceil \log k \rceil \sim n \log n$ comparisons. Here is the simplest way to see this: suppose that we already determined the ordering of the first $n-1$ elements. Then already only the n-th element must be "inserted", which can obviously be done with $\lceil \log n \rceil$ comparisons.

This algorithm, as well as any other sorting algorithm working with comparisons, can be represented by a binary tree. The root corresponds to the first comparison; depending on its result, the algorithm branches into one of the children of the root. Here, we make another comparison, etc. Every leaf corresponds to a complete ordering.

Remark. In the above sorting algorithm, we only counted the comparisons. For a real implementation, one should also take into account the other operations, e.g., the movement of data, etc. From this point of view, the above algorithm is not good since every insertion may require the movement of all elements placed earlier and this may cause $\Omega(n^2)$ extra steps. There exist, however, sorting algorithms requiring altogether only $O(n \log n)$ steps.

d) Convex hull

The determination of the convex hull of n planar points is as basic among the geometrical algorithms as sorting for data processing. The points are given by their coordinates: $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$. We assume, for simplicity, that the points are in *general position*, i.e., no 3 of them is on one straight line. We want to determine those indices $i_0, \ldots, i_{k-1}, i_k = i_0$ for which $p_{i_0}, \ldots, p_{i_{k-1}}, p_{i_k}$ are the vertices of the convex hull of the given point set, in this order along the convex hull (starting anticlockwise, say, from the point with the smallest abscissa).

The idea of "insertion" gives a simple algorithm here, too. Sort the elements by their x_i coordinates; this can be done in time $O(n \log n)$. Suppose that $p_1, \ldots, p_n$ are already indexed in this order. Delete the point p_n and determine the convex hull of the points $p_1, \ldots, p_{n-1}$: let this be the sequence of points $p_{j_0}, \ldots, p_{j_{m-1}}, p_{j_m}$ where $j_0 = j_m = 1$.

Now, the addition of p_n consists of deleting the arc of the polygon $p_{j_0}, \ldots, p_{j_m}$ "visible" from p_n and replacing it with the point p_n. Let us determine the first and last elements of the sequence $p_{j_0}, \ldots, p_{j_m}$ visible from p_n, let these be p_{j_a} and p_{j_b}. Then the convex hull sought for is $p_{j_0}, \ldots, p_{j_a}, p_n, p_{j_b}, p_{j_m}$ (Figure 8.1).

How to determine whether some vertex p_{j_s} is visible from p_n? The point p_{n-1} is evidently among the vertices of the polygon and is visible from p_n; let $j_t = n-1$. If $s < t$ then, obviously, p_{j_s} is visible from p_n if and only if p_n is below the line $p_{j_s}p_{j_{s+1}}$. Similarly, if $s > t$ then p_{j_s} is visible from p_n if and only if p_n is above the line $p_{j_s}p_{j_{s-1}}$. In this way, it can be decided about every p_{j_s} in $O(1)$ steps whether it is visible from p_n.

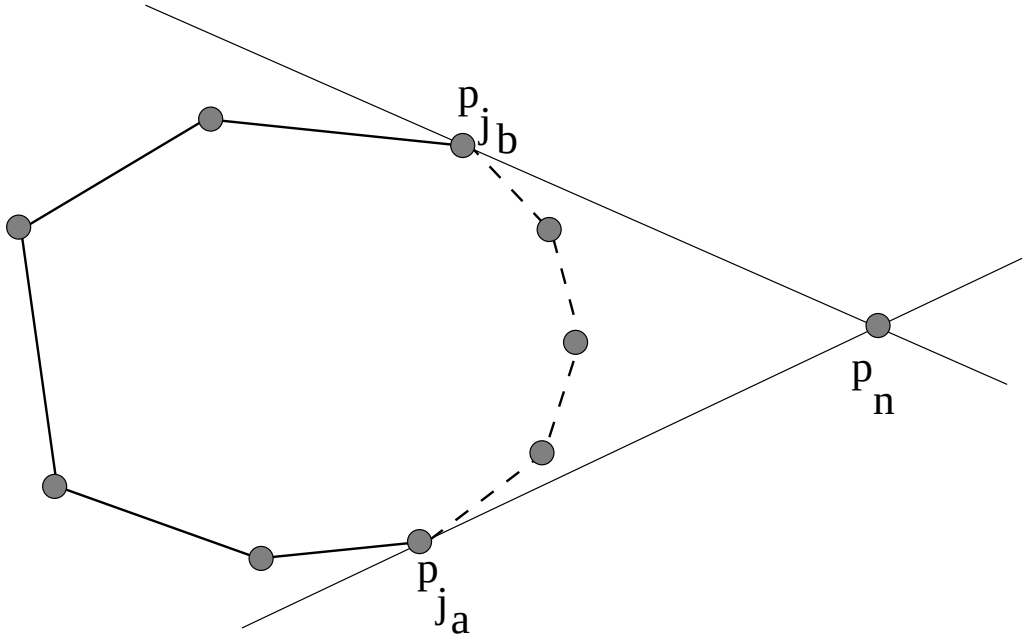

Figure 8.1: A simple convex hull algorithm

Using this, we can determine a and b in $O(\log n)$ steps and we can perform the "insertion" of the point p_n. This recursion gives an algorithm with $O(n \log n)$ steps.

It is worth separating here the steps in which we do computations with the coordinates of the points, from the other steps (of combinatorial character). We do not know namely, how large are the coordinates of the points, whether multiple-precision computation is needed, etc. Analysing the described algorithm, we can see that the coordinates needed to be taken into account only in two ways: at the sorting, when we had to make comparisons among the abscissas, and at deciding whether point p_n was above or below the straight line determined by the points p_i and p_j. The last one can be also formulated by saying that we must determine the orientation of the triangle $p_i p_j p_k$. This can be done in several ways using the tools of analytic geometry without division.

The above algorithm can again be described by a binary decision tree: each of its nodes corresponds either to the comparison of the abscissas of two given points or to the determination of the orientation of a triangle given by three points. The algorithm gives a tree of depth $O(n \log n)$. (Many other algorithms looking for the convex hull lead to a decision tree of similar depth.)

Exercise 8.1.2. *Show that the problem of sorting n real numbers can be reduced in a linear number of steps to the problem of determining the convex hull of n planar points.*

Exercise 8.1.3. *Show that the determination of the convex hull of the points $p_1, \ldots, p_n$ can be performed in $O(n)$ steps provided that the points are already sorted by their x coordinates.*

To formalize the notion of a decision tree let us be given the set A of possible inputs, the set B of possible outputs and a set Φ of functions defined on A with values in $\{1, \ldots, d\}$, the *test-functions*. A *decision tree* is a rooted tree whose internal nodes

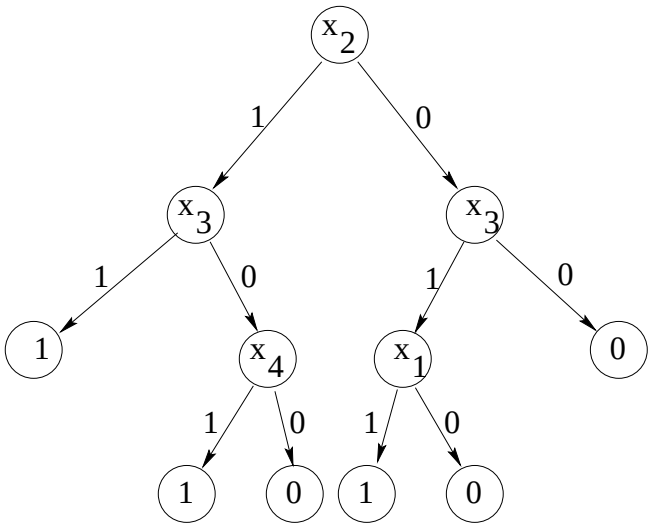

Figure 8.2: A simple decision tree

(including the root) have d children (the tree is d-regular), its leaves are labeled with the elements of B, the other nodes with the functions of Φ. We assume that for every vertex, the edges to its children are numbered in some order.

Every decision tree determines a function $f : A \to B$. For any $a \in A$, starting from the root, we walk down to a leaf as follows. If we are in an internal node v then we compute the test function assigned to v at the place a; if its value is i then we step further to the i-th child of node v. In this way, we arrive at a leaf w; the value of $f(a)$ is the label of w.

The question is that for a given function f, what is the decision tree with minimum depth computing it.

In the simplest case, we want to compute a Boolean function $f(x_1, \ldots, x_n)$ and every test that can be made in the vertices of the decision tree is testing the value of one of the variables. In this case, we call the decision tree *simple*. Every simple decision tree is binary, the internal nodes are indexed with the variables, the leaves with 0 and 1.

The decision tree corresponding to binary search over the interval $[1, 2^n]$ can be considered as simple, if the variables are the bits of the number and we regard the consecutive comparisons as asking for the next bit of the unknown number a. The decision tree for sorting is not simple: there, the tests (comparisons) are not independent since the ordering is transitive. We denote by $D(f)$ the minimal depth of a simple decision tree computing a Boolean function f.

Example 8.1.1. Consider the Boolean function

$$f(x_1, x_2, x_3, x_4) = (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4).$$

This is computed by the simple decision tree in Figure 8.2. This shows that $D(f) \leq 3$. It is easy to see that we cannot compute this funtion by a simple decision tree of depth 2, and hence $D(f) = 3$.

Every decision tree can also be considered a two-person "twenty questions"-like game. One player (Xavier) thinks of an element $a \in A$, and it is the task of the

other player (Yvette) to determine the value of $f(a)$. For this, she can pose questions to Xavier. Her questions cannot be, however, arbitrary, she can only ask the value of some test function in Φ. How many questions do suffice for her to compute the answer? Yvette's strategy corresponds to a decision tree, and Xavier plays optimally if with his answers, he drives Yvette to the leaf farthest away from the root. (Xavier can "cheat, as long as he is not caught"—i.e., he can change his mind about the element $a \in A$ as long as the new one still makes all his previous answers correct. In case of a simple decision tree, Xavier has no such worry at all.)

8.2 Nondeterministic decision trees

The notion learned in Chapter 5, *nondeterminism*, helps in other complexity-theoretic investigations, too. In the decision-tree model, the same idea can be formulated as follows (we will only consider the case of simple decision trees). Let $f : \{0,1\}^n \to \{0,1\}$ be the function to be computed. Two numbers characterize the nondeterministic decision-tree complexity (similarly to having two complexity classes for non-deterministic polynomial time, namely NP and co-NP). For every input x, let $D(f,x)$ denote the minimum number of those variables whose value already determines the value of $f(x)$. Let

$$D_0(f) = \max\{D(f,x) : f(x) = 0\}, \qquad D_1(f) = \max\{D(f,x) : f(x) = 1\}.$$

In other words, $D_0(f)$ is the smallest number with the property that for all inputs x with $f(x) = 0$, we can test $D_0(f)$ variables in such a way that knowing these, the value of the function is already determined (it may depend on x which variables we will test). The number $D_1(f)$ can be characterized similarly. Obviously,

$$D(f) \geq \max\{D_0(f), D_1(f)\}.$$

It can be seen from the examples below that equality does not necessarily hold here.

Example 8.2.1. Assign a Boolean variable x_e to each edge e of the complete graph K_n. Then every assignment corresponds to an n-point graph (we connect with edges those pairs whose assigned value is 1). Let f be the Boolean function with $\binom{n}{2}$ variables whose value is 1 if in the graph corresponding to the input, the degree of every node is at least one and 0 otherwise (i.e., if there is an isolated point). Then $D_0(f) \leq n-1$ since if there is an isolated point in the graph it is enough to know about the $n-1$ edges leaving it that they are not in the graph. It is also easy to see that we cannot infer an isolated point from the adjacency or nonadjacency of $n-2$ pairs, and thus

$$D_0(f) = n - 1.$$

Similarly, if there are no isolated points in a graph then this can be proved by the existence of $n-1$ edges (it is enough to know one edge leaving each node and at least one of the edges even covers 2 nodes). If the input graph is an $(n-1)$-star then fewer than $n-1$ edges are not enough. Therefore

$$D_1(f) = n - 1.$$

Thus, whichever is the case, we can know the answer after $n-1$ lucky questions. On the other hand, if we want to decide which one is the case then we cannot know in advance which edges to ask; it can be shown that the situation is as bad as it can be, namely

$$D(f) = \binom{n}{2}.$$

We return to the proof of this in the next section (exercise 8.3.3).

Example 8.2.2. Let now G be an arbitrary n-point graph an let us assign a variable to each of its vertices. An assignment of the variables corresponds to a subset of the vertices. Let the value of the function f be 0 if this set is independent in the graph and 1 otherwise. This property can be simply expressed by a Boolean formula:

$$f(x_1, \dots, x_n) = \bigvee_{ij \in E(G)} (x_i \wedge x_j).$$

If the value of this Boolean function is 1 then this can be found out already from testing 2 vertices, but of course not from testing a single point, i.e.

$$D_1(f) = 2.$$

On the other hand, if after testing certain points we are sure that the set is independent then the vertices that we did not ask must form an independent set. Thus

$$D_0(f) \geq n - \alpha$$

where α is the maximum number of independent points in the graph. It can also be proved (see Theorem 8.3.6) that if n is a prime and a cyclic permutation of the points of the graph maps the graph onto itself, and the graph has some edges but is not complete, then

$$D(f) = n.$$

We see therefore that $D(f)$ can be substantially larger than the maximum of $D_0(f)$ and $D_1(f)$, moreover, it can be that $D_1(f) = 2$ and $D(f) = n$. However, the following beautiful relation holds:

Theorem 8.2.1.

$$D(f) \leq D_0(f) D_1(f)$$

if f is non-constant.

Proof. We use induction over the number n of variables. If $n = 1$ then the inequality is trivial. It is also trivial if f is constant, so from now on we suppose f is not constant.

Let, say, $f(0, \dots, 0) = 0$; then $k \leq D_0(f)$ variables can be chosen such that fixing their values to 0, the function is 0 independently of the other variables. We can assume that the first k variables have this property.

Next, consider the following decision algorithm. We ask the value of the first k variables; let the obtained answers be $a_1, \ldots, a_k$. Fixing these, we obtain a Boolean function

$$g(x_{k+1}, \ldots, x_n) = f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n).$$

Obviously, $D_0(g) \leq D_0(f)$ and $D_1(g) \leq D_1(f)$. We claim that the latter inequality can be strengthened:

$$D_1(g) \leq D_1(f) - 1.$$

Consider an input $(a_{k+1}, \ldots, a_n)$ of g with $g(a_{k+1}, \ldots, a_n) = 1$. (If $g \equiv 0$, then the above inequality is trivially true, so we can suppose such $(a_{k+1}, \ldots, a_n)$ exist.) Together with the bits $a_1, \ldots, a_k$, this gives an input of the Boolean function f for which $f(a_1, \ldots, a_n) = 1$. According to the definition of the quantity $D_1(f)$, one can choose $m \leq D_1(f)$ variables, say, $x_{i_1}, \ldots, x_{i_m}$ of f such that fixing them some values a_{i_j}, the value of f becomes 1 independently of the other variables. One of the first k variables must occur among these m variables; otherwise, $f(0, \ldots, 0, a_{k+1}, \ldots, a_n)$ would have to be 0 (due to the fixing of the first k variables) but would also have to be 1 (due to the fixing of $x_{i_1}, \ldots, x_{i_m}$), which is a contradiction. Thus, in the function g, at the position $a_{k+1}, \ldots, a_k$, only $m-1$ variables must be fixed to obtain the identically 1 function. From this, the claim follows. From the induction hypothesis,

$$D(g) \leq D_0(g)D_1(g) \leq D_0(f)(D_1(f) - 1)$$

and hence

$$D(f) \leq k + D(g) \leq D_0(f) + D(g) \leq D_0(f)D_1(f).$$

□

In Example 8.2.2, we could define the function by a disjunctive 2-normal form and then $D_1(f) = 2$ follows. This is not an accidental coincidence.

Proposition 8.2.2. *If f is expressible by a disjunctive k-normal form then $D_1(f) \leq k$. If f is expressible by a conjunctive k-normal form then $D_0(f) \leq k$.*

Proof. It is enough to prove the first assertion. Let $(a_1, \ldots, a_n)$ be an input for which the value of the function is 1. Then there is an elementary conjunction in the disjunctive normal form whose value is 1. If we fix the variables occurring in this conjunction then the value of the function will be 1 independently of the values of the other variables. □

In fact, the reverse is also true.

Proposition 8.2.3. *A non-constant Boolean function is expressible by a disjunctive [conjunctive] k-normal form if and only if $D_1(f) \leq k$ [$D_0(f) \leq k$].*

Proof. Let $\{x_{i_1}, \ldots, x_{i_m}\}$ be a subset of the variables minimal with respect to containment, that can be fixed in such a way as to make the obtained function is identically 1. (Such a subset is called a *minterm*.)

We will show that $m \leq k$. Let us namely assign the value 1 to the variables $x_{i_1}, \ldots, x_{i_m}$ and 0 to the others. According to the foregoing, the value of the function is 1. By the definition of the quantity $D_1(f)$, we can fix in this assignment k values in such a way as to make the function identically 1. We can assume that we only fix 1's, i.e., we only fix some of the variables $x_{i_1}, \ldots, x_{i_m}$. But then, due to the minimality of the set $\{x_{i_1}, \ldots, x_{i_m}\}$, we had to fix all of them, and hence $m \leq k$.

Let us prepare for every minterm S the elementary conjunction $E_S = \bigwedge_{x_i \in S} x_i$ and take the disjunction of these. We obtain a disjunctive k-normal form this way. It is easy to check that this defines the function f. □

8.3 Lower bounds on the depth of decision trees

We mentioned that decision trees as computation models have the merit that non-trivial lower bounds can be given for their depth. First we mention, however, a non-trivial lower bound also called *information-theoretic estimate.*

Lemma 8.3.1. *If the range of f has t elements then the depth of every decision tree of degree d computing f is at least* $\log_d t$.

Proof. A d-regular rooted tree of depth h has at most d^h leaves. Since every element of the range of f must occur as a label of a leaf it follows that $t \geq d^h$. □

As an application, let us take an arbitrary sorting algorithm. The input of this is a permutation $a_1, \ldots, a_n$ of the elements $1, 2, \ldots, n$, its output is the ordered sequence $1, 2, \ldots, n$, while the test functions compare two elements:

$$\varphi_{ij}(a_1, \ldots, a_n) = \begin{cases} 1, & \text{if } a_i < a_j \\ 0, & \text{otherwise.} \end{cases}$$

Since there are $n!$ possible outputs, the depth of any binary decision tree computing the complete order is at least $\log n! \sim n \log n$. The sorting algorithm mentioned in the introduction makes at most $\lceil \log n \rceil + \lceil \log(n-1) \rceil + \cdots + \lceil \log 1 \rceil \sim n \log n$ comparisons.

This bound is often very weak; if e.g., only a single bit must be computed then it says nothing. Another simple trick for proving lower bounds is the following observation.

Lemma 8.3.2. *Assume that there is an input $a \in A$ such that no matter how we choose k test functions, say, $\varphi_1, \ldots, \varphi_k$, there is an $a' \in A$ for which $f(a') \neq f(a)$ but $\varphi_i(a') = \varphi_i(a)$ holds for all $1 \leq i \leq k$. Then the depth of every decision tree computing f is greater than k.*

For application, let us see how many comparisons suffice to find the largest one of n elements. In a championship by elimination $n-1$ comparisons are enough for this. Lemma 8.3.1 gives only $\log n$ for lower bound; but we can apply Lemma 8.3.2 as follows. Let $a = (a_1, \ldots, a_n)$ be an arbitrary permutation, and consider $k < n-1$ comparison tests. The pairs (i, j) for which a_i and a_j will be compared form a graph G over the underlying set $\{1, \ldots, n\}$. Since it has fewer than $n-1$ edges this graph falls into two disconnected parts, G_1 and G_2. Without loss of generality, let G_1 contain

the maximal element and let p denote its number of vertices. Let $a' = (a'_1, \dots a'_n)$ be the permutation containing the numbers $1, \dots, p$ in the positions corresponding to the vertices of G_1 and the numbers $p+1, \dots, n$ in those corresponding to the vertices of G_2; the order of the numbers within both sets must be the same as in the original permutation. Then the maximal element is in different places in a and in a' but the given k tests give the same result for both permutations.

Exercise 8.3.1. *Show that to pick the median of $2n+1$ numbers,*
(a) at least $2n$ comparisons are needed;
(b) $O(n)$ comparisons suffice.*

In what follows we show estimates for the depth of some more special decision trees, applying, however, some more interesting methods. First we mention a result of Best, Schrijver and van Emde-Boas, and Rivest and Vuillemin, which gives a lower bound of unusual character for the depth of decision trees.

Theorem 8.3.3. *Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function. Let N denote the number of those substitutions making the value of the function "1" and let 2^k be the largest power of 2 dividing N. Then the depth of any simple decision tree computing f is at least $n-k$.*

Proof. Consider an arbitrary simple decision tree of depth d that computes the function f, and a leaf of this tree. Here, $m \le d$ variables are fixed, therefore there are at least 2^{n-m} inputs leading to this leaf. All of these correspond to the same function value, therefore the number of inputs leading to this leaf and giving the function value "1" is either 0 or 2^{n-m}. This number is therefore divisible by 2^{n-d}. Since this holds for all leaves, the number of inputs giving the value "1" is divisible by 2^{n-d} and hence $k \ge n-d$. □

With the suitable extension of the above argument we can prove the following theorem (details of the proof are left as an exercise to the reader).

Theorem 8.3.4. *Given an n-variable Boolean function f, construct the following polynomial: $\Psi_f(t) = \sum f(x_1, \dots, x_n) t^{x_1+\dots+x_n}$ where the summation extends to all $(x_1, \dots, x_n) \in \{0,1\}^n$. Prove that if f can be computed by a simple decision tree of depth d, then $\Psi_f(t)$ is divisible by $(t+1)^{n-d}$.* □

We call a Boolean function f of n variables *evasive* if it cannot be computed by a decision tree of length smaller than n. It follows from Theorem 8.3.3 that *if a Boolean function has an odd number of substitutions making it "1" then the function is evasive.*

We obtain another important class of evasive functions by symmetry-conditions. A Boolean function is called *symmetric* if every permutation of its variables leaves its value unchanged. E.g., the functions $x_1 + \dots + x_n$, $x_1 \vee \dots \vee x_n$ and $x_1 \wedge \dots \wedge x_n$ are symmetric. A Boolean function is symmetric if and only if its value depends only on how many of its variables are 0 or 1.

Proposition 8.3.5. *Every non-constant symmetric Boolean function is evasive.*

Proof. Let $f : \{0,1\}^n \to \{0,1\}$ be the Boolean function in question. Since f is not constant, there is a j with $1 \le j \le n$ such that if $j-1$ variables have value 1 then the

function's value is 0 but if j variables are 1 then the function's value is 1 (or the other way around).

Using this, we can propose the following strategy to Xavier. Xavier thinks of a 0-1-sequence of length n and Yvette can ask the value of each of the x_i. Xavier answers 1 on the first $j-1$ questions and 0 on every following question. Thus after $n-1$ questions, Yvette cannot know whether the number of 1's is $j-1$ or j, i.e., she cannot know the value of the function. □

Symmetric Boolean functions are very special; the following class is significantly more general. A Boolean function of n variables is called *weakly symmetric* if for all pairs x_i, x_j of variables, there is a permutation of the variables that takes x_i into x_j but does not change the value of the function. e.g., the function

$$(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee \cdots \vee (x_{n-1} \wedge x_n) \vee (x_n \wedge x_1)$$

is weakly symmetric but not symmetric. The question below (the so-called generalized Aandera-Rosenberg-Karp conjecture) is open:

Conjecture 8.3.1. *If a non-constant monotone Boolean function is weakly symmetric then it is evasive.*

We show that this conjecture is true in an important special case.

Theorem 8.3.6. *If a non-constant monotone Boolean function is weakly symmetric and the number of its variables is a prime number then it is evasive.*

Proof. Let p be the number of variables (emphasizing that this number is a prime). We use the group-theoretic result that if a prime p divides the order of a group, then the group has an element of order p. In our case, those permutations of the variables that leave the value of the function invariant form a group, and from the week symmetry it follows that the order of this group is divisible by p. Thus the group has an element of order p. This means that with a suitable labeling of the variables, the substitution $x_1 \to x_2 \to \cdots \to x_p \to x_1$ does not change the value of the function.

Now consider the number

$$M = \sum f(x_1, \ldots, x_p)(p-1)^{x_1+\cdots+x_p} = \Psi_f(p-1). \tag{8.1}$$

It follows that in the definition of M, if in some term, not all the values $x_1, \ldots, x_p$ are the same, then p identical terms can be made from it by cyclic substitution. The contribution of such terms is therefore divisible by p. Since the function is not constant and is monotone, it follows that $f(0, \ldots, 0) = 0$ and $f(1, \ldots, 1) = 1$, from which it can be seen that M gives remainder $(-1)^p$ modulo p, which contadicts Theorem 8.3.4. □

Important examples of weakly symmetric Boolean functions are any *graph properties*. Consider an arbitrary property of graphs, e.g., planarity; we only assume that if a graph has this property then every graph isomorphic to it also has it. We can specify a graph with n points by fixing its vertices (let these be $1, \ldots, n$), and for all pairs $\{i, j\} \subseteq \{1, \ldots, n\}$, introduce a Boolean variable x_{ij} with value 1 if i and j are connected and 0 if they are not. In this way, the planarity of n-point graph can be considered a Boolean function with $\binom{n}{2}$ variables. Now, this Boolean function is weakly

symmetric: for every two pairs, say, $\{i, j\}$ and $\{u, v\}$, there is a permutation of the vertices taking i into u and j into v. This permutation also induces a permutation on the set of point pairs that takes the first pair into the second one and does not change the planarity property.

A graph property is called *trivial* if either every graph has it or no one has it. A graph property is *monotone* if whenever a graph has it each of its subgraphs has it. For most graph properties that we investigate (connecivity, the existence of a Hamiltonian circuit, the existence of complete matching, colorability, etc.) either the property itself or its negation is monotonic.

The Aandera-Rosenberg-Karp conjecture, in its original form, concerns graph properties:

Conjecture 8.3.2. *Every non-trivial monotonic graph property is evasive, i.e., every decision tree that decides such a graph property and can only test whether two nodes are connected, has depth* $\binom{n}{2}$.

This conjecture is proved for a number of graph properties: for a general property, what is known is only that the tree has depth $\Omega(n^2)$ (Rivest and Vuillemin) and that the theorem is true if the number of points is a prime power (Kahn, Saks and Sturtevant). The analogous conjecture is also proved for bipartite graphs (Yao).

Exercise 8.3.2. *Prove that the connectedness of a graph is a evasive property.*

Exercise 8.3.3.

(a) Prove that if n is even then on n fixed points, the number of graphs not containing isolated points is odd.

(b) If n is even then the graph property that in an n-point graph there is no isolated point, is evasive.

(c) This statement holds also for odd n.*

Exercise 8.3.4. *A* tournament *is a complete graph each of whose edges is directed. Each tournament can be described by* $\binom{n}{2}$ *bits saying how the individual edges of the graph are directed. In this way, every property of tournaments can be considered an* $\binom{n}{2}$*-variable Boolean function. Prove that the tournament property that there is a 0-degree vertex is evasive.*

Among the more complex decision trees, the *algebraic decision trees* are important. In this case, the input is n real numbers $x_1, \ldots, x_n$ and every test function is described by a polynomial; in the internal nodes, we can go in three directions according to whether the value of the polynomial is negative, 0 or positive (sometime, we distinguish only two of these and the tree branches only in two). An example is provided for the use of such a decision tree by sorting, where the input can be considered n real numbers and the test functions are given by the polynomials $x_i - x_j$.

A less trivial example is the determination of the convex hull of n planar points. Recall that the input here is $2n$ real numbers (the coordinates of the points), and the test functions are represented either by the comparison of two coordinates or by the determination of the orientation of a triangle. The points $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3)

form a triangle with positive orientation if and only if

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0.$$

This can be considered therefore the determination of the sign of a second-degree polynomial. The algorithm described in Section 8.1 gives thus an algebraic decision tree in which the test functions are given by polynomials of degree at most two and whose depth is $O(n \log n)$.

The following theorem of Ben-Or provides a general lower bound on the depth of algebraic decision trees. Before the formulation of the theorem, we introduce an elementary topological notion. Let $U \subseteq \mathbf{R}^n$ be a set in the n-dimensional space. Two points x_1, x_2 of the set U are called *equivalent* if there is no decomposition $U = U_1 \cup U_2$ for which $x_i \in U_i$ and the closure of U_1 is disjoint from the closure of U_2. The equivalence classes of this equivalence relation are called the *components* of U. We call a set *connected* if it has only a single connected component.

Theorem 8.3.7 (Ben-Or)**.** *Suppose that the set $U \subseteq \mathbf{R}^n$ has at least N connected components. Then every algebraic decision tree deciding $x \in U$ whose test functions are polynomials of degree at most d, has depth at least $\log N / \log(6d) - n$. If $d = 1$ then the depth of every such decision tree is at least $\log_3 N$.*

Proof. We give the proof first for the case $d = 1$. Consider an algebraic decision tree of depth h. This has at most 3^h leaves. Consider a leaf reaching the conclusion $x \in U$. Let the results of the tests on the path leading here be, say,

$$f_1(x) = 0, \quad \ldots, \quad f_j(x) = 0, \quad f_{j+1}(x) > 0, \quad \ldots, \quad f_h(x) > 0.$$

Let us denote the set of solutions of this set of equations and inequalities by K. Then every input $x \in K$ leads to the same leaf and therefore we have $K \subseteq U$. Since every test function f_i is linear, the set K is convex and is therefore connected. So, K is contained in a single connected component of U. It follows that the inputs belonging to different components of U lead to different leaves of the tree. Therefore $N \leq 3^h$, which proves the statement referring to the case $d = 1$.

In the general case, the proof must be modified because K is not necessarily convex and so not necessarily connected either. Instead, we can use an important result from algebraic geometry (a theorem of Milnor and Thom) implying that the number of connected components of K is at most $(2d)^{n+h}$. From this, it follows similarly to the first part that

$$N \geq 3^h (2d)^{n+h} \geq (6d)^{n+h},$$

which implies the statement of the theorem. □

For an application, consider the following problem: given n real numbers $x_1, \ldots, x_n$; let us decide whether they are all different. We consider an elementary step the comparison of two given numbers, x_i and x_j. This can have three outcomes: $x_i < x_j$, $x_i = x_j$ and $x_i > x_j$. What is the decision tree with the smallest depth solving this problem?

It is very simple to give a decision tree of depth $n \log n$. Let us namely apply an arbitrary sorting algorithm to the given elements. If anytime during this, two compared elements are found to be equal then we can stop since we know the answer. If not then after $n \log n$ steps, we can order the elements completely, and thus they are all different.

Let us convince ourselves that $\Omega(n \log n)$ comparisons are indeed needed. Consider the following set:

$$U = \{\, (x_1, \ldots, x_n) : x_1, \ldots, x_n \text{ are all different } \}.$$

This set has exactly $n!$ connected components (two n-tuples belong to the same component if they are ordered in the same way). So, according to Theorem 8.3.7, every algebraic decision tree deciding $x \in U$ in which the test functions are linear, has depth at least $\log_3(n!) = \Omega(n \log n)$. The theorem also shows that we cannot gain an order of magnitude with respect to this even if we permitted quadratic or other bounded-degree polynomials as test polynomials.

We have seen that the convex hull of n planar points in general position can be determined by an algebraic decision tree of depth $n \log n$ in which the test polynomials have degree at most two. Since the problem of sorting can be reduced to the problem of determining the convex hull it follows that this is essentially optimal.

Exercise 8.3.5. *(a) If we allow a polynomial of degree n^2 as test function then a decision tree of depth 1 can be given to decide whether n numbers are different.*

(b) If we allow degree n polynomials as test functions then a depth n decision tree can be given to decide whether n numbers are different.

Exercise 8.3.6. *Given are $2n$ different real numbers: $x_1, \ldots, x_n, y_1, \ldots, y_n$. We want to decide whether it is true that after ordering them, there is a x_j between every pair of y_i's. Prove that this needs $\Omega(n \log n)$ comparisons.*

Chapter 9

Algebraic computations

Performing algebraic computations is a fundamental computational task, and its complexity theory is analogous to the complexity theory of Turing machine computations, but in some respects it is more complicated. We have already discussed some aspects of algebraic computations (power computation, Euclidean Algorithm, modulo m computations, Gaussian elimination) in Section 3.1.

9.1 Models of algebraic computation

In the *algebraic model* of computation the input is a sequence of numbers $(x_1, \ldots, x_n)$, and during our computation, we can perform algebraic operations (addition, subtraction, multiplication, division). The output is an algebraic expression of the input variables, or perhaps several such expressions. The numbers can be from any field, but we usually use the field of the reals in this section. Unlike e.g., in Section 1.3, we do not worry about the bit-size of these numbers, not even whether they can be described in a finite way (except in Section 9.2.1 and at the end of Section 9.2.5, where we deal with multiplication of very large integer numbers).

To be more precise, an *algebraic computation* is a finite sequence of instructions, where the k-th instruction is one of the following:

(A1) $R_k = x_j$ $(1 \leq j \leq n)$ (reading an input),

(A2) $R_k = c$ $(c \in \mathbb{R})$ (assigning a constant),

(A3) $R_k = R_i \star R_j$ $(1 \leq i, j < k)$ (arithmetic operations)

(here $\star$ is any of the operations of addition, subtraction, multiplication or division). The *length* of this computation is the number of instructions of type (A2) and (A3). We must make sure that none of the expressions we divide with is identically 0, and that the result of the algebraic computation is correct whenever we do not divide by zero.

We sometimes refer to the values R_i as the *partial results* of the computation. The *result* of the computation is a subsequence of the partial results: $(R_{i_1}, \ldots, R_{i_k})$. Often this is just one value, in which case we may assume that this is the last partial result (whatever comes after is superfluous).

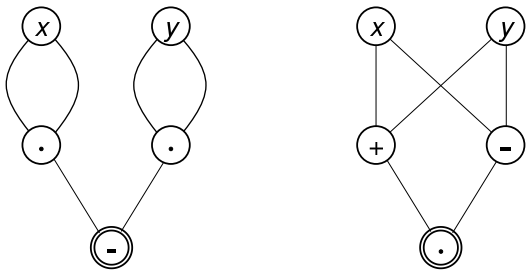

Figure 9.1: Algebraic circuits representing computations (9.1) and (9.2).

As an example, the expression $x^2 - y^2$ can be evaluated using three operations:

$$R_1 = x; \quad R_2 = y; \quad R_3 = R_1 \cdot R_1; \quad R_4 = R_2 \cdot R_2; \quad R_5 = R_3 - R_4. \tag{9.1}$$

An alternate evaluation, using a familiar identity, is the following:

$$R_1 = x; \quad R_2 = y; \quad R_3 = R_1 + R_2; \quad R_4 = R_1 - R_2; \quad R_5 = R_3 \cdot R_4. \tag{9.2}$$

Sometimes we want to distinguish multiplying by a constant from multiplying two expressions; in other words, we consider as a separate operation

(A4) $R_k = cR_i$ ($c \in \mathbb{R}$, $1 \leq i < k$) (multiplying by a constant),

even though it can be obtained as an operation of type (A2) followed by an operation of the type (A3).

Often, one needs to consider the fact that not all operations are equally costly: we do not count (A1) (reading the input); furthermore, multiplying by a constant, adding or subtracting two variables (called *linear operations*) are typically cheaper than multiplication or division (called *nonlinear operations*). For example, (9.1) and (9.2) use the same number of operations, but the latter computation uses fewer multiplications. We will see that counting non-linear operations is often a better measure of the complexity of an algorithm, both in the design of algorithms and in proving lower bounds on the number of steps.

An algebraic computation can also be described by a circuit. An *algebraic circuit* is a directed graph that does not contain any directed cycle (i.e., it is *acyclic*). The sources (the nodes without incoming edges) are called *input nodes*. We assign a variable or a constant to each input node. The sinks (the nodes without outgoing edges) are called *output nodes*. (In what follows, we will deal most frequently with circuits that have a single output node.) Each node v of the graph that is not a source has indegree 2, and it is labeled with one of the operation symbols $+, -, \cdot, /$, and it performs the corresponding operation on the two incoming numbers (whose order is specified, so that we know which of the two is, e.g., the dividend and the divisor). See Figure 9.1.

Every algebraic computation translates into an algorithm on the RAM machine (or Turing machine), if the input numbers are rational. However, the cost of this computation is then larger, since we have to count the bit-operations, and the number of bits in the input can be large, and the partial results can be even larger. If we want to make sure that such a computation takes polynomial time, we must make sure that the underlying algebraic computation has polynomial length (as a function of the number of input numbers), and also that the number of bits in every partial result is bounded by a polynomial in the number of bits in the input.

9.2 Multiplication

We start with discussing the operation of multiplication of numbers, matrices and polynomials (and also the related topic of inverting matrices). Multiplication of numbers is trivial as an algebraic computation (it is a single step in this model), so we discuss its bit-complexity; while this is a diversion, the tricks and methods applicable here are in many ways analogous to the algebraic computation algorithms for the multiplication of matrices and polynomials.

9.2.1 Arithmetic operations on large numbers

Suppose that we want to do arithmetic operations with really large integers, with thousands, perhaps even millions of bits, and we want to do it more efficiently than the method we learn in elementary school. (Forty years ago this would have been a theoretical exercise; now, with cryptography using such numbers, it has become a very important issue.) In our analysis, we count only operations between the input bits of the two numbers, and ignore the additional work involved in organizing the computation (shifting around the numbers etc.). This extra cost would not change the order of magnitude of the work to be performed, but would make the analysis machine-dependent.

To be more exact, suppose that the numbers are given in their binary representations:

$$u = \overline{u_{n-1} \dots u_1 u_0} = u_0 + 2u_1 + 2^2 u_2 + \dots, +2^{n-1} u_{n-1}$$
$$v = \overline{v_{n-1} \dots v_1 v_0} = v_0 + 2v_1 + 2^2 v_2 + \dots, +2^{n-1} v_{n-1}$$

(for simplicity, we assume they have the same number of bits). We want the result in binary form too, e.g., for multiplication:

$$w = uv = \overline{w_{2n-1} \dots w_1 w_0} = w_0 + 2w_1 + 2^2 w_2 + \dots + 2^{2n-1} w_{2n-1}$$

For addition and subtraction, we cannot gain any substantial time: the method we learn in school takes $4n$ bit-operations, and just to read the numbers takes $2n$ time. But, somewhat surprisingly, the situation is quite different for multiplication. Using the traditional method for multiplication, this takes about n^2 bit operations (every bit of u must be multiplied with every bit of v; while these multiplications are trivial in binary, we have to write down about n^2 bits).

One would think that there is nothing better, but in fact we can save some work. We start with a simple idea. Suppose that $n = 2m$ is even, then we can write

$$u = 2^m U_1 + U_0, \qquad v = 2^m V_1 + V_0,$$

where $U_1 = \overline{u_{2m-1} \dots u_m}$, $U_0 = \overline{u_{m-1} \dots u_0}$, and similarly $V_1 = \overline{v_{2m-1} \dots v_m}$ and $V_0 = \overline{v_{m-1} \dots v_0}$. Then

$$uv = 2^{2m} U_1 V_1 + 2^m (U_0 V_1 + U_1 V_0) + U_0 V_0.$$

We could try to use this formula to compute the product recursively; but it seems that to evaluate this expression, even if we ignore additions, we have to perform four multiplications on m-bit integers, and it is easy to see that this does not lead to any gain in the number of bit-operations. The key observation is that with a little algebra, three multiplications are enough: Since

$$U_0V_1 + U_1V_0 = (U_1 - U_0)(V_0 - V_1) + U_0V_0 + U_1V_1,$$

we can express the product as

$$(2^{2m} + 2^m)U_1V_1 + 2^m(U_1 - U_0)(V_0 - V_1) + (2^m + 1)U_0V_0.$$

This way we have reduced the multiplication of two $2m$-bit integers to three multiplications of m-bit integers, and a few additions and multiplications by powers of 2. It is easy to count these additional operations, to get that they involve at most $22m$ bit operations.

If we denote by $T(n)$ the number of operations used by this recursive algorithm, then

$$T(2m) \leq 3T(m) + 22m.$$

This formula gives, by induction, an upper bound on the number of bit-operations, at least if the number of bits is a power of 2:

$$\begin{aligned} T(2^k) &\leq 3T(2^{k-1}) + 22 \cdot 2^{k-1} \leq \cdots \leq 3^k + 22(2^{k-1} + 3 \cdot 2^{k-2} + \cdots + 3^{k-1}) \\ &= 3^k + 22(3^k - 2^k) < 23 \cdot 3^k. \end{aligned}$$

To multiply two integers with n bits for a general n, we can "pad" the numbers with leading 0-s to increase the number of their bits to the next power of 2. If $k = \lceil \log n \rceil$, then this algorithm will compute the product of two n-bit integers using

$$T(n) \leq T(2^k) < 23 \cdot 3^k < 23 \cdot 3^{1+\log n} = 69 \cdot n^{\log 3} < 69 \cdot n^{1.585}$$

bit-operations.

This simple method to compute the product of two large integers more efficiently than the method learned in elementary school can be improved substantially. We will see that using more advanced methods (discrete Fourier transforms) much less bit-operations suffice.

9.2.2 Matrix multiplication

Assume that we want to multiply the matrices

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$

(for simplicity, we consider only the case when the matrices are square). The product matrix is

$$C = \begin{pmatrix} c_{11} & \dots & c_{1n} \\ \vdots & & \vdots \\ c_{n1} & \dots & c_{nn} \end{pmatrix} \quad \text{where} \quad c_{ij} = a_{i1}b_{1j} + \cdots + a_{in}b_{nj}. \tag{9.3}$$

Performing these arithmetic operations in the simple way takes n^3 multiplications and $n^2(n-1) \sim n^3$ additions. Normally, we think of multiplications as more costly than additions, so it could be useful to reduce the number of multiplications, even if it would mean to increase the number of additions. (However, we will see that, surprisingly, we can also reduce the number of additions.)

Strassen noticed that the multiplication of 2×2 matrices can be carried out using 7 multiplications and 18 additions, instead of the 8 multiplications and 4 additions in (9.3). We form 7 products:

$$\begin{aligned} u_0 &= (a_{11} + a_{22})(b_{11} + b_{22}), \\ u_1 &= (a_{21} + a_{22})b_{11}, \\ u_2 &= a_{11}(b_{12} - b_{22}), \\ u_3 &= a_{22}(b_{21} - b_{11}), \\ u_4 &= (a_{11} + a_{12})b_{22}, \\ u_5 &= (a_{21} - a_{11})(b_{11} + b_{12}), \\ u_6 &= (a_{12} - a_{22})(b_{21} + b_{22}). \end{aligned} \tag{9.4}$$

Then we can express the entries of the product matrix as follows:

$$\begin{aligned} c_{11} &= u_0 + u_3 - u_4 + u_6, \\ c_{21} &= u_1 + u_3, \\ c_{12} &= u_2 + u_4, \\ c_{22} &= u_0 + u_2 - u_1 + u_5. \end{aligned} \tag{9.5}$$

To have to perform 14 extra additions to save one multiplication does not look like a lot of gain, but we see how this gain is realized once we extend the method to larger matrices. Similarly as for multiplication of integers, we show how to reduce the multiplication of $(2n) \times (2n)$ matrices to the multiplication of $n \times n$ matrices. Let A, B and $C = AB$ be $(2n) \times (2n)$ matrices, and let us split each of them into four $n \times n$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Then $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$, and we can use the formulas (9.4) and (9.5) to compute these four matrices using only 7 multiplications and 18 additions of $n \times n$ matrices. (Luckily, the verification of the formulas (9.4) and (9.5), which we did not write down, does not use commutativity of multiplication, so it remains valid for matrices.) Assuming that we start with a $2^k \times 2^k$ matrix, we can do this splitting recursively, until

we get down to 1×1 matrices (which can be multiplied using a single multiplication of numbers). If the number of rows and columns is not a power of 2, we start with adding all-0 rows and columns to increase the size to the nearest power of 2.

Do we save any work by this more complicated algorithm? Let $M(n)$ denote the number of multiplications, and $S(n)$ the number of additions, when this algorithm is applied to $n \times n$ matrices. Then

$$M(2n) = 7M(n) \quad \text{and} \quad S(2n) = 18n^2 + 7S(n).$$

Clearly $M(1) = 1$, $S(1) = 0$, and it is easy to prove by induction on k that

$$M(2^k) = 7^k \quad \text{and} \quad S(2^k) = 6(7^k - 4^k).$$

Let $k = \lceil \log n \rceil$, then

$$M(n) = M(2^k) = 7^k < 7^{1+\log n} = 7n^{\log 7} < 7n^{2.81},$$

and similarly

$$S(n) < 42n^{\log 7} < 42n^{2.81}.$$

We see that while for $n = 2$ Strassen's method only gained a little in the number of multiplications (and lost a lot in the number of additions), through this iteration we improved both the number of multiplications and the number of additions, at least for large matrices.

It is not easy to explain where the formulas (9.4) and (9.5) come from; in a sense, this is not even understood today, since it is open how much the exponent of n in the complexity of matrix multiplication can be reduced by similar methods. The current best algorithm, due to Williams, uses $O(n^{2.3727})$ multiplications and additions.

9.2.3 Inverting matrices

Let B be a $(2n) \times (2n)$ matrix, which we partition into 4 parts:

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{9.6}$$

We can bring B to a block-diagonal form similarly as we would do for a 2×2 matrix:

$$\begin{pmatrix} I & 0 \\ -B_{21}B_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} I & -B_{11}^{-1}B_{12} \\ 0 & I \end{pmatrix} = \begin{pmatrix} B_{11} & 0 \\ 0 & B_{22} - B_{21}B_{11}^{-1}B_{12} \end{pmatrix}. \tag{9.7}$$

To simplify notation, let $C = B_{22} - B_{21}B_{11}^{-1}B_{12}$. Inverting and expressing B^{-1}, we get

$$\begin{aligned} B^{-1} &= \begin{pmatrix} I & -B_{11}^{-1}B_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} B_{11}^{-1} & 0 \\ 0 & C^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -B_{21}B_{11}^{-1} & I \end{pmatrix} \\ &= \begin{pmatrix} B_{11}^{-1} + B_{11}^{-1}B_{12}C^{-1}B_{21}B_{11}^{-1} & -B_{11}^{-1}B_{12}C^{-1} \\ -C^{-1}B_{21}B_{11}^{-1} & C^{-1} \end{pmatrix} \end{aligned} \tag{9.8}$$

This is a messy formula, but it describes how to compute the inverse of a $(2n) \times (2n)$ matrix using two matrix inversions (for B_{11} and C), 6 matrix multiplications and 2 additions (one of which is in fact a subtraction), all performed on $n \times n$ matrices. We could use this recursively as before, but there is a problem: how do we know that B_{11} is invertible? This does not follow even if we assume that B is invertible.

The way out is to use the identity

$$A^{-1} = (A^\top A)^{-1} A^\top. \tag{9.9}$$

This shows that if we can invert the matrix $B = A^\top A$, then, at the cost of a further matrix multiplication, we can compute the inverse of A. (We do not count the cost of computing the transpose of A, which involves only moving numbers around, no algebraic operations.) Now if A is nonsingular, then B is symmetric and positive definite. Hence the principal submatrix B_{11} in the decomposition (9.6) is also symmetric and positive definite. Furthermore, identity (9.7) implies that C is also symmetric and positive definite.

These facts have three important consequences. First, it follows that B_{11} and C are nonsingular, so the inverses B_{11}^{-1} and C^{-1} in (9.8) make sense. Second, it follows that when computing B_{11}^{-1} and C^{-1} recursively, then we stay in the territory of inverting symmetric and positive definite matrices, and so we don't have to appeal to the trick (9.9) any more. Third, it follows that $B_{21} = B_{12}^\top$, which saves us two multiplications, since $B_{21}B_{11}^{-1} = (B_{11}^{-1}B_{12})^\top$ and $C^{-1}B_{21}B_{11}^{-1} = (B_{11}^{-1}B_{12}C^{-1})^\top$ do not need to be computed separately.

Let $I^+(n)$ denote the minimum number of multiplications needed to invert an $n \times n$ positive definite matrix, and let $L(n)$ denote the minimum number of multiplications needed to compute the product of two $n \times n$ matrices. It follows from formula (9.8) that

$$I^+(2n) \le 2I^+(n) + 4L(n).$$

Using the matrix multiplication algorithm given in Section 9.2.2, we get that

$$I^+(2^{k+1}) \le 2I^+(2^k) + 4 \cdot 7^k,$$

which implies by induction that

$$I^+(2^k) \le 7^k.$$

Using (9.9), we get that a nonsingular $2^k \times 2^k$ matrix can be inverted using $3 \cdot 7^k$ multiplications. Just as in Section 9.2.2, this implies a bound for general n: an $n \times n$ matrix can be inverted using no more than $21 \cdot n^{\log 7}$ multiplications. The number of additions can be bounded similarly.

9.2.4 Multiplication of polynomials

Suppose that we want to compute the product of two real polynomials in one variable, of degree n. Given

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n, \quad \text{and} \quad Q(x) = b_0 + b_1 x + \cdots + b_n x^n,$$

we want to compute their product

$$R(x) = P(x)Q(x) = c_0 + c_1 x + \cdots + c_{2n} x^{2n}.$$

The coefficients of this polynomial can be computed by the formulas

$$c_i = a_0 b_i + a_1 b_{i-1} + \cdots + a_i b_0. \tag{9.10}$$

This is often called the *convolution* of the sequences $(a_0, a_1, \ldots, a_n)$ and $(b_0, b_1, \ldots, b_n)$. To compute every coefficient by these formulas takes $(n+1)^2$ multiplications.

We can do better if we use the fact that we can substitute into a polynomial. Let us substitute the values $0, 1, \ldots, 2n$ into the polynomials. In other words, we compute the values $P(0), P(1), \ldots, P(2n)$ and $Q(0), Q(1), \ldots, Q(2n)$, and then compute their products $R(j) = P(j)Q(j)$. From here, the coefficients of R can be determined by solving the equations

$$\begin{aligned} c_0 &= R(0) \\ c_0 + c_1 + c_2 + \cdots + c_{2n} &= R(1) \\ c_0 + 2c_1 + 2^2 c_2 + \cdots + 2^{2n} c_{2n} &= R(2) \\ &\vdots \\ c_0 + (2n)c_1 + (2n)^2 c_2 + \cdots + (2n)^{2n} c_{2n} &= R(2n) \end{aligned} \tag{9.11}$$

This does not seem to be a great idea, since we need about n^2 multiplications (and about the same number of additions) to compute the values $P(0), P(1), \ldots, P(2n)$ and $Q(0), Q(1), \ldots, Q(2n)$; it takes a small number of multiplications to get the values $R(0), R(1), \ldots, R(2n)$, but then of the order of n^3 multiplications and additions to solve the system (9.11) if we use Gaussian elimination (fewer, if we use the more sophisticated methods discussed in Section 9.2.2, but still substantially more than n^2). We see some gain, however, if we distinguish two kinds of multiplications: multiplication by a fixed constant, or multiplication of two expressions containing the parameters (the coefficients a_i and b_i). Recall, that additions and multiplications by a fixed constant are *linear operations*. The computation of the values $P(0), P(1), \ldots, P(2n)$ and $Q(0), Q(1), \ldots, Q(2n)$, as well as the solution of equations (9.11), takes only linear operations. Nonlinear operations are needed only in the computation of the $R(j)$, so their number is only $2n + 1$.

It would be very useful to reduce the number of linear operations too. The most time-consuming part of the above is solving equations (9.11); it takes of the order of n^3 operations if we use Gaussian elimination (these are all linear, but still a lot). Using more of the special structure of the equations, this can be reduced to $O(n^2)$ operations. But we can do even better, if we notice that there is nothing special about the substitutions $0, 1, \ldots, 2n$, we could use any other $2n + 1$ real or even complex numbers. As we are going to discuss in Section 9.2.5, substituting appropriate roots of unity leads to a much more efficient method for the multiplication of polynomials.

9.2.5 Discrete Fourier transform

Let $P(x) = a_0 + a_1 x + \cdots + a_n x^n$ be a real polynomial, and fix any $r > n$. Let $\varepsilon = e^{2\pi i/r}$ be the first r-th root of unity, and consider the values

$$\hat{a}_k = P(\varepsilon^k) = a_0 + a_1 \varepsilon^k + a_2 \varepsilon^{2k} + \cdots + a_n \varepsilon^{nk} \qquad (k = 0, \ldots, r-1). \tag{9.12}$$

The sequence $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{r-1})$ is called the *discrete Fourier transform of order* r of the sequence of coefficients $(a_0, a_1, \ldots, a_n)$. We will often append $r - n - 1$ zeros to this sequence, to get a sequence $(a_0, a_1, \ldots, a_{r-1})$ of length r.

Discrete Fourier transforms have a number of very nice properties and important applications, of which we only discuss those related to polynomial multiplication.

We start with some simple but basic properties. First, the inverse transformation can be described by similar formulas:

$$a_k = \frac{1}{r}\big(\hat{a}_0 + \hat{a}_1\varepsilon^{-k} + \hat{a}_2\varepsilon^{-2k} + \cdots + \hat{a}_{r-1}\varepsilon^{-(r-1)k}\big) \qquad (k = 0, \ldots, r-1). \quad (9.13)$$

This can be verified by substituting the definition of $\hat{a}_k$ into these formulas. Second, assume that $r > 2n$, and let $(b_0, \ldots, b_{r-1})$ and $(c_0, \ldots, c_{r-1})$ be the coefficient sequences of the polynomials $Q(x)$ and $R(x) = P(x)Q(x)$, and let $(\hat{b}_0, \ldots, \hat{b}_{r-1})$ and $(\hat{c}_0, \ldots, \hat{c}_{r-1})$ be their Fourier transforms of order r. Since $\hat{a}_k$ is a the value of P at ε^k, we get that

$$\hat{c}_k = \hat{a}_k\hat{b}_k. \quad (9.14)$$

The main point in using discrete Fourier transforms is that they can be computed very fast; this method is one of the most successful algorithmic tools in computations. To describe a fast method for computing the discrete Fourier transform, suppose that $r = 2s$ is even. The Fourier transform (of order r) of a sequence $(a_0, a_1, \ldots, a_{r-1})$ can be split into two parts:

$$\begin{aligned}\hat{a}_k &= a_0 + a_1\varepsilon^k + \cdots + a_{r-1}\varepsilon^{(r-1)k} \\ &= (a_0 + a_2\varepsilon^{2k} + \cdots + a_{2s-2}\varepsilon^{(2s-2)k}) \\ &\quad + \varepsilon^k(a_1 + a_3\varepsilon^{2k} + \cdots + a_{2s-1}\varepsilon^{(2s-2)k}). \end{aligned} \quad (9.15)$$

Both expressions in parenthesis are Fourier transforms themselves: since ε^2 is the first s-th root of unity, they are Fourier transforms of order s of the two sequences $(a_0, a_2, \ldots, a_{2s-2})$ and $(a_1, a_3, \ldots, a_{2s-1})$. So we have reduced the computation of a Fourier transform of order $r = 2s$ to the computation of two Fourier transforms of order s. We can do this recursively.

How much work is involved? Let $K(r)$ denote the number of arithmetic operations this algorithm uses to perform a Fourier transform of order $r = 2s$. Recursively, we need $2K(s)$ operations to compute the two smaller Fourier transforms. We need $2s - 2$ multiplications to compute the powers of ε. Once we have these powers, we need only two further arithmetic operations to apply (9.15), but we have to do so for every k, so we need $4s$ operations. Putting these together, we get

$$K(2s) \le 2K(s) + 6s.$$

If $s = 2^m$ is a power of 2, then this inequality implies, by induction, that

$$K(2^m) \le 3m \cdot 2^m.$$

For a general r, we can choose $m = \lceil \log r \rceil$, and get

$$K(r) \le K(2^m) \le 3(1 + \log r)2^{1+\log r} = 6(1 + \log r)r.$$

This is a much better bound than the $O(r^2)$ operations we get from the definition.

As an application, let us return to multiplying two polynomials P and Q of degree n. This can be done by computing Fourier transforms of order $r = 2n + 2$ (this takes $O(n \log n)$ arithmetic operations), then computing the product of the values $P(\varepsilon^k)$ and $Q(\varepsilon^k)$ (this takes $O(n)$ further operations), and then computing the inverse Fourier transform by essentially the same method as for the "forward" Fourier transform (this takes $O(n \log n)$ operations). So the product of two polynomials of degree n can be computed using $O(n \log n)$ arithmetic operations.

As another application of discrete Fourier transforms, we show that two n-bit numbers can be multiplied using $O(n \log^3 n)$ bit-operations. (This bound can be improved to $O(n \log n \log \log n)$ bit-operations with more care.) We can use the multiplication of polynomials. Let $u = \overline{u_{n-1} \dots u_1 u_0}$ and $v = \overline{v_{n-1} \dots v_1 v_0}$ be two positive integers, given in binary. Consider the polynomials

$$U(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_{n-1} x^{n-1} \quad \text{and} \quad V(x) = v_0 + v_1 x + v_2 x^2 + \cdots + v_{n-1} x^{n-1}$$

Then $u = U(2)$ and $v = V(2)$, and hence $uv = U(2)V(2)$. As we have seen, the product polynomial UV can be computed using $O(n \log n)$ arithmetic operations. The substitution of 2 can be computed using $O(n)$ further arithmetic operations.

However, here we are counting not arithmetic operations, but bit-operations. The integers that we compute (the $2n + 1$ coefficients of the product polynomial) are not larger than n, and so the substitution of 2 into the product UV (computing the result in binary) takes no more than $O(n \log n)$ bit-operations.

One has to be more careful with computing the Fourier transform and its inverse, since the roots of unity are complex irrational numbers, which have to be computed with some finite but suitably large precision. Computing with complex numbers just means computing with twice as many real numbers, and arithmetic operations between complex numbers just mean two or six arithmetic operations between real numbers, so using complex numbers does not cause any trouble here. But we have to address the issue of precision.

If we compute the binary representations of the real and complex parts of the roots of unity up to $6 \log n$ bits, then the error we make is less than n^{-6}. Even when multiplied by an integer not larger than $O(n)$, the error is still only $O(n^{-5})$, and if $O(n)$ such terms are added up, the error remains only $O(n^{-4})$. Thus we get the values of $U(\varepsilon^k)$ and $V(\varepsilon^k)$ with error $O(n^{-4})$, and since $|U(\varepsilon^k)| \leq n$ and $|V(\varepsilon^k)| \leq n$, it follows that we get the product $U(\varepsilon^k)V(\varepsilon^k)$ with error $O(n^{-3})$. Applying inverse Fourier transformation to these values, we get the result with error $O(n^{-1})$. Rounding these values to the nearest integer, we get the value uv. All numbers involved in these computations have $O(\log n)$ bits, so an arithmetic operation between them takes $O(\log^2 n)$ bit-operations. This means a total of $O(n \log^3 n)$ bit-operations.

9.3 Algebraic complexity theory

We return to the notations introduced in Section 9.1.

9.3.1 The complexity of computing square-sums

As a warm-up, we prove the following lower bound:

Theorem 9.3.1. *Every algebraic computation for $x_1^2 + \cdots + x_n^2$ from inputs $x_1, \ldots, x_n$ must contain at least n nonlinear operations.*

If you feel that this must be trivial, recall that the expression $x_1^2 - x_2^2$ can be computed using a single multiplication, from the formula $x_1^2 - x_2^2 = (x_1 - x_2)(x_1 + x_2)$.

Proof. The main idea is to replace nonlinear (A3) operations (multiplications and divisions) by operations of type (A4) (multiplication with a constant) and show that the output is still correct for several inputs, which will lead to a contradiction.

Suppose that there is an algebraic computation for $x_1^2 + \cdots + x_n^2$, having $m < n$ nonlinear operations, which we label $1, 2, \ldots, m$. We fix some real numbers, $a_1, \ldots, a_n$, such that if we input $x_i = a_i$, then there are no divisions by zero in the circuit. We denote the value of R_j by u_j for this input. Then we replace (in acyclic order) every operation of the form $R_k = R_i \cdot R_j$ by $R_k = u_j \cdot R_i$ (operation of type (A4)) and write down the equation $R_j = u_j$ where R_j is a linear function of the input variables x_i, as all earlier nonlinear operations have been already replaced by linear ones. We similarly replace operations of the form $R_k = R_i/R_j$ by $R_k = (1/u_j) \cdot R_i$ and write down the equation $R_j = u_j$.

The m equations that we wrote down can be viewed as a system of linear equations for the unknowns $x_1, \ldots, x_n$. This system is solvable, since the values $x_i = a_i$ satisfy it. Furthermore, the number of equations is m, which is less than the number n of unknowns, and hence the system has an at least one-dimensional family of solutions. A one-dimensional subspace of the solutions can be represented as $x_i = a_i + tb_i$, where the b_i are real numbers, not all zero, and t ranges over all reals.

Let us substitute $x_i = a_i + tb_i$. The $R_j = u_j$ equations remain valid (for any t), thus the circuit makes the same computations as the original one. The output will be also the same, *linear* expression in $x_1, \ldots, x_n$. This means that

$$(a_1 + tb_1)^2 + \cdots + (a_n + tb_n)^2$$

is a linear function of t. But this is a contradiction, since the coefficient of t^2 is $b_1^2 + \cdots + b_n^2$, which is positive. □

9.3.2 Evaluation of polynomials

Let $f(\mathbf{x}, y) = x_1 y + \cdots + x_n y^n$ be a polynomial in one variable y. We denote the coefficients by $x_1, \ldots, x_n$, to emphasize that they are not fixed, but real parameters. We want to evaluate f at a given value $y = u$ with some $\mathbf{x} = \mathbf{a}$. The following familiar identity, called the *Horner scheme* achieves this with just n multiplications and $n - 1$ additions:

$$f(\mathbf{a}, u) = (\ldots((a_n u + x_{n-1})u + x_{n-2})u \cdots + x_1)u.$$

Of course, there are many concrete polynomials that can be evaluated faster. For example, the value of

$$y^n + \binom{n}{n-1} y^{n-1} + \cdots + ny + 1 = (y+1)^n$$

can be computed using a single addition and $O(\log n)$ multiplications. However, in the general case, the Horner scheme is best possible:

Theorem 9.3.2. *Every algebraic computation computing $f(\mathbf{x}, y)$ from inputs $\mathbf{x} = (x_1, \ldots, x_n)$ and y must contain at least n nonlinear operations.*

Proof. The proof is based on the same idea as Theorem 9.3.1, but we have to do it differently. The nonlinear operations will be replaced by a sum of two quantities, the first of which will be linear in $\mathbf{x}$ and the second a a rational function of y with real coefficients.

Suppose that there is an algebraic computation involving m nonlinear operations, where $m < n$. Going through the computation in the order in which it is given, we modify the nonlinear operations as follows.

The output of the k-th operation will be replaced by an expression of the special form

$$R_k = \mathbf{u}_k \cdot \mathbf{x} + d_k(y), \tag{9.16}$$

where $\mathbf{u}_k \in \mathbb{R}^n$, and $d_k(y)$ is a rational function of y with real coefficients (so it is linear in $x_1, \ldots, x_n$, but not necessarily in y). Furthermore, for some of these operations (not necessarily for all of them) we write down a dependence relation between the input numbers of the form

$$\mathbf{v}_k \cdot \mathbf{x} = h_k(y), \tag{9.17}$$

where again $\mathbf{v}_k \in \mathbb{R}^n$, and $h_k(y)$ is a rational function of y with real coefficients.

The operations $R_k = x_i$ and $R_k = y$ are already in the above form, while in case of $R_k = R_i \pm R_j$ the equations $\mathbf{u}_k = \mathbf{u}_i \pm \mathbf{u}_j$ and $d_k(y) = d_i(y) \pm d_j(y)$ are suitable choices, while for $R_k = cR_i$ the equations $\mathbf{u}_k = c\mathbf{u}_i$ and $d_k(y) = cd_i(y)$ are good. For these we do not have to write down a dependence relation 9.17.

To describe how this replacement is done, let us start with the most difficult case, when the k-th step is the division $R_k = R_i/R_j$. We have some subcases to consider.

(1) Suppose that $\mathbf{u}_k$ (in the output R_k of the k-th step) is linearly independent of the vectors $\mathbf{v}_r$ ($r < k$) of the previous conditions (9.17). We add a new condition to the list (9.17):

$$\mathbf{u}_j \cdot \mathbf{x} + d_j(y) = 1,$$

(in other words, we take $\mathbf{v}_k = \mathbf{u}_j$ and $h_k(y) = 1 - d_j(y)$). The new output of the operation will be R_i (in other words, $\mathbf{u}_k = \mathbf{u}_i$ and $d_k(y) = d_i(y)$).

(2) Suppose that $\mathbf{u}_j$ is a linear combination of the vectors v_r ($r < k$):

$$\mathbf{u}_j = \sum_{r=1}^{k-1} \alpha_r \mathbf{v}_r,$$

but $\mathbf{u}_i$ is not. Define the rational function

$$\overline{d_j}(y) = d_j(y) + \sum_{r=1}^{k-1} \alpha_r h_r(y)$$

(where $\alpha_r = 0$ if we did not write down a new equation at the computation of R_r). Add a new condition to the list (9.17):

$$\mathbf{u}_i \cdot x + d_i(y) = 1,$$

and let the new output of the operation be $1/\overline{d_j}(y)$. (So $\mathbf{u}_k = \mathbf{0}$ in this case.)

(3) Finally, suppose that both $\mathbf{u}_i$ and $\mathbf{u}_k$ are a linear combinations of the vectors $\mathbf{v}_r$ $(r < k)$:

$$\mathbf{u}_j = \sum_{r=1}^{k-1} \alpha_r \mathbf{v}_r, \qquad \mathbf{u}_i = \sum_{r=1}^{k-1} \beta_r \mathbf{v}_r.$$

Define the rational functions

$$\overline{d_j}(y) = d_j(y) + \sum_{r=1}^{k-1} \alpha_r h_r(y), \quad \overline{d_i}(y) = d_i(y) + \sum_{r=1}^{k-1} \beta_r h_r(y),$$

and let the new output be $\overline{d}_i(y)/\overline{d}_j(y)$. There is no new condition of the type (9.17) to record.

Multiplication steps are modified similarly. From the construction the following statement follows directly.

Claim 9.3.3. *The vectors $\mathbf{v}_k$ in the conditions* (9.17) *are linearly independent.*

Claim 9.3.4. *For every input satisfying conditions* (9.17)*, the modified algorithm computes the same values as the original at each step.*

Proof. The proof is by induction on the index of the step. Suppose that the Claim holds for the first $k-1$ outputs. If the k-th step is an addition, subtraction or multiplication by a constant, then the Claim holds for the k-th output trivially. If the k-th step is a division, and (1) holds, then

$$\frac{\mathbf{u}_i \cdot \mathbf{x} + d_i(y)}{\mathbf{u}_j \cdot \mathbf{x} + d_j(y)} = \mathbf{u}_i \cdot \mathbf{x} + d_i(y) = \mathbf{u}_k \cdot \mathbf{x} + d_k(y).$$

If (2) applies, then

$$\overline{d}_j(y) = d_j(y) + \sum_{r=1}^{k-1} \alpha_r h_r(y) = \sum_{r=1}^{k-1} \alpha_r (\mathbf{v}_r \cdot \mathbf{x}) = \mathbf{u}_j \cdot \mathbf{x} + d_j(y),$$

and so

$$\frac{\mathbf{u}_i \cdot \mathbf{x} + d_i(y)}{\mathbf{u}_j \cdot \mathbf{x} + d_j(y)} = \frac{1}{\mathbf{u}_j \cdot \mathbf{x} + d_j(y)} = \frac{1}{\overline{d}_j(y)} = d_k(y).$$

In case (3), and in the case of multiplication, the argument is similar. □

In particular, if we apply this computation with any input that satisfies the conditions (9.17), then the modified computation will have the same output as the original. Fixing an arbitrary value of y, the conditions (9.17) give a system of linear equations on the coefficients x_i. The number of these equations is at most $m < n$. Since the left hand sides are linearly independent, this system has an infinite number of solutions; in fact, it has a one-parameter linear family $x_i = a_i(y) + tb_i$ of solutions. Here the $a_i(y)$ are rational functions of y and the b_i can be obtained as solutions of the homogeneous system of equations $\mathbf{c} \cdot \mathbf{x} = 0$, and therefore they are independent of y. Not all the b_i are 0.

For every t, the original algorithm computes on input $x_i = a_i(y) + tb_i$ and y the value

$$\sum_{i=1}^{n}(a_i(y) + tb_i)y^i = \sum_{i=1}^{n} a_i(y)y^i + t\sum_{i=1}^{n} b_i y^i.$$

On the other hand, these inputs satisfy the conditions (9.17), and so the output is

$$\mathbf{u} \cdot (\mathbf{a}(y) + t\mathbf{b}) + d(y) = \mathbf{u} \cdot \mathbf{a}(y) + t\mathbf{u} \cdot \mathbf{b} + d(y)$$

for every t. So the coefficient of t must be the same in both expressions: $\sum_{i=1}^{n} b_i y^i = \mathbf{u} \cdot \mathbf{b}$. But since $\mathbf{b}$ and $\mathbf{u}$ are independent of y, and not all b_i are 0, this cannot hold for every y. □

9.3.3 Formula complexity and circuit complexity

Let $P(x_1, \ldots, x_n)$ be a polynomial with real coefficients. Most often, such a polynomial is given by a *formula*, using addition, subtraction and multiplication. A formula can be defined as an algebraic circuit (without divisions) where every outdegree is at most 1.

The *complexity* of a formula is the number of arithmetic operations in it. Different formulas representing the same polynomials can have very different complexity. For example, consider the expansion

$$(x_1 + y_1)(x_2 + y_2)\ldots(x_n + y_n) = x_1x_2\ldots x_n + y_1x_2\ldots x_n + \cdots + y_1y_2\ldots y_n.$$

Both sides are formulas representing the same polynomial, but the left side has complexity $2n - 1$, while the right side has complexity $(n - 1)2^n$. We define the *formula complexity* of a polynomial as the minimum complexity in any formula representing the polynomial. We define the *circuit complexity* of a polynomial as the minimum number of non-source nodes in any algebraic circuit computing the polynomial. Clearly this is at most as large as the formula complexity, but it can be smaller, as the next example shows.

Let us illustrate the subtleties of circuit complexity and formula complexity on two important polynomials. The determinant

$$\det(X) = \begin{vmatrix} x_{11} & \ldots & x_{1n} \\ \vdots & & \vdots \\ x_{n1} & \ldots & x_{nn} \end{vmatrix}$$

is a polynomial in n^2 variables. Using the familiar expansion of the determinant, it takes $(n-1)n!$ multiplications to evaluate it. On the other hand, using Gaussian elimination, it can be evaluated with $O(n^3)$ multiplications, and in fact this can be improved to $O(M(n)) = O(n^{2.3727})$ multiplications (see Exercise 9.3.3).

There is no compact (polynomial-size) formula to describe the determinant. In fact, there is no easy substantial improvement over the full expansion. But it can be proved that the determinant does have a formula of size $n^{O(\log n)}$. It is not known, however, whether the determinant has a polynomial-size formula (probably not).

The other polynomial to discuss here is the *permanent*. This is very similar to the determinant: it has a similar expansion into $n!$ terms, the only difference is that all expansion terms are added up with positive sign. So for example

$$\operatorname{per}\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = x_{11}x_{22} + x_{12}x_{21}.$$

The permanent plays an important role in various questions in combinatorics as well as in statistical physics. We mention only one: the permanent of the adjacency matrix of a bipartite graph G (in the bipartite sense) with n nodes in each class is the number of perfect matchings in G.

It turns out that the evaluation of the permanent is NP-hard, even if its entries are 0 or 1. This implies that there is no hope to find an algebraic computation of polynomial length for the evaluation of the permanent. It is even less likely to find a polynomial size formula for the permanent, but (quite frustratingly) this is not proved.

Exercise 9.3.1. *(a) Show by an example that in an algebraic computation of length n, where the input numbers are rational with bit complexity polynomial in n, the bit complexity of the result can be exponentially large.*

(b) Prove that if an algebraic computation of length at most n, the constants (in operations of type (A2)) as well as the input numbers are rational with bit complexity at most n, and we know that the output is an integer with at most n bits, then the output can be computed using a polynomial number of bit operations.

Exercise 9.3.2. *An LUP-decomposition of an $n \times n$ matrix A is a triple (L, U, P) of $n \times n$ matrices such that $A = LUP$, where L is a a lower triangular matrix with 1's in the diagonal, U is an upper triangular matrix, and P is a permutation matrix (informally, this means that we represent the matrix as the product of a lower triangular and an upper triangular matrix, up to a permutation of the columns). Show that if we can multiply two matrices with $M(n)$ arithmetic operations, then we can compute the LUP-decomposition of an $n \times n$ matrix with $O(M(N))$ arithmetic operations.*

Exercise 9.3.3. *Show that if we can multiply two $n \times n$ matrices with $M(n)$ arithmetic operations, then we can compute the determinant of a matrix using $O(M(n))$ arithmetic operations.*

Exercise 9.3.4. *Show that if we can invert an $n \times n$ matrix with $I(n)$ arithmetic operations, then we multiply two $n \times n$ matrices using $O(I(3n))$ arithmetic operations.*

Exercise 9.3.5. *Prove that to compute the product of an $n \times n$ matrix and a vector:*

$$\begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & & \vdots \\ x_{n1} & \dots & x_{nn} \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

takes at least n^2 nonlinear operations.

Chapter 10

Parallel algorithms

New technology makes it more urgent to develop the mathematical foundations of parallel computation. In spite of the energetic research done, the search for a canonical model of parallel computation has not settled on a model that would strike the same balance between theory and practice as the Random Access Machine. The main problem is the modeling of the communication between different processors and subprograms: this can happen on immediate channels, along paths fixed in advance, "radio broadcast" like, etc.

A similar question that can be modeled in different ways is the synchronization of the clocks of the different processors: this can happen with some common signals, or not even at all.

In this chapter, we treat only one model, the so-called parallel Random Access Machine, which has been elaborated most from a complexity-theoretic point of view. Results achieved for this special case expose, however, some fundamental questions of the parallelizability of computations. The presented algorithms can be considered, on the other hand, as programs written in some high-level language: they must be implemented according to the specific technological solutions.

10.1 Parallel random access machines

The most investigated mathematical model of machines performing parallel computation is the parallel Random Access Machine (PRAM). This consists of some fixed number p of identical Random Access Machines (processors). The program store of the machines is common and they also have a common memory consisting, say, of the cells $x[i]$ (where i runs through the integers). It will be convenient to assume (though it would not be absolutely necessary) that each processor owns an infinite number of program cells $u[i]$. At the beginning $u[0]$ contains the serial number of the processor (otherwise all processor would execute the same operations). Each processor can read and write its own cells $u[i]$ as well as the common memory cells $x[i]$. In other words, to the instructions allowed for the Random Access Machine, we must add the instructions

$$
\begin{array}{llll}
u[i] := 0; & u[i] := u[i] + 1; & u[i] := u[i] - 1; & \\
u[i] := u[i] + u[j]; & u[i] := u[i] - u[j]; & u[u[i]] := u[j]; & u[i] := u[u[j]]; \\
u[i] := x[u[j]]; & x[u[i]] := u[j]; & \text{IF } \ u[i] \leq 0 \ \text{ THEN } \ \text{GOTO } \ p &
\end{array}
$$

Furthermore, we also add multiplication and division to the instructions, that is we can also use $u[i] := u[i] * u[j]$ and $u[i] := u[i] \div u[j]$ where $*$ denotes multiplication and $a \div b$ is the largest c for which $|a| \geq |b| * c$. These are added so that each processor can compute from its serial number in a single step $x(f(u[0]))$, the cell of the input it has to read first, if f is some simple function.

We write the *input* into the cells $x[1], x[2], \ldots$. In addition to the input and the common program, we must also specify how many processors will be used; we can write this into the cell $x[-1]$. The processors carry out the program in parallel but in lockstep. (Since they can refer to their own name they will not necessarily compute the same thing.) We use a logarithmic cost function: the cost of writing or reading an integer k from a memory cell $x[t]$ or $u[t]$ is the total number of digits in k and t, i.e., approximately $\log |k| + \log |t|$. (In case of multiplication and division we also add to this the product of their digits.) The next step begins after each processor has finished the previous step. The machine stops when each processor arrives at a program line in which there is no instruction. The *output* is the content of the cells $x[i]$.

An important question to decide is how to regulate the use of the common memory. What happens if several processors want to write to or read from the same memory cell? Several conventions exist for the avoidance of these conflicts. We mention four of these:

- Two processors must not read from or write to the same cell. We call this the *exclusive-read, exclusive-write* (EREW) model. We could also call it *(completely) conflict-free*. This must be understood in such a way that it is the responsibility of the programmer to prevent attempts of simultaneous access to the same cell. If such an attempt occurs the machine signals program error.

- Maybe the most natural model is the one in which we permit many processors to read the same cell at the same time but when they want to write this way, this is considered a program error. This is called the *concurrent-read, exclusive-write* (CREW) model, and could also be called *half conflict-free*.

- Several processors can read from the same cell and write to the same cell but only if they want to write the same thing. (The machine signals a program error only if two processors want to write different numbers into the same cell). We call this model *concurrent-read, concurrent-write* (CRCW); it can also be called *conflict-limiting*.

- Many processors can read from the same cell or write to the same cell. If several ones want to write into the same cell the processor with the smallest serial number succeeds: this model is called *priority concurrent-read, concurrent-write* (P-CRCW), or shortly, the *priority* model.

Exercise 10.1.1. *a) Show that we can select the smallest from n numbers using n^2 processors on the conflict-limiting model in $O(1)$ steps.*

b) Show that this can be done using n processors in $O(\log \log n)$ steps.

Exercise 10.1.2.

a) Prove that one can determine which one of two 0-1-strings of length n is lexicographically larger, using n processors, in $O(1)$ steps on the priority model and in $O(\log n)$ steps on the conflict-free model.

b) Show that on the conflict-free model, this actually requires $\Omega(\log n)$ steps.*

c) How many steps are needed on the other two models?*

Exercise 10.1.3. *Show that the sum of two 0-1-sequences of length at most n, as binary numbers, can be computed with n^2 processors in $O(1)$ steps on the priority model.*

Exercise 10.1.4. *a) Show that the sum of n 0-1-sequences of length at most n as binary numbers can be computed, using n^3 processors, in $O(\log n)$ steps on the priority model.*

b) Show that n^2 processors are also sufficient for this.*

c) Show the same on the conflict-free model.*

d) How many steps are needed to multiply two n bit integers n^2 processors on the conflict-free model?

Exercise 10.1.5. *An interesting and non-unique representation of integers is the following. We write every n as $n = \sum_{i=0}^{r} b_i 4^i$ where $-3 \leq b_i \leq 3$ for each i. Show that the sum of two numbers given in such form can be computed using n processors in $O(1)$ steps on the conflict-free model.*

On the PRAM machines, it is necessary to specify the number of processors not only since the computation depends on this but also since this is — besides the time and the storage — an important complexity measure of the computation. If it is not restricted then we can solve very difficult problems very fast. We can decide, e.g., the 3-colorability of a graph if, for each coloring of the set of vertices and each edge of the graph, we make a processor that checks whether in the given coloring, the endpoints of the given edge have different colors. The results must be summarized yet, of course, but on the conflict-limiting machine, this can be done in a single step.

First, it might sound scary that an algorithm might need n^2 or n^3 processors. However, the following fundamental statement due to Brent says, informally, that if a problem can be solved faster in parallel using many processors, then the same is true for less processors. For this define the *total work* of an algorithm as the sum of the number of steps of all processors. (Here we ignore the cost function.)

It is easy to convince ourselves that the following statement holds.

Proposition 10.1.1. *If a computation can be performed with any number of processors in t steps and w total work (on any model), then for every positive integer p it can be solved with p processors in $\frac{w}{p} + t$ steps (on the same model).*

In particular, it can be performed on a sequential Random Access Machine in $w + t$ steps.

Proof. Suppose that in the original algorithm w_i processors are active in the i-th step, so $w = \sum_{i=1}^{t} w_i$. The i-th step can be obviously performed by p processors in $\lceil \frac{w_i}{p} \rceil$ steps, so we need in total $\sum_{i=1}^{t} \lceil \frac{w_i}{p} \rceil \leq \sum_{i=1}^{t} (\frac{w_i}{p} + 1) \leq \frac{w}{p} + t$ steps using p processors. □

As a corollary, if we have an algorithm using, say, n^2 processors for $\log n$ steps, then for every p (e.g., $p = \sqrt{n}$, or $p = 32$) we can make another that uses p processors and makes $(n^2 \log n)/p + \log n$ steps.

The fundamental question of the complexity theory of parallel algorithms is just the opposite of this: given is a sequential algorithm with time w and we would like to implement it on p processors in essentially w/p (say, in $O(w/p) \log w$) steps.

It is obvious that the above models are stronger and stronger since they permit more and more. It can be shown, however, that the computations we can do on the strongest one, the priority model, are not much faster than the ones performable on the conflict-free model (at least if the number of processors is not too large). The following lemma is concerned with such a statement.

Lemma 10.1.2. *For every program $\mathcal{P}$, there is a program $\mathcal{Q}$ such that if $\mathcal{P}$ computes some output from some input with p processors in time t on the priority model then $\mathcal{Q}$ computes on the conflict-free model the same with $O(p^2)$ processors in time $O(t \log^2 p)$.*

Proof. A separate processor of the conflict-free machine will correspond to every processor of the priority machine. These are called *supervisor processors*. Further, every supervisor processor will have p *subordinate* processors. One *step* of the priority machine computation will be simulated by a *stage* of the computation of the conflict-free machine.

The basic idea of the construction is that whatever is in the priority machine after a given step of the computation in a given cell z, should be contained, in the corresponding stage of the computation of the conflict-free machine, in each of the cells with addresses $2pz, 2pz + 1, \ldots, 2pz + p - 1$. If in a step of the priority machine, processor i must read or write cell z then in the corresponding stage of the conflict-free machine, the corresponding supervisor processor will read or write the cell with address $2pz + i$. This will certainly avoid all conflicts since the different processors use different cells modulo p.

We must make sure, however, that by the end of the stage, the conflict-free machine writes into each cell $2pz, 2pz + 1, \ldots, 2pz + p - 1$ the same number the priority rule would write into z in the corresponding step of the priority machine. For this, we insert a *phase* consisting of $O(\log p)$ auxiliary steps accomplishing this to the end of each stage.

First, each supervisor processor i that in the present stage has written into cell $2pz + i$, writes a 1 into cell $2pz + p + i$. Then, in what is called the first step of the phase, it looks whether there is a 1 in cell $2pz + p + i - 1$. If yes, it goes to sleep for the rest of the phase. Otherwise, it writes a 1 there and "wakes" a subordinate. In general, at the beginning of step k, processor i will have at most 2^{k-1} subordinates awake (including, possibly, itself); these (at least the ones that are awake) will examine the corresponding cells $2pz + p + i - 2^{k-1}, ..., 2pz + p + i - (2^k - 1)$. The ones that find a 1 go to sleep. Each of the others writes a 1, wakes a new subordinate, sends it 2^{k-1}

steps left while itself goes 2^k steps left. Whichever subordinate gets below $2pz+p$ goes to sleep; if a supervisor i does this, it knows already that it has "won".

It is easy to convince ourselves that if in the corresponding step of the priority machine, several processors wanted to write into cell z then the corresponding supervisor and subordinate processors cannot get into conflict while moving in the interval $[2pz+p, 2pz+2p-1]$. It can be seen namely that in the k-th step, if a supervisor processor i is active then the active processors $j \leq i$ and their subordinates have written 1 into each of the 2^{k-1} positions downwards starting with $2pz+p+i$ that are still $\geq 2pz+p$. If a supervisor processor or one of its subordinates started to the right from them and would reach a cell $\leq i$ in the k-th step, it will necessarily step into one of these 1's and go to sleep, before it could get into conflict with the i-th supervisor processor or its subordinates. This also shows that always a single supervisor will win, namely the one with the smallest number.

The winner still has the job to see to it that what it wrote into the cell $2pz+i$ will be written into each cell of interval $[2pz, 2pz+p-1]$. This is easy to do by a procedure very similar to the previous one: the processor writes the desired value into cell $2pz$, then it wakes a subordinate; the two of them write the desired value into the cells $2pz+1$ and $2pz+2$ then they wake one subordinate each, etc. When they all have passed $2pz+p-1$ the phase has ended and the next simulation stage can start.

We leave to the reader to plan the waking of the subordinates.

Each of the above "steps" requires the performance of several program instructions but it is easy to see that only a bounded number is needed, whose cost is, even in case of the logarithmic-cost model, only $O(\log p + \log z)$. In this way, the time elapsing between two simulating stages is only $O(\log p(\log p + \log z))$. Since the simulated step of the priority machine also takes at least $\log z$ units of time the running time is thereby increased only $O(\log^2 p)$-fold. □

In what follows if we do not say otherwise we use the conflict-free (EREW) model. According to the previous lemma, we could have agreed on one of the other models.

Randomization is, as we will see at the end of the next section, an even more important tool in the case of parallel computations than in the sequential case. The *randomized parallel Random Access Machine* differs from the above introduced parallel Random Access Machine only in that each processor has an extra cell in which, with probability 1/2, there is always 0 or an 1. If the processor reads this bit then a new random bit occurs in the cell. The random bits are completely independent (both within one processor and between different processors).

10.2 The class NC

We say that a program for the parallel Random Access Machine is an NC-program if there are constants $c_1, c_2 > 0$ such that for all inputs x the program works conflict-free with $O(|x|^{c_1})$ processors in time $O(\log^{c_2} |x|)$. (According to Lemma 10.1.2, it would not change this definition if we used e.g., the priority model instead.)

The class NC of languages consists of those languages $\mathcal{L} \subseteq \{0,1\}^*$ whose characteristic function can be computed by an NC-program.

Remark. The goal of the introduction of the class NC is not to model practically implementable parallel computations. In practice, we can generally use much more than logarithmic time but (at least in the foreseeable future) only on much fewer than polynomially many processors. The goal of the notion is to describe those problems solvable with a polynomial number of operations, with the additional property that these operations are maximally parallelizable (in case of an input of size n, on the completely conflict-free machine, $\log n$ steps are needed even to let all input bits have an effect on the output).

Obviously, $NC \subseteq P$. It is not known whether equality holds here but the answer is probably no.

Around the class NC, a complexity theory can be built similar to the one around the class P. The NC-*reduction* of a language to another language can be defined and, e.g., inside the class P, it can be shown that there are languages that are P-complete, i.e., to which every other language in P is NC-reducible. We will not deal with the details of this; rather, we confine ourselves to some important examples.

Proposition 10.2.1. *The adjacency-matrices of graphs containing a triangle form a language in* NC.

Algorithm 10.2.2. The NC-algorithm is essentially trivial. Originally, let $x[0] = 0$. First, we determine the number n of points of the graph. Then we instruct the processor with serial number $i+jn+kn^2$ to check whether the point triple (i, j, k) forms a triangle. If no then the processor halts. If yes then it writes a 1 into the 0'th common cell and halts. Whether we use the conflict-limiting or the priority model, we have $x[0] = 1$ at the end of the computation if and only if the graph has a triangle. (Notice that this algorithm makes $O(1)$ steps.)

Our next example is less trivial, moreover, at the first sight, it is surprising: the connectivity of graphs. The usual algorithms (breadth-first or depth-first search) are namely strongly sequential: every step depends on the result of the earlier steps. For the parallelization, we use a trick similar to the one we used earlier for the proof of Savitch's theorem (Theorem 4.2.4).

Proposition 10.2.3. *The adjacency matrices of connected graphs form a language in NC.*

Algorithm 10.2.4. We will describe the algorithm on the conflict-limiting model. Again, we instruct the processor with serial number $i + jn + kn^2$ to watch the triple (i, j, k). If it sees two edges in the triple then it inserts the third one. (If several processors want to insert the same edge then they all want to write the same thing into the same cell and this is permitted.) If we repeat this t times then, obviously, exactly those pairs of points will be connected whose distance in the original graph is at most 2^t. In this way, repeating $O(\log n)$ times, we obtain a complete graph if and only if the original graph was connected.

Clearly, it can be similarly decided whether in a given graph, there is a path connecting two given points, moreover, even the distance of two points can be determined by a suitable modification of the above algorithm.

Exercise 10.2.1. *Suppose we are given a graph G with vertices $\{1, 2, \ldots, n\}$ and non-*

negative edge-lengths, $c(uv)$. Define the $n \times n$ matrix A as $A(i,i) = 0$, $A(i,j) = c(ij)$, if ij is an edge and $A(i,j) = +\infty$ otherwise.

Consider the matrix-product where the product (of two elements) is replaced by addition and the addition (of the products of the pairs) is replaced by the minimum operation. Show that if we compute A^n for this matrix (which can be done in NC*, similarly to (Lemma 3.1.2)), then we get the shortest path for any pair of points.*

Exercise 10.2.2. *In Chapter 9 we have defined algebraic decision trees and algebraic formula and their depth/length.*

a) Suppose F is a length L algebraic formula, which contains only the $\{+,-,\cdot\}$ operations. Show that we can construct a $3\log L$ deep algebraic decision tree that computes F.

b) The same is true for Boolean formula and Boolean circuits of indegree 2.

c) Even if the algebraic formula F contains division, we can still construct a $4\log L$ deep algebraic decision tree that computes F.

Exercise 10.2.3. *a) Show that two length n sorted sequences can be merged into an sorted sequence on the conflict-free model with n processors in $O(\log n)$ steps. (Hint: merge the even and odd indexed subsequences recursively.)*

b) A length n sequence can be sorted on the conflict-free model with n processors in $O(\log^2 n)$ steps.

The next algorithm is maybe the most important tool of the theory of parallel computations.

Theorem 10.2.5 (Csánky's Theorem)**.** *The determinant of an arbitrary integer matrix can be computed by an* NC *algorithm. Consequently, the invertible matrices form an* NC*-language.*

Algorithm 10.2.6. We present an algorithm proposed by Chistov. The idea is now to try to represent the determinant by a suitable matrix power-series. Let B be an $n \times n$ matrix and let B_k denote the $k \times k$ submatrix in its left upper corner. Assume first that these submatrices B_k are not singular, i.e., that their determinants are not 0. Then B is invertible and according to the known formula for the inverse, we have

$$(B^{-1})_{nn} = \det B_{n-1} / \det B$$

where $(B^{-1})_{nn}$ denotes the element standing in the right lower corner of the matrix B^{-1}. Hence

$$\det B = \frac{\det B_{n-1}}{(B^{-1})_{nn}}.$$

Continuing this, we obtain

$$\det B = \frac{1}{(B^{-1})_{nn} \cdot (B_{n-1}^{-1})_{n-1,n-1} \cdots (B_1^{-1})_{11}}.$$

Let us write B in the form $B = I - A$ where $I = I_n$ is the $n \times n$ identity matrix. Assuming, for a moment, that the elements of A are small enough, the following series expansion holds:

$$B_k^{-1} = I_k + A_k + A_k^2 + \cdots ,$$

which gives

$$(B_k^{-1})_{kk} = 1 + (A_k)_{kk} + (A_k^2)_{kk} + \cdots .$$

Hence

$$\begin{aligned}\frac{1}{(B_k^{-1})_{kk}} =& \frac{1}{1 + (A_k)_{kk} + (A_k^2)_{kk} + \cdots} \\ =& 1 - [(A_k)_{kk} + (A_k^2)_{kk} + \cdots] + [(A_k)_{kk} + (A_k^2)_{kk} + \cdots]^2 - \cdots ,\end{aligned}$$

and hence

$$\det B = \prod_{k=1}^{n} (1 - [(A_k)_{kk} + (A_k^2)_{kk} + \cdots] + [(A_k)_{kk} + (A_k^2)_{kk} + \cdots]^2 - \cdots).$$

We cannot, of course, compute these infinite series composed of infinite series. We claim, however, that it is enough to compute only n terms from each series. More exactly, let us substitute tA in place of A where t is a real variable. For small enough t, the matrices $I_k - tA_k$ are certainly not singular and the above series expansions hold. We gain, however, more. After substitution, the formula looks as follows:

$$\det(I - tA) = \prod_{k=1}^{n} (1 - [t(A_k)_{kk} + t^2(A_k^2)_{kk} + \cdots] + [t(A_k)_{kk} + t^2(A_k^2)_{kk} + \cdots]^2 - \cdots).$$

Now comes the decisive idea: the left-hand side is a polynomial of t of degree at most n, hence from the power series on the right-hand side, it is enough to compute only the terms of degree at most n. In this way, $\det(I - tA)$ consists of the terms of degree at most n of the following polynomial:

$$F(t) = \prod_{k=1}^{n} [1 - \sum_{j=0}^{n} (- \sum_{m=1}^{n} t^m (A_k^m)_{kk})^j].$$

Now, however complicated the formula defining $F(t)$ may seem, it can be computed easily in the NC sense. Deleting from it the terms of degree higher than n, we get a polynomial identical to $\det(I - tA)$. Also, as a polynomial identity, our identity holds for all values of t, not only for the small ones, and no nonsingularity assumptions are needed. Substituting $t = 1$ here, we obtain $\det B$.

We note that Mahajan and Vinay gave in 1997 a combinatorial algorithm for computing the determinant that does not use division and thus can be also used for computing the determinants of matrices over rings. This algorithm can be also parallelized, using $O(\log^2 n)$ time in the EREW (completely conflict-free) model (with n^6 processors).

Randomization for parallel machines is even more important than in the sequential case. We define the *randomized* NC, or RNC, class of languages similarly as the class BPP was defined. RNC consists of those languages $\mathcal{L}$ for which there is a number $c > 0$ and a program computing, on each input $x \in \{0,1\}^*$, on the randomized $PRAM$ machine, with $O(|x|^c)$ processors (say, in a completely conflict-free manner), in time $O(\log |x|^c)$, either a 0 or a 1. If $x \in \mathcal{L}$ then the probability of the result 0 is smaller than $1/4$, if $x \notin \mathcal{L}$ then the probability of the result 1 is smaller than $1/4$.

Using Theorem 5.1.3 with random substitutions, we arrive at the following important application:

Corollary 10.2.7. *The adjacency matrices of the graphs with complete matchings form a language in* RNC.

It must be noted that the algorithm only determines whether the graph has a complete matching but it does not give the matching if it exists. This, significantly harder, problem can also be solved in the RNC sense (by an algorithm of Karp, Upfal and Wigderson).

Exercise 10.2.4. *Consider the following problem. Given a Boolean circuit and its input, compute its output. Prove that if this problem is in NC then P=NC.*

Chapter 11

Communication complexity

With many algorithmic and data processing problems, the main difficulty is the transport of information between different processors. Here, we will discuss a model which—in the simplest case of 2 participating processors—attempts to characterize the part of complexity due to the moving of data.

Let us be given thus two processors, and assume that each of them knows only part of the input. Their task is to compute some output from this; we will only consider the case when the output is a single bit, i.e., they want to determine some property of the (whole) input. We abstract from the time and other costs incurred by the local computation of the processors; we consider therefore only the communication between them. We would like to achieve that they solve their task having to communicate as few bits as possible. Looking from the outside, we will see that one processor sends a bit ε_1 to the other one; then one of them (maybe the other one, maybe the same one) sends a bit ε_2, and so on. At the end, both processors must "know" the bit to be computed.

To make it more graphic, instead of the two processors, we will speak of two *players*, Alice and Bob. Imagine that Alice lives in Europe and Bob lives in New Zealand before the age of the internet; then the assumption that the cost of communication dwarfs the cost of local computations is rather realistic.

What is the algorithm in the area of algorithmic complexity is the *protocol* in the area of communication complexity. This means that we prescribe for each player, for each stage of the game where his/her input is x and bits $\varepsilon_1, \dots, \varepsilon_k$ were sent so far (including who sent them) whether the next turn is his/her (this can only depend on the messages $\varepsilon_1, \dots, \varepsilon_k$ and not on x; it must namely be also known to the other player to avoid conflicts), and if yes then—depending on these—what bit must be sent. Each player knows this protocol, including the "meaning" of the messages of the other player (in case of what inputs could the other one have sent it). We assume that both players obey the protocol.

It is easy to give a trivial protocol: Let Alice send Bob the part of the input known to her. Then Bob can already compute the end result and communicate it to Alice using a single bit. We will see that this can be, in general, far from the optimum. We will also see that in the area of communication complexity, some notions can be formed that are similar to those in the area of algorithmic complexity, and these are often easier to handle. But before, we give a very simple example where the trivial protocol is not

optimal. Later we'll see that indeed, for many of the simple communication tasks the trivial protocol is best possible.

Suppose that both Alice and Bob have two bits, and they want to decide whether the binary sum of each input is the same. The trivial protocol would be that one of them, say Alice, sends her input to Bob, who then does all the computation (not too hard) and returns a bit telling Alice the answer. But of course, instead of sending both bits, it suffices to send the binary sum, which saves one bit.

One can argue that in this example, the protocol is still the trivial one: the only important information about Alice's input is the binary sum of her bits, so she does send Bob all but the redundant information she has.

11.1 Communication matrix and protocol-tree

Let Alice's possible inputs be $a_1, \ldots, a_n$ and Bob's possible inputs $b_1, \ldots, b_m$ (since the local computation is free it is indifferent for us how these are coded). Let c_{ij} be the value to be computed for inputs a_i and b_j. The matrix $C = (c_{ij})_{i=1}^{n}{}_{j=1}^{m}$ is called the *communication matrix* of the problem in question. This matrix completely describes the problem: both players know the whole matrix C. Alice knows the index i of a row of C, while Bob knows the index j of a column of C. Their task is to determine the element c_{ij}. The trivial protocol is that e.g., Alice sends Bob the number i; this means $\lceil \log n \rceil$ bits. (If $m < n$ then it is better, of course, to proceed the other way.)

Let us see first what a protocol means for this matrix. First of all, the protocol must determine who starts. Suppose that Alice sends first a bit ε_1. This bit must be determined by the index i known to Alice; in other words, the rows of C must be divided in two parts according to $\varepsilon_1 = 0$ or 1. The matrix C is thus decomposed into two submatrices, C_0 and C_1. This decomposition is determined by the protocol, therefore both players know it. Alice's message determines which one of C_0 and C_1 contains her row. From now on therefore the problem has been narrowed down to the corresponding smaller matrix.

The next message decomposes C_0 and C_1. If the sender is Bob then he divides the columns into two classes; if it is Alice then she divides the rows again. It is not important that the second message has the same "meaning" in each case, i.e., that it divides the same rows [columns] in the matrices C_0 and C_1; moreover, it is also possible that it subdivides the rows of C_0 and the columns of C_1 (Alice's message "0" means that "I have more to say", and her message "1" that "it is your turn").

Proceeding this way, we see that the protocol corresponds to a decomposition of the matrix to ever smaller submatrices. In each "turn", every actual submatrix is divided into two submatrices either by a horizontal or by a vertical split. (Such a decomposition into submatrices is called a *guillotine-decomposition.*) It is important to note that rows and columns of the matrix can be divided into two parts in an arbitrary way; their original order plays no role.

When does this protocol stop? If the players have narrowed down the possibilities to a submatrix C' then this means that both know that the row or column of the other one belongs to this submatrix. If from this, they can tell the result in all cases then either all elements of this submatrix are 0 or all are 1.

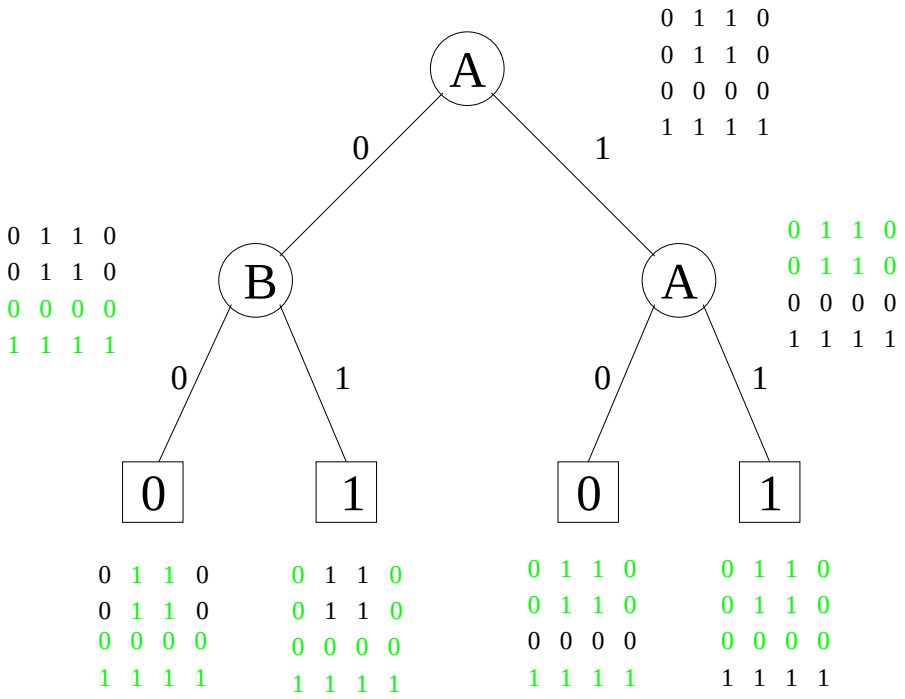

Figure 11.1: Protocol-tree

In this way, the determination of communication complexity leads to the following combinatorial problem: in how many turns can we decompose a given 0-1 matrix into matrices consisting of all 0's and all 1's, if in each turn, every submatrix obtained so far can only be split in two, horizontally or vertically? (If we obtain an all-0 or all-1 matrix earlier than the required number of turns, we can stop splitting it. But sometimes, it will be more useful to pretend that we keep splitting even in this case; formally, we agree that an all-0 matrix consisting of 0 rows can be split from an all-0 matrix as well as from an all-1 matrix.)

We can associate a binary tree to this process. Every point of the tree is a submatrix of C. The root is the matrix C, its left child is C_0 and its right child is C_1. The two children of every matrix are obtained by dividing its rows or columns into two classes. The leaves of the tree are all-0 or all-1 matrices.

Following the protocol, the players move on this tree from the root to some leaf. If they are in some node then whether its children arise by a horizontal or vertical split determines who sends the next bit. The bit is 0 or 1 according to whether the row [column] of the sender is in the left or right child of the node. If they arrive to a leaf then all elements of this matrix are the same and this is the answer to the communication problem. The *time requirement* of the protocol is the depth of this tree. The *communication complexity* of matrix C is the smallest possible time requirement of all protocols solving it. We denote it by $\kappa(C)$.

Note that if we split each matrix in each turn (i.e., if the tree is a complete binary tree) then exactly half of its leaves is all-0 and half is all-1. This follows from the fact that we have split all matrices of the penultimate "generation" into an all-0 matrix and an all-1 matrix. In this way, if the depth of the tree is t then among its leaves, there are 2^{t-1} all-1 (and just as many all-0). If we stop earlier on the branches where we

arrive earlier at an all-0 or all-1 matrix it will still be true that the number of all-1 leaves is at most 2^{t-1} since we could continue the branch formally by making one of the split-off matrices "empty".

This observation leads to a simple but important lower bound on the communication complexity of the matrix C. Let $\mathrm{rk}(C)$ denote the rank of matrix C.

Lemma 11.1.1.

$$\kappa(C) \geq 1 + \log \mathrm{rk}(C).$$

Proof. Consider a protocol-tree of depth $\kappa(C)$ and let $L_1, \ldots, L_N$ be its leaves. These are submatrices of C. Let M_i denote the matrix (having the same size as C) obtained by writing 0 into all elements of C not belonging to L_i. By the previous remark, we see that there are at most $2^{\kappa(C)-1}$ non-0 matrices M_i; it is also easy to see that all of these have rank 1. Now,

$$C = M_1 + M_2 + \cdots + M_N,$$

and thus, using the well-known fact from linear algebra that the rank of the sum of matrices is not greater than the sum of their rank,

$$\mathrm{rk}(C) \leq \mathrm{rk}(M_1) + \cdots + \mathrm{rk}(M_N) \leq 2^{\kappa(C)-1}.$$

This implies the lemma. □

Corollary 11.1.2. *If the rows of matrix C are linearly independent then the trivial protocol is optimal.*

Consider a simple but important communication problem to which this result is applicable and which will be an important example in several other aspects.

Example 11.1.1. Both Alice and Bob know some 0-1 sequence of length n; they want to decide whether the two sequences are equal.

The communication matrix belonging to the problem is obviously a $2^n \times 2^n$ identity matrix. Since its rank is 2^n no protocol is better for this problem than the trivial ($n+1$ bit) one.

By another, also simple reasoning, we can also show that almost this many bits must be communicated not only for the worst input but for almost all inputs:

Theorem 11.1.3. *Consider an arbitrary communication protocol deciding about two 0-1-sequence of length n whether they are identical, and let $h > 0$. Then the number of sequences $a \in \{0,1\}^n$ such that the protocol uses fewer than h bits on input (a, a) is at most 2^h.*

Proof. For each input (a, b), let $J(a, b)$ denote the "record" of the protocol, i.e., the 0-1-sequence formed by the bits sent to each other. We claim that if $a \neq b$ then $J(a, a) \neq J(b, b)$; this implies the theorem trivially since the number of $< h$-length records is at most 2^h.

Suppose that $J(a, a) = J(b, b)$ and consider the record $J(a, b)$. We show that this is equal to $J(a, a)$.

Suppose that this is not so, and let the i-th bit be the first one in which they differ. On the inputs (a, a), (b, b) and (a, b) not only the first $i-1$ bits are the same but also the direction of communication. Alice namely cannot determine in the first $i-1$ steps whether Bob has the sequence a or b, and since the protocol determines for her whether it is her turn to send, it determines this the same way for inputs (a, a) and (a, b). Similarly, the i-th bit will be sent in the same direction on all three inputs, say, Alice sends it to Bob. But at this time, the inputs (a, a) and (a, b) seem to Alice the same and therefore the i-th bit will also be the same, which is a contradiction. Thus, $J(a, b) = J(a, a)$.

The protocol terminates on input (a, b) by both players knowing that the two sequences are different. But from Alice's point of view, her own input as well as the communication are the same as on input (a, a), and therefore the protocol comes to wrong conclusion on that input. This contradiction proves that $J(a, a) \neq J(b, b)$. □

One of the main applications of communication complexity is that sometimes we can get a lower bound on the number of steps of algorithms by estimating the amount of communication between certain data parts. To illustrate this we give another proof for part (b) of Theorem 1.2.3. Recall that a *palindrome* is a string with the property that it is equal to its reverse. Now we will prove that every one-tape Turing machine needs $\Omega(n^2)$ steps to decide about a sequence of length $2n$ whether it is a palindrome.

Proof. Consider an arbitrary one-tape Turing machine deciding this question. Let us seat Alice and Bob in such a way that Alice sees cells $n, n-1, \ldots, 0, -1, \ldots$ of the tape and Bob sees its cells $n+1, n+2, \ldots$; we show the structure of the Turing machine to both of them. At start, both see therefore a string of length n and must decide whether these strings are equal (Alice's sequence is read in reverse order).

The work of the Turing machine offers a simple protocol to Alice and Bob: Alice mentally runs the Turing machine as long as the scanning head is on her half of the tape, then she sends a message to Bob: "the head moves over to you with this and this internal state". Then Bob runs it mentally as long as the head is in his half, and then he tells Alice the internal state with which the head must return to Alice's half, etc. So, if the head moves over k times from one half to the other one then they send each other $k \cdot \log |\Gamma|$ bits (where Γ is the set of states of the machine). At the end, the Turing machine writes the answer into cell 0 and Alice will know whether the word is a palindrome. For the price of 1 bit, she can let Bob also know this.

According to Theorem 11.1.3, we have therefore at most $2^{n/2}$ palindromes with $k \log |\Gamma| < n/2$, i.e., for most inputs, the head passed between the cells n and $(n+1)$ at least cn times, where $c = 1/(2 \log |\Gamma|)$. This is still only $\Omega(n)$ steps but a similar reasoning shows that for all $h \geq 0$, with the exception of $2^h \cdot 2^{n/2}$ inputs, the machine passes between cells $(n-h)$ and $(n-h+1)$ at least cn times. For the sake of proving this, consider a palindrome α of length $2h$ and write in front of it a sequence β of length $n-h$ and behind it a sequence γ of length $n-h$. The sequence obtained this way is a palindrome if and only if $\beta = \gamma^{-1}$ where we denoted by γ^{-1} the inversion of γ. By Theorem 11.1.3 and the above reasoning, for every α there are at most $2^{n/2}$ strings β for which on input $\beta\alpha\beta^{-1}$, the head passes between cells $n-h$ and $n-h+1$ fewer than cn times. Since the number of α's is 2^h the assertion follows.

If we add up this estimate for all h with $0 \le h \le n/2$ the number of exceptions is at most

$$2^{n/2} + 2 \cdot 2^{n/2} + 4 \cdot 2^{n/2} + \cdots + 2^{n/2-1} \cdot 2^{n/2} < 2^n,$$

hence there is an input on which the number of steps is at least $(n/2) \cdot (cn) = \Omega(n^2)$. □

Exercise 11.1.1. *Show that the following communication problems cannot be solved with fewer than the trivial number of bits $(n+1)$: The inputs of Alice and Bob are two subsets of an n-element set, X and Y. They must decide whether*

- *X and Y are disjoint;*
- *$|X \cap Y|$ is even.*

11.2 Examples

We give some examples when the trivial protocol (Alice sending all her information to Bob) is not optimal. The examples are not simple, demonstrating how carefully planned protocols can save a lot.

Example 11.2.1. There is a tree T with n nodes, known to both players. Alice has a subtree T_A and Bob has a subtree T_B. They want to decide whether the subtrees have a common vertex.

The trivial protocol uses obviously $\log M + 1$ bits where M is the number of subtrees. M can even be greater than 2^{n-1} if e.g., T is a star. So the trivial protocol may take n bits. (For different subtrees, Alice's message must be different. If Alice gives the same message for subtrees T_A and T'_A and, say, $T_A \not\subseteq T'_A$ then T_A has a vertex v that is not in T'_A; if Bob's subtree consists of the single point v then he cannot find the answer based on this message.)

Consider, however, the following protocol: Alice chooses a vertex $x \in V(T_A)$ and sends it to Bob (we reserve a special message for the case when T_A is empty; in this case, they will be done). If x is also a vertex of the tree T_B then they are done (Bob has a special message for this case). If not, then Bob determines the point of T_B closest to x (this is the node where the path from x to any node of T_B hits T_B). He sends this node y to Alice.

Now if y is in T_A, then Alice knows that the two trees have a common point; if y is not in T_A then the two trees have no common points at all. She sends one bit to tell the result.

This protocol uses only $1 + 2\lceil \log(n+1) \rceil$ bits (see Figure 11.2).

Exercise 11.2.1. *Prove that in Example 11.2.1, any protocol requires at least $\log n$ bits.*

Exercise* 11.2.2. *Refine the protocol in Example 11.2.1 that it uses only $\log n + \log\log n + 1$ bits.*

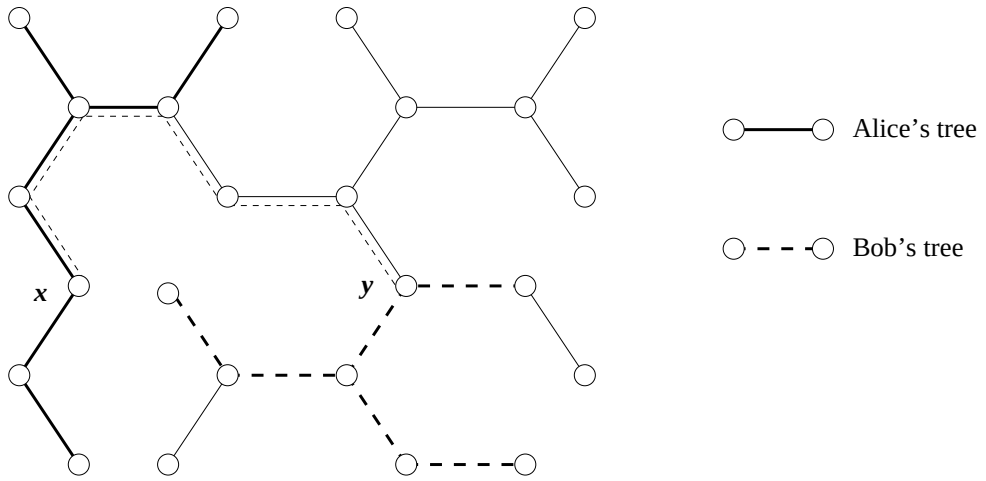

Figure 11.2: Protocol for disjointness of subtrees

Example 11.2.2. Given is a graph G with n points. Alice knows a vertex set S_A spanning a complete subgraph and Bob knows an independent vertex set S_B in the graph. They want to decide whether the two subgraphs have a common vertex.

If Alice wants to give the complete information to Bob about the vertex set known to her then $\log M$ bits would be needed, where M is the number of complete subgraphs. This can be, however, even $2^{n/2}$, i.e., (in the worst case) Alice must use $\Omega(n)$ bits. The situation is similar with Bob.

The following protocol is significantly more economical. Alice checks whether the set S_A has a vertex with degree at most $n/2-1$. If there is one, then she sends to Bob a 1 and then the name of such a vertex v. Then both of them know that Alice's set consists only of v and some of its neighbors, i.e., they reduced the problem to a graph with $n/2$ vertices.

If every node of S_A has degree larger than $n/2-1$ then Alice sends to Bob only a 0. Then Bob checks whether the set S_B has a vertex with degree larger than $n/2-1$. If it has then it sends Alice a 1 and the name of such a node w. Similarly to the foregoing, after this both of them will know that besides w, the set S_B can contain only vertices that are not neighbors of w, and they thus again succeeded in reducing the problem to a graph with at most $(n+1)/2$ vertices.

Finally, if every vertex of S_B has degree at most $n/2-1$, Bob sends a 0 to Alice. After this, they know that their sets are disjoint.

The above turn uses at most $O(\log n)$ bits and since it decreases the number of vertices of the graph to half, it will be repeated at most $\log n$ times. Therefore, the complete protocol is only $O((\log n)^2)$. More careful computation shows that the number of used bits is at most $\lceil \log n \rceil (2 + \lceil \log n \rceil)/2$.

11.3 Non-deterministic communication complexity

As with algorithms, the nondeterministic version plays an important role also with protocols. This can be defined—in a fashion somewhat analogous to the notion of

"witness", or "testimony"—in the following way. We want that for every input of Alice and Bob for which the answer is 1, a "superior being" can reveal a short 0-1 sequence convincing both Alice and Bob that the answer is indeed 1. They do not have to believe the revelation of the "superior being" but if they signal anything at all this can only be whether on their part they accept the proof or not. This non-deterministic protocol consists therefore of certain possible "revelations" $z_1, \dots, z_n \in \{0,1\}^t$ all of which are acceptable for certain inputs of Alice and Bob. For a given pair of inputs, there is an z_i acceptable for both of them if and only if for this pair of inputs, the answer to the communication problem is 1. The parameter t, the length of the z_i's is the *complexity* of the protocol. Finally, the *nondeterministic communication complexity* of matrix C is the minimum complexity of all non-deterministic protocols applicable to it; we denote this by $\kappa_{ND}(C)$

Example 11.3.1. In Example 11.1.1, if the superior being wants to prove that the two strings are different it is enough for her to declare: "Alice's i-th bit is 0 while Bob's is not." This is—apart from the textual part, which belongs to the protocol—only $\lceil \log n \rceil + 1$ bits, i.e., much less than the complexity of the optimal deterministic protocol.

We remark that even the superior being cannot give a proof that two words are equal in fewer than n bits, as we will see right away.

Example 11.3.2. Suppose that Alice and Bob know a convex polygon each in the plane, and they want to decide whether the two polygons have a common point.

If the superior being wants to convince the players that their polygons are not disjoint she can do this by revealing a common point. Both players can check that the revealed point indeed belongs to their polygon.

We can notice that in this example, the superior being can also easily prove the negative answer: if the two polygons are disjoint then it is enough to reveal a straight line such that Alice's polygon is on its left side, Bob's polygon is on its right side. (We do not discuss here the exact number of bits in the inputs and the revelations.)

Let z be a possible revelation of the superior being and let H_z be the set of all possible pairs (i,j) for which z "convinces" the players that $c_{ij} = 1$. We note that if $(i_1, j_1) \in H_z$ and $(i_2, j_2) \in H_z$ then (i_1, j_2) and (i_2, j_1) also belong to H_z: since $(i_1, j_1) \in H_z$, Alice, possessing i_1, accepts the revelation z; since $(i_2, j_2) \in H_z$, Bob, possessing j_2, accepts the revelation z; thus, when they have (i_1, j_2) both accept z, hence $(i_1, j_2) \in H_z$.

We can therefore also consider H_z as a submatrix of C consisting of all 1's. The submatrices belonging to the possible revelations of a nondeterministic protocol cover the 1's of the matrix C since the protocol must apply to all inputs with answer 1 (it is possible that a matrix element belongs to several such submatrices). The 1's of C can therefore be covered with at most $2^{\kappa_{ND}(C)}$ all-1 submatrices.

Conversely, if the 1's of the matrix C can be covered with 2^t all-1 submatrices then it is easy to give a non-deterministic protocol of complexity t: the superior being reveals only the number of the submatrix covering the given input pair. Both players verify whether their respective input is a row or column of the revealed submatrix. If yes then they can be convinced that the corresponding matrix element is 1. We have

thus proved the following statement:

Lemma 11.3.1. *$\kappa_{ND}(C)$ is the smallest natural number t for which the 1's of the matrix C can be covered with 2^t all-1 submatrices.*

In the negation of Example 11.3.1, the matrix C is the $2^n \times 2^n$ identity matrix. Obviously, only the 1×1 submatrices of this are all-1, the covering of the 1's requires therefore 2^n such submatrices. Thus, the non-deterministic complexity of this problem is also n.

Let $\kappa(C) = s$. Then C can be decomposed into 2^s submatrices half of which are all-0 and half are all-1. According to Lemma 11.3.1, the nondeterministic communication complexity of C is therefore at most $s - 1$. Hence

$$\kappa_{ND}(C) \leq \kappa(C) - 1.$$

Example 11.3.1 shows that there can be a big difference between the two quantities.

Let $\overline{C}$ denote the matrix obtained from C by changing all 1's to 0 and all 0's to 1. Obviously, $\kappa(C) = \kappa(\overline{C})$. Example 11.3.1 also shows that $\kappa_{ND}(C)$ and $\kappa_{ND}(\overline{C})$ can be very different. On the basis of the previous remarks, we have

$$\max\{1 + \kappa_{ND}(C),\ 1 + \kappa_{ND}(\overline{C})\} \leq \kappa(C).$$

The following important theorem shows that here, already, the difference between the two sides of the inequality cannot be too great.

Theorem 11.3.2 (Aho-Ullman-Yannakakis)**.**

$$\kappa(C) \leq (\kappa_{ND}(C) + 2) \cdot (\kappa_{ND}(\overline{C}) + 2).$$

We will prove a sharper inequality. In case of an arbitrary 0-1 matrix C, let $\varrho(C)$ denote the largest number t for which C has a $t \times t$ submatrix in which—after a suitable rearrangement of the rows and columns—there are all 1's in the main diagonal and all 0's everywhere above the main diagonal. Obviously,

$$\varrho(C) \leq \mathrm{rk}(C),$$

and Lemma 11.3.1 implies

$$\log \varrho(C) \leq \kappa_{ND}(C).$$

The following inequality therefore implies Theorem 11.3.2.

Theorem 11.3.3.

$$\kappa(C) \leq 2 + \log \varrho(C)(2 + \kappa_{ND}(\overline{C}))$$

unless C is the all zero matrix.

Proof. We use induction on $\varrho(C)$. If $\varrho(C) = 1$ then the protocol is trivial. Let $\varrho(C) > 1$ and $p = \kappa_{ND}(\overline{C})$. Then the 0's of the matrix C can be covered with 2^p all-0 submatrices, say, $M_1, \ldots, M_{2^p}$. We want to give a protocol that decides the communication problem

with at most $(p+2)\log \varrho(C)$ bits. The protocol fixes the submatrices M_i, this is therefore known to the players.

For every submatrix M_i, let us consider the matrix A_i formed by the rows of C intersecting M_i and the matrix B_i formed by the columns of C intersecting M_i. The basis of the protocol is the following, very easily verifiable, statement.

Claim 11.3.4.

$$\varrho(A_i) + \varrho(B_i) \leq \varrho(C).$$

Now, we can describe the following protocol.

Alice checks whether there is an index i for which M_i intersects her row and for which $\varrho(A_i) \leq \frac{1}{2}\varrho(C)$. If yes, then she sends "1" and the index i to Bob, with this the first phase of the protocol has ended. If not, then she sends "0". Now, Bob checks whether there is an index i for which M_i intersects his column and $\varrho(B_i) \leq \frac{1}{2}\varrho(C)$. If yes, then he sends a "1" and the index i to Alice. Else he sends "0". Now the first phase has ended in any case.

If either Alice or Bob find a suitable index in the first phase, then by the communication of at most $p+2$ bits they have restricted the problem to a matrix C' ($= A_i$ or B_i) for which $\varrho(C') \leq \frac{1}{2}\varrho(C)$. Hence the theorem follows by induction.

If both players sent "0" in the first phase then they can finish the protocol: the answer is "1". Indeed, if there was a 0 in the intersection of Alice's row and Bob's column then this would belong to some submatrix M_i. However, for these submatrices, we have on the one hand

$$\varrho(A_i) > \frac{1}{2}\varrho(C)$$

(since they did not suit Alice), on the other hand

$$\varrho(B_i) > \frac{1}{2}\varrho(C)$$

since they did not suit Bob. But this contradicts the above claim. □

It is interesting to formulate another corollary of the above theorem (compare it with Lemma 11.1.1):

Corollary 11.3.5.

$$\kappa(C) \leq 2 + \big(\log rk(C)\big)\big(\kappa_{ND}(\overline{C}) + 2\big)$$

unless C is the all zero matrix.

To show the power of Theorems 11.3.2 and 11.3.3 consider the examples treated in Section 11.2. If C is the matrix corresponding to Example 11.2.1 (in which 1 means that the subtrees are disjoint) then $\kappa_{ND}(\overline{C}) \leq \lceil \log n \rceil$ (it is sufficient to name a common vertex). It is also easy to obtain that $\kappa_{ND}(C) \leq 1 + \lceil \log(n-1) \rceil$ (if the subtrees are disjoint then it is sufficient to name an edge of the path connecting them, together with telling that after deleting it, which component will contain T_A and which one T_B). It can also be shown that the rank of C is $2n$. Therefore, whichever of Theorem 11.3.2 or

11.3.3 we use, we get a protocol using $O((\log n)^2)$ bits. This is much better than the trivial one but is not as good as the special protocol treated in Section 11.2.

Let now C be the matrix corresponding to Example 11.2.2. It is again true that $\kappa_{ND}(\overline{C}) \leq \lceil \log n \rceil$, for the same reason as above. It can also be shown that the rank of C is exactly n. From this it follows, by Theorem 11.3.3, that $\kappa(C) = O((\log n)^2)$ which is (apart from a constant factor) the best know result. It must be mentioned that what is known for the value of $\kappa_{ND}(C)$, is only the estimate $\kappa_{ND} = O((\log n)^2)$ coming from the inequality $\kappa_{ND} \leq \kappa$.

Remark. We can continue dissecting the analogy of algorithms and protocols a little further. Let us be given a set $\mathcal{H}$ of (for simplicity, quadratic) 0-1 matrices. We say that $\mathcal{H} \in \mathrm{P}^{\mathrm{comm}}$ if the communication complexity of every matrix $C \in \mathcal{H}$ is not greater than a polynomial of $\log \log N$ where N is the number of rows of the matrix. (I.e., if the complexity is a good deal smaller than the trivial $1 + \log N$.) We say that $\mathcal{H} \in \mathrm{NP}^{\mathrm{comm}}$ if the non-deterministic communication complexity of every matrix $C \in \mathcal{H}$ is not greater than a polynomial of $\log \log N$. We say that $\mathcal{H} \in \mathrm{co{-}NP}^{\mathrm{comm}}$ if the matrix set $\{\, \overline{C} : C \in \mathcal{H} \,\}$ is in $\mathrm{NP}^{\mathrm{comm}}$. Then Example 11.3.1 shows that

$$\mathrm{P}^{\mathrm{comm}} \neq \mathrm{NP}^{\mathrm{comm}},$$

and Theorem 11.3.2 implies

$$\mathrm{P}^{\mathrm{comm}} = \mathrm{NP}^{\mathrm{comm}} \cap \mathrm{co{-}NP}^{\mathrm{comm}}.$$

Exercise 11.3.1. *Prove claim 11.3.4.*

Exercise 11.3.2. *Show that in Theorems 11.3.2 and 11.3.3 and in Corollary 11.3.5, with more care, the factor* $(2 + \kappa_{ND})$ *can be replaced with* $(1 + \kappa_{ND})$.

11.4 Randomized protocols

In this part, we give an example showing that randomization can decrease the complexity of protocols significantly. We consider again the problem whether the inputs of the two players are identical. Both inputs are 0-1 sequences of length n, say x and y. We can also view these as natural numbers between 0 and $2^n - 1$. As we have seen, the communication complexity of this problem is $n + 1$.

If the players are allowed to choose random numbers then the question can be settled much easier, by the following protocol. The only change on the model is that both players have a random number generator; these generate independent bits (it does not restrict generality if we assume that the bits of the two players are independent of each other, too). The bit computed by the two players will be a random variable; the protocol is good if this is equal to the "true" value with probability at least $2/3$.

Protocol: Alice chooses a random prime number p in the interval $1 < p < n$ and divides x by p with remainder. Let the remainder be r; then Alice sends Bob the numbers p and r. Bob checks whether $y \equiv r \pmod{p}$. If not, then he knows that $x \neq y$. If yes, then he concludes that $x = y$.

First we note that this protocol uses only $2\log N + 1$ bits since $1 \leq r \leq p \leq N$. The problem is that it may be wrong; let us find out in what direction and with what probability. If $x = y$, then it gives always the right result. If $x \neq y$, then it is conceivable that x and y give the same remainder at division by p and so the protocol arrives at a wrong conclusion. This occurs if p divides the difference $d = |x - y|$. Let $p_1, \ldots, p_k$ be the prime divisors of d, then

$$d \geq p_1 \cdots p_k \geq 2 \cdot 3 \cdot 5 \cdot \cdots \cdot q,$$

where q is the k-th prime number.

It is a known number-theoretical fact that for large enough q we have, say,

$$2 \cdot 3 \cdot 5 \cdot \cdots \cdot q > 2^q.$$

Since $d < 2^n$ it follows from this that $q < n$ and therefore $k \leq \pi(n)$ (where $\pi(n)$ is the number of primes up to n). Hence the probability that we have chosen a prime divisor of d can be estimated as follows:

$$\mathsf{P}(p \textit{ divides } d) = \frac{k}{\pi(N)} \leq \frac{\pi(n)}{\pi(N)}.$$

Now, according to the prime number theorem, we have $\pi(n) \approx n/\log n$ and so if we choose $N = cn$ then the above bound is asymptotically $1/c$, i.e., it can be made arbitrarily small with the choice of c. At the same time, the number of bits to be transmitted is only $2\log N + 1 = 2\log n +$ constant.

Remark. The use of randomness does not help in every communication problem this much. We have seen in the exercises that determining the disjointness or the parity of the intersection of two sets behaves, from the point of view of deterministic protocols, as the decision of the identity of 0-1 sequences. These problems behave, however, already differently from the point of view of protocols that also allow randomization: Chor and Goldreich have shown that $\Omega(n)$ bits are needed for the randomized computation of the parity of intersection, and Kalyanasundaram and Schnittger proved similar lower bound for the randomized communication complexity of the decision of disjointness of two sets.

Chapter 12

An application of complexity: cryptography

The complexity of a phenomenon can be the main obstacle of exploring it. Our book—we hope—proves that complexity is not only an obstacle to research but also an important and exciting subject. It goes, however, beyond this: it has applications where it is precisely the complexity of a phenomenon that is exploited. We have discussed the problem of generating *pseudorandom numbers* in Chapter 7. This chapter treats another such subject: *cryptography*, i.e., the science of secret codes. It was the application of the results of complexity theory that elevated secret codes beyond the well-known (military, intelligence) applications, and made them one of the most important ingredient of computer security, electronic trade, internet etc.

12.1 A classical problem

Sender wants to send a message x to Receiver (where x is e.g., a 0-1-sequence of length n). The goal is that when the message gets into the hands of any unauthorized third party, she should not understand it. For this, we encode the message, which means that instead of the message, Sender sends a code y of it, from which the receiver can recompute the original message but the unauthorized eavesdropper cannot. For this, we use a key d that is (say) also a 0-1-sequence of length n. Only Sender and Receiver know this key.

Thus, Sender computes a "code" $y = f(x, d)$ that is also a 0-1-sequence of length n. We assume that for all d, $f(\cdot, d)$ is a bijective mapping of $\{0,1\}^n$ to itself. Then $f^{-1}(\cdot, d)$ exists and thus Receiver, knowing the key d, can reconstruct the message x. The simplest, frequently used function f is $f(x, d) = x \oplus d$ (bitwise addition modulo 2).

Remark. This so-called *one-time pad* method is very safe. It was used during World War II for communication between the American President and the British Prime Minister. Its disadvantage is that it requires a very long key. It can be expensive to make sure that Sender and Receiver both have such a common key; but note that the key can be sent at a safer time and by a completely different method than the message.

12.2 A simple complexity-theoretic model

Let us look at a problem now that has—apparently—nothing to do with the above one. From a certain bank, we can withdraw money using an ATM. The client types his name or account number (in practice, he inserts a card on which these data are stored) and a password. The bank's computer checks whether this is indeed the client's password. If this checks out, the automaton hands out the desired amount of money. In theory, only the client knows this password (it is not written on his card!), so if he takes care that nobody else can find it out, then this system provides complete security.

The problem is that the bank must also know the password and therefore a bank employee can abuse it. Can one design a system in which it is impossible to figure out the password, even with the knowledge of the complete password-checking program? This seemingly self-contradictory requirement is satisfiable!

Here is a solution (not a very efficient one, and certainly never used in practice, but one that illustrates the idea how complexity theory enters this field). The client takes n nodes numbered from 1 to n, draws in a random Hamiltonian circuit and then adds arbitrary additional edges. He remembers the Hamiltonian circuit; this will be his password. He gives the whole graph to the bank (without marking the Hamiltonian circuit in it).

If somebody shows up at the bank in the name of the client and gives a set of edges on the n nodes as her password, the bank checks it whether it is a Hamiltonian circuit of the graph stored there. If so, the password will be accepted; if not, it will be rejected.

Now it seems that we have made it easier to impersonate our client: the impostor does not have to know the password (the particular Hamiltonian circuit); she can give any other Hamiltonian circuit of the client's graph. But note that even if she learns the graph, she must still solve the problem of finding a Hamiltonian circuit in a graph. And this is NP-hard!

Remarks. 1. Instead of the Hamiltonian circuit problem, we could have based the system on any other NP-complete problem.

2. We glossed over a difficult question: how many more edges should the client add to the graph and how? The problem is that the NP-completeness of the Hamiltonian circuit problem means only that its solution is hard in the *worst case*. We don't know how to construct *one* graph in which there is a Hamiltonian circuit but it is hard to find.

It is a natural idea to try to generate the graph by random selection. If we chose it randomly from among all n-point graphs then it can be shown that in it, with large probability, it is easy to find a Hamiltonian circuit. If we chose a random one among all n-point graphs with m edges then the situation is similar both with too large m and with too small m. The case $m = n \log n$ at least seems hard. In some cases, one can show that certain randomized constructions yield instances of NP-hard problems that are hard with high probability (in the sense that if one could solve a random instance in polynomial time with non-negligible probability, then we could solve all instances in

randomized polynomial time). These studies are beyond the scope of this book.

12.3 Public-key cryptography

In this section, we describe a system that improves on the methods of classical cryptography in several points. Let us note first of all that the system intends to serve primarily civil rather than military goals. For using electronic mail, in particular, if we use it for electronic commerce, we must recreate some tools of traditional correspondence like envelope, signature, company letterhead, etc.

The system has $N \geq 2$ participants. Every participant has a public key e_i (she will publish it e.g., in a phone-book-like directory) and a secret key d_i known to herself only. There is, further, a publicly known encoding/decoding function that computes from every message x and (secret or public) key e a message $f(x, e)$. (The message x and its code must come from some easily specifiable set H; this can be e.g., $\{0,1\}^n$ but can also be the set of residue classes modulo m. We assume that the message itself contains the names of the sender and receiver also in "human language".) For every $x \in H$ and every i with $1 \leq i \leq N$, we must have

$$f(f(x, e_i), d_i) = f(f(x, d_i), e_i) = x. \tag{12.1}$$

If participant i wants to send a message to j then she sends the message $y = f(f(x, d_i), e_j)$ instead. From this, j can compute the original message by the formula $x = f(f(y, d_j), e_i)$.

For this system to be usable, trivially it must satisfy

(C1) $f(x, e_i)$ can be computed efficiently from x and e_i.

The security of the system will be guaranteed by

(C2) $f(x, d_i)$ cannot be computed efficiently even in the knowledge of x, e_i and an arbitrary number of $d_{j_1}, \ldots, d_{j_h}$ $(j_r \neq i)$.

By "efficient", we mean polynomial time, but the system makes sense under other resource-bounds too. A function f with the above properties will be called a *trapdoor function.*

Condition (C1) guarantees that if participant i sends a message to participant j then she can encode it in polynomial time and the addressee can decode it in polynomial time. Condition (C2) can be interpreted to say that if somebody encoded a message x with the public key of a participant i and then she lost the original then no coalition of the participants can restore the original (efficiently) if i is not among them. This condition provides the "security" of the system. It implies, besides the classical requirement, a number of other security conditions.

Proposition 12.3.1. *Only j can decode a message addressed to j.*

Proof. Assume that a group $k_1, \ldots, k_r$ of unauthorized participants finds out the message $f(f(x, d_i), e_j)$, and knows even who sent it to whom. Suppose that they can

compute x efficiently from this. Then $k_1, \dots, k_r$ and i together could compute x also from $f(x, e_j)$. Let, namely, $z = f(x, e_j)$; then $k_1, \dots, k_r$ and i knows the message $f(x, e_j) = f(f(z, d_i), e_j)$ and thus using the method of $k_1, \dots, k_j$, can compute z. But from this, they can compute x by the formula $x = f(z, d_i)$, which contradicts condition (C2). □

The following can be verified by similar reasoning:

Proposition 12.3.2. *Nobody can forge a message in the name of i, i.e., participant j receiving a message that he can successfully decode using the public key of i (and his own private key), can be sure that the message could have been sent only by i.*

Proposition 12.3.3. *j can prove to a third person (e.g., in a court of justice) that i has sent the given message; in the process, the secret elements of the system (the keys d_i) need not be revealed.*

Proposition 12.3.4. *j cannot change the message (and have it accepted e.g., in a court as coming from i) or send it in the name of i to somebody else.*

It is not at all clear, of course, whether trapdoor functions exists. Several such functions have been proposed; many of the proposed systems turned out to be insecure later on—the corresponding complexity conditions were not true.) In the next section, we describe one such system that is one of the earliest, and is most widely used (and of course, to our current knowledge, is secure).

12.4 The Rivest-Shamir-Adleman code (RSA code)

In a simpler version of this system (in its abbreviated form, the *RSA code*), the "post office" generates two n-digit prime numbers, p and q for itself, and computes the number $m = pq$. It publishes this number (but the prime decomposition remains secret!). Then it generates, for each subscriber, a number e_i with $1 \le e_i < m$ that is relatively prime to $(p-1)$ and $(q-1)$. (It can do this by generating a random e_i between 0 and $(p-1)(q-1)$ and checking by the Euclidean algorithm whether it is relatively prime to $(p-1)(q-1)$. If it is not, it tries a new number. It is easy to see that after an expected number of $\log n$ trials, it finds a good number e_i with high probability.) Then, using the Euclidean algorithm, it finds a number d_i with $1 \le d_i < m$ such that

$$e_i d_i \equiv 1 \pmod{(p-1)(q-1)}.$$

(here $(p-1)(q-1) = \varphi(m)$, the number of positive integers smaller than m and relatively prime to it). The public key is the number e_i, the secret key is the number d_i. The message x itself is considered a natural number with $0 \le x < m$ (if it is longer then it will be cut into pieces). The encoding function is defined by the formula

$$f(x, e) = x^e \pmod{m} \qquad 0 \le f(x, e) < m.$$

The same formula serves for decoding, only with d in place of e.

The inverse relation between coding and decoding (formula 12.1) follows from the "little" Fermat theorem. By definition, $e_i d_i = 1 + \varphi(m)r = 1 + r(p-1)(q-1)$ where r is a natural number. Thus, if $(x, p) = 1$ then

$$f(f(x, e_i), d_i) \equiv (x^{e_i})^{d_i} = x^{e_i d_i} = x(x^{p-1})^{r(q-1)} \equiv x \pmod{p}.$$

On the other hand, if $p|x$ then obviously

$$x^{e_i d_i} \equiv 0 \equiv x \pmod{p}.$$

Thus

$$x^{e_i d_i} \equiv x \pmod{p}$$

holds for all x. It similarly follows that

$$x^{e_i d_i} \equiv x \pmod{q},$$

and hence

$$x^{e_i d_i} \equiv x \pmod{m}.$$

Since both the first and the last number are between 0 and $m-1$ it follows that they are equal, i.e., $f(f(x, e_i), d_i) = x$.

It is easy to check condition (C1): knowing x, e_i and m, the remainder of x^{e_i} after division by m can be computed in polynomial time, as we have seen it in Chapter 3. Condition (C2) holds only in the following, weaker form:

(C2') $f(x, d_i)$ cannot be computed efficiently from the knowledge of x and e_i.

This condition can be formulated to say that with respect to a composite modulus, extracting the e_i-th root cannot be accomplished in polynomial time without knowing the prime decomposition of the modulus. We cannot prove this condition (even with the hypothesis P≠NP) but at least it seems true according to the present state of number theory.

Several objections can be raised against the above simple version of the RSA code. First of all, the post office can decode every message, since it knows the numbers p, q and the secret keys d_i. But even if we assume that this information will be destroyed after setting up the system, unauthorized persons can still misuse the system. In fact, *Every participant of the system can solve any message sent to any other participant.* (This does not contradict condition (C2') since participant j of the system knows, besides x and e_i, also the key d_j.)

Indeed, consider participant j and assume that she got her hands on the message $z = f(f(x, d_i), e_k)$ sent to participant k. Let $y = f(x, d_i)$. Participant j solves the message not meant for her as follows. She computes a factoring $u \cdot v$ of $(e_j d_j - 1)$, where $(u, e_k) = 1$ while every prime divisor of v also divides e_k. To do this, she computes, by the Euclidean algorithm, the greatest common divisor v_1 of e_k and $e_j d_j - 1$, then the greatest common divisor v_2 of e_k and $(e_j d_j - 1)/v_1$, then the greatest common divisor v_3 of e_k and $(e_j d_j - 1)/(v_1 v_2)$, etc. This process terminates in at most $t = \lceil \log(e_j d_j - 1) \rceil$ steps, i.e., $v_t = 1$. Then $v = v_1 \cdots v_t$ and $u = (e_j d_j - 1)/v$ gives the desired factoring.

Notice that $(\varphi(m), e_k) = 1$ and therefore $(\varphi(m), v) = 1$. Since $\varphi(m)|e_j d_j - 1 = uv$, it follows that $\varphi(m)|u$. Since $(u, e_k) = 1$, there are natural numbers s and t with $se_k = tu + 1$. Then

$$z^s \equiv y^{se_k} = y(y^u)^t \equiv y \pmod{m}$$

and hence

$$x \equiv y^{e_i} \equiv z^{e_i s}.$$

Thus, participant j can also compute x.

Exercise 12.4.1. *Show that even if all participants of the system are honest an outsider can cause harm as follows. Assume that the outsider gets two versions of one and the same letter, sent to two different participants, say $f(f(x, d_i), e_j)$ and $f(f(x, d_i), e_k)$ where $(e_j, e_k) = 1$ (with a little luck, this will be the case). Then he can reconstruct the text x.*

Now we describe a better version of the RSA code. Every participant generates two n-digit prime numbers, p_i and q_i and computes the number $m_i = p_i q_i$. Then she generates for herself a number e_i with $1 \leq e_i < m_i$ relatively prime to $(p_i - 1)$ and $(q_i - 1)$. With the help of the Euclidean algorithm, she finds a number d_i with $1 \leq d_i < m_i$ for which

$$e_i d_i \equiv 1 \pmod{(p_i - 1)(q_i - 1)}.$$

The public key consists of the pair (e_i, m_i) and the secret key of the pair (d_i, m_i).

The message itself will be considered a natural number. If $0 \leq x < m_i$ then the encoding function will be defined, as before, by the formula

$$f(x, e_i, m) \equiv x^{e_i} \pmod{m_i}, \qquad 0 \leq f(x, e_i, m_i) < m_i.$$

Since, however, different participants use different moduli, it will be practical to extend the definition to a common domain, which can even be chosen to be the set of natural numbers. Let x be written in a base m_i notation: $x = \sum_j x_j m_i^j$, and compute the function by the formula

$$f(x, e_i, m_i) = \sum_j f(x_j, e_i, m_i) m_i^j.$$

We define the decoding function similarly, using d_i in place of e_i.

For the simpler version it follows, similarly to what was said above, that these functions are inverses of each other, that (C1) holds, and that it can also be conjectured that (C2) holds. In this version, the "post office" holds no non-public information, and of course, each key d_j has no information on the other keys. Therefore, the above mentioned errors do not occur.

Chapter 13

Circuit complexity

A central, but extremely difficult problem in the theory of computation is to prove lower bounds on the time and space complexity of various computational tasks. The key problem is whether P=NP, but much simpler questions remain unanswered. The approach of classical logic, trying to extend the methods that were used in Chapter 2 to prove undecidability results, seems to fail badly.

Another, more promising approach to proving lower bounds on the computational complexity of a problem is combinatorial. This approach focuses on the Boolean circuit model of computation, and tries to analyze the (rather complex) flow of information through the steps of the computation. We illustrate this method by a beautiful (but rather difficult) proof in this spirit (it might underline the difficulty of these questions that this is perhaps the easiest proof to tell in this area!).

We discuss two very simple functions, already introduced in Chapter 1: the *majority function*

$$\text{MAJORITY}(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if at least } n/2 \text{ of the variables is 1;} \\ 0, & \text{otherwise.} \end{cases}$$

and the *parity function* or XOR function

$$\text{PARITY}(x_1, \ldots, x_n) = x_1 + x_2 + \cdots + x_n \pmod 2.$$

These functions are of course very easy to compute, but suppose we want to do it in parallel in very little time. Instead of going into the complications of PRAMs, let us consider a more general model of parallel computation, namely, Boolean circuits with small depth. We will allow arbitrary fan-in and fan-out (this is analogous to the concurrent-read-concurrent-write model of parallel computation), although as the following exercise shows, this would not be necessary.

Exercise 13.0.2. *If a Boolean circuit has fan-in 2, and it computes a function in n variables that depends on each of its variables, then its depth is at least* $\log n$.

We can recall now from Chapter 1 that every Boolean function can be computed by a Boolean circuit of depth 2. However, it is easy to see that even for simple functions like the majority function, the resulting circuit is exponentially large. On the other hand, if we have a polynomial time algorithm to compute a Boolean function, then this can be converted (again, as described in Chapter 1) to a Boolean circuit of polynomial size.

However, this circuit will have large (typically linear, if you are careful, logarithmic) depth.

Can we simultaneously restrict the size to polynomial and the depth to less than logarithmic? The answer is negative even for quite simple functions. In a series of increasingly stronger results, Furst–Saxe–Sipser, Ajtai, Yao and Hastad proved that every constant-depth circuit computing the parity function has exponential size, and that every polynomial-size circuit computing the parity function has (essentially) logarithmic depth. Let us state the result in detail (the proof is too complex to be given here).

Theorem 13.0.1. *Every circuit with n input bits and depth d that computes the parity function has at least $2^{(1/10)n^{1/(d-1)}}$ gates.*

Not much later Razborov proved analogous results for the majority function. In fact, he proved a stronger result by allowing circuits that may have *parity gates*, or XOR gates, in addition to the usual AND, OR and NOT gates, where a parity gate computes the binary sum of any number of bits. The proof, though not easy, can be reproduced here for the enjoyment of the truly interested.

13.1 Lower bound for the Majority Function

Let us start with the exact statement of the theorem.

Theorem 13.1.1. *If C is a circuit of depth d, with* AND, OR, XOR, *and* NOT *gates that computes the majority function of $2n-1$ input bits, then C has at least $2^{n^{(1/2d)}}/10\sqrt{n}$ gates.*

The idea of the proof is to introduce "approximations" of the gates used during the computation. Using the approximate gates, instead of the real gates, one computes an approximation of the majority function. The quality of the approximation will be measured in terms of the number of inputs on which the modified circuit differs from the original. The main point of the approximation is to keep the computed function "simple" in some sense. We will show that every "simple" function, and in particular the approximation we compute, differs from the majority function on a significant fraction of the inputs. Since the approximation of each gate has a limited effect on the function computed, we can conclude that many approximations had to occur.

The proof will be easier to describe if we generalize the result to a family of closely related functions, the *k-threshold functions* f_k. The k-threshold function is 1 when at least k of the inputs are 1. As Exercise 13.1.1 shows, if there is a circuit of size s that computes the majority function of $2n-1$ elements in depth d, then for each k, $1 \leq k \leq n$, there is a circuit of depth d and size at most s that computes the k-threshold function on n elements. Therefore, any exponential lower bound for f_k implies a similar bound for the majority function. We shall consider $k = \lceil (n+h+1)/2 \rceil$ for an appropriate h.

Exercise 13.1.1. *Suppose that there is a circuit of size s that computes the majority function of $2n-1$ elements in depth d. Show that for each k, $1 \leq k \leq n$, there is*

a circuit of depth d and size at most s that computes the k-threshold function on n elements.

Each Boolean function can be expressed as a polynomial over the two-element field $GF(2)$. (So in this chapter addition of 0-1 polynomials will be always understood modulo 2.) In fact, such a representation can be computed following the computation described by the circuit. If p_1 and p_2 are polynomials representing two functions, then p_1+p_2 is the polynomial corresponding to the XOR of the two functions. The polynomial p_1p_2 corresponds to their AND, which makes it easy to see that $(p_1 + 1)(p_2 + 1) + 1$ corresponds to their OR. The polynomial $1 - p$ corresponds to the negation of the function represented by the polynomial p.

The measure of simplicity of a Boolean function f for this proof is the degree of the polynomial representing the function or for short, the *degree* of the function. Note that the inputs have degree 1, i.e., they are very simple. But the degree may grow very fast as we follow the circuit; in fact, since we do not restrict fan-in, a single OR gate can produce a function with arbitrarily high degree!

The trick is to show that these functions can be approximated by polynomials of low degree. The following lemma will serve as the basis for the approximation.

Lemma 13.1.2. *Let $g_1, \ldots, g_m$ be Boolean functions of degree at most h on n binary inputs. If $r \geq 1$ and $f = \vee_{i=1}^{m} g_i$, then there is a function f' of degree at most rh that differs from f on at most 2^{n-r} inputs.*

Proof. Let us go through the indices $1, 2, \ldots, m$ one by one, and for each such number, flip a coin. If we get HEAD, we select this number; else, we move on. Let I_1 be the set of numbers selected (so I_1 is a random subset of $\{1, \ldots, m\}$). We repeat this experiment r times, to get the random subsets $I_1, \ldots, I_r$. Let

$$f'_j = \sum_{i \in I_j} g_i,$$

and consider $f' = \vee_{j=1}^{r} f'_j$. We claim that the probability that f' satisfies the requirements of the lemma is non-zero.

It is clear that the degree of the polynomial for f' is at most rh. Furthermore, consider an input α; we claim that the probability that $f'(\alpha) \neq f(\alpha)$ is at most 2^{-r}. To see this, consider two cases. If $g_i(\alpha) = 0$ for every i, then both $f(\alpha) = 0$ and $f'(\alpha) = 0$. On the other hand, if there exists an index i for which $g_i(\alpha) = 1$, then $f(\alpha) = 1$ and for each j, $f'_j(\alpha) = 0$ independently with probability at most $1/2$. Therefore, $f'(\alpha) = 0$ with probability at most 2^{-r}, and the expected number of inputs on which $f' \neq f$ is at most 2^{n-r}. Hence for at least one particular choice of the sets I_j, the polynomial f' differs from f on at most 2^{n-r} inputs. □

Corollary 13.1.3. *The same holds for the* AND *function instead of* OR.

Proof. Let $f = \wedge_{i=1}^{m} g_i = 1 + f_0$, where $f_0 = \vee_{i=1}^{m}(1 + g_i)$. Since the degree of the functions $1 + g_i$ is at most h, the function $f' = 1 + f'_0$ satisfies the requirements. □

Next we show that any function of low degree has to differ from the k-threshold function on a significant fraction of the inputs.

Lemma 13.1.4. *Let $n/2 < k \leq n$. Every polynomial with n variables of degree $h = 2k - n - 1$ differs from the k-threshold function on at least $\binom{n}{k}$ inputs.*

Proof. Let g be a polynomial of degree h and let $\mathcal{B}$ denote the set of vectors where it differs from f_k. Let $\mathcal{A}$ denote the set of all 0-1 vectors of length n containing exactly k 1's.

For each Boolean function f, consider the summation function $\hat{f}(x) = \sum_{y \leq x} f(y)$. It is trivial to see that the summation function of the monomial $x_{i_1} \cdots x_{i_r}$ is 1 for the incidence vector of the set $\{i_1, \ldots, i_r\}$ and 0 on all other vectors. Hence it follows that f has degree at most h if and only if $\hat{f}$ vanishes on all vectors with more than h 1's. In contrast to this, the summation function of the k-threshold f_k is 0 on all vectors with fewer than k 1's, but 1 on all vectors with exactly k 1's.

Consider the matrix $M = (m_{ab})$ whose rows are indexed by the members of $\mathcal{A}$, and whose columns are indexed by the members of $\mathcal{B}$, and

$$m_{ab} = \begin{cases} 1, & \text{if } a \geq b, \\ 0, & \text{otherwise.} \end{cases}$$

We want to show that the columns of this matrix generate the whole space $GF(2)^{\mathcal{A}}$. This will imply that $|\mathcal{B}| \geq |\mathcal{A}| = \binom{n}{k}$.

Let $a_1, a_2 \in \mathcal{A}$ and let $a_1 \wedge a_2$ denote their coordinatewise minimum. Then we have, by the definition of $\mathcal{B}$,

$$\sum_{\substack{b \leq a_1 \\ b \in \mathcal{B}}} m_{a_2 b} = \sum_{\substack{b \leq a_1 \wedge a_2 \\ b \in \mathcal{B}}} 1 = \sum_{u \leq a_1 \wedge a_2} \big(f_k(u) + g(u)\big) = \sum_{u \leq a_1 \wedge a_2} f_k(u) + \sum_{u \leq a_1 \wedge a_2} g(u).$$

The second term of this last expression is 0, since $a_1 \wedge a_2$ contains at least $h + 1$ 1's. The first term is also 0 except if $a_1 = a_2$. The columns of M therefore generate the unit vector corresponding to the coordinate a_1, and so they generate the whole space. □

It is easy now to complete the proof theorem 13.1.1. Assume that there is a circuit of size s and depth d to compute the k-threshold function for inputs of size n. Now apply Lemma 13.1.2 with $r = \lfloor n^{1/(2d)} \rfloor$ to approximate the OR and AND gates in this circuit. The functions computed by the gates at the i-th level will be approximated by polynomials of degree at most r^i. Therefore, each resulting approximation p_k of the k-threshold function will have degree at most r^d. Lemma 13.1.4 implies that for $k = \lceil (n + r^d + 1)/2 \rceil$, the polynomial p_k differs from the k-threshold function on at least $\binom{n}{k}$ inputs. This shows that $s 2^{n-r} \geq \binom{n}{k}$. From this, routine calculations yield that

$$s \geq \binom{n}{k} 2^{r-n} > \frac{2^r}{10\sqrt{n}},$$

which establishes the desired exponential lower bound. □

Exercise* 13.1.2. *Let* Mod3 *denote the Boolean function whose value is 1 if and only if the sum of its inputs is divisible by 3. Prove that if a constant depth circuit of* AND, OR, XOR *and* NOT *gates computes* Mod3, *then its size is exponential.*

13.2 Monotone circuits

Perhaps the deepest result on circuit complexity was obtained by Razborov in 1985. The main point is that he does not make any assumption on the depth of the circuit; but, unfortunately, he still has to make a rather strong restriction, namely *monotonicity*. A Boolean circuit is *monotone*, if all of its input nodes are unnegated variables, and it has no NOT gates. Obviously, such a circuit can compute only monotone functions; but it is not difficult to see that every monotone function can be computed by a monotone circuit. The monotonicity of the function is not a serious restriction; many interesting functions in NP (e.g. matching, clique, colorability etc.) are monotone. For example, the k-clique function is defined as follows: it has $\binom{n}{2}$ variables x_{ij}, $1 \leq i < j \leq n$, and its value is 1 if and only if the graph described by the particular setting of the variables has a clique of size k. Since some of these are NP-complete, it follows that every problem in NP can be reduced in polynomial time to the computation of a monotone function in NP.

Razborov gave a superpolynomial lower bound on the monotone circuit complexity of the clique problem, without any restriction on the depth. This result was strengthened by Andreev, Alon and Boppana, to an exponential lower bound on the monotone circuit complexity of the k-clique function.

Unfortunately, while, as we have seen, restricting the functions to monotone functions does not mean a substantial restriction of generality, the restriction of the circuits to monotone circuits does. É. Tardos constructed a family of monotone Boolean functions which can be computed in polynomial time, but need an exponential number of gates if we use a monotone circuit.

Chapter 14

Interactive proofs

14.1 How to save the last move in chess?

Alice and Bob are playing chess over the phone. They want to interrupt the game for the night; how can they do it so that the person to move should not get the improper advantage of being able to think about his move whole night? At a tournament, the last move is not made on the board, only written down, put in an envelope, and deposited with the referee. But now the two players have no referee, no envelope, no contact other than the telephone line. The player making the last move (say, Alice) has to tell something; but this information should not be enough to determine the move, Bob would gain undue advantage. The next morning (or whenever they continue the game) she has to give some additional information, some "key", which allows Bob to reconstruct the move.

Surely this is impossible?! If she gives enough information the first time to uniquely determine his move, Bob will know her move; if the information given the first time allows several moves, then she can think about it overnight, and change her mind without being noticed. If we measure information in the sense of classical information theory, then there is no way out of this dilemma. But complexity comes to our help: it is not enough to communicate information, it must also be processed.

To describe a solution, we need a few words on algorithmic number theory. Given a natural number N, it can be tested in polynomial time whether it is a prime. These algorithms are also practically quite efficient, and work well up to several hundred digits. However, all the known prime testing algorithms have the (somewhat strange) feature that if they conclude that N is not a prime, they do not (usually) find a decomposition of N into a product of two smaller natural numbers (usually, they conclude that N is not a prime by finding that N violates Fermat's "Little" Theorem). It seems impossible to find the prime factorization of a natural number N in polynomial time.

Remark. Of course, very powerful supercomputers and massively parallel systems can be used to find decompositions by brute force for fairly large numbers; the current limit is around 100 digits, and the difficulty grows very fast (exponentially) with number of digits. To find the prime decomposition of a number with 400 digits is way beyond the possibilities of computers in the foreseeable future.

Returning to our problem, Alice and Bob can agree to encode every move as a 4-digit number. Alice extends these four digits to a prime number with 200 digits. (She can randomly extend the number and then test if the resulting number is a prime. By the Prime Number Theorem, she will have a success out of every $\ln 10^{200} \approx 460$ trials.) She also generates another prime with 201 digits and computes the product of them. The result is a number N with 400 or 401 digits; she sends this number to Bob. Next morning, she sends both prime factors to Bob. He checks that they are primes, their product is N, and reconstructs Alice's move from the first four digits of the smaller prime.

The number N contains all the information about her move: this consists of the first four digits of the smaller prime factor of N. Alice has to commit herself to the move when sending N. To find out the move of Alice, Bob would have to find the prime factors of N; this is, however, hopeless. So Bob only learns the move when the factors are revealed the next morning.

What Alice and Bob have established is an electronic "envelope" of a method to deposit information at a certain time that can be retrieved at a given later time, and cannot be changed in the meanwhile. The key ingredient of their scheme is *complexity*: the computational difficulty to find the factorization of an integer. In this scheme, factoring could be replaced by any other problem in NP that is (probably) not in P.

To give an exact definition, we restrict ourselves to hiding a single bit. (Longer messages can be hidden bit-by-bit.) So define a *keybox* as a function $f : \{0,1\} \times \{0,1\}^n \to \{0,1\}^N$ with the following properties:

(a) the function is polynomial time computable (this in particularly implies that N is bounded by a polynomial of n);

(b) $f(0,y) \neq f(1,y)$ $(y \in \{0,1\}^n)$;

(c) for every randomized polynomial time algorithm $\mathcal{A} : \{0,1\}^N \to \{0,1\}$ and for every $x \in \{0,1\}$, if $y \in \{0,1\}^n$ is chosen uniformly at random, then the probability that $f(\mathcal{A}(f(x,y)) = x$ is at most negligibly larger than $1/2$.

The above scheme shows how to construct a keybox if we assume that prime factorization is difficult.

14.2 How to check a password – without knowing it?

In a bank, a cash machine works by name and password. This system is safe as long as the password is kept secret. But there is one week point in security: the computer of the bank must store the password, and the programmer may learn it and later misuse it.

Complexity theory provides a scheme where the bank can verify that the costumer does indeed know the password — without storing the password itself. One solution uses the same construction as our telephone chess example. The password is a 200-digit prime number P (this is, of course, too long for everyday use, but it illustrates the idea best). When the costumer chooses the password, he also chooses another prime with 201 digits, forms the product N of the two primes, and tells the bank the number N. When the teller is used, the costumer tells his name and the password P. The computer of the bank checks whether or not P is a divisor of N; if so, it accepts P as a

proper password. The division of a 400 digit number by a 200 digit number is a trivial task for a computer.

Let us assume now that a programmer learns the number N stored along with the files of our costumer. To use this in order to impersonate the costumer, he has to find a 200-digit number that is a divisor of N; but this is essentially the same problem as finding the prime factorization of N, and, as remarked above, is hopelessly difficult. So — even though all the necessary information is contained in the number N — the computational complexity of the factoring problem protects the password of the costumer!

There are many other schemes to achieve the same, let us recall the one already mention in Chapter 12.2, where the customer chooses a *Hamilton cycle* in a graph with a 1000 vertices and adds 5000 further edges to it.

Many other problems in mathematics (in a sense, every problem in NP that is not in P) gives rise to a password scheme.

14.3 How to use your password – without telling it?

The password scheme discussed in the previous section is secure if the program is honest; but what happens if the program itself contains a part that simply reads off the password the costumer uses, and tells it to the programmer? In this case, the password is compromised if it is used but once. There does not seem to be any way out — how could one use the passport without telling it to the computer?

It sounds paradoxical, but *there is a scheme which allows the costumer to convince the bank that he knows the password — without giving the slightest hint as to what the password is!* We will give an informal description of the idea (following Blum (1987)), by changing roles: let me be the costumer and you (the audience) play the role of the computer of the bank. I will use two overhead projectors. The graph G shown on the first projector is the graph known to the bank; I label its nodes for convenience. My password is a Hamilton cycle (a polygon going through each node exactly once) in this graph, which I know but do not want to reveal.

I have prepared two further transparencies. The first contains a polygon with the same number of nodes as G, drawn with random positions for the nodes and without labels. The second transparency, laid over the first, contains the labels of the nodes and the edges that complete it to a drawing of the graph G (with the nodes in different positions). The two transparencies are covered by a sheet of paper.

Now the audience may choose: should I remove the top sheet or the two top sheets? No matter which choice is made, the view contains no information about the Hamiltonian cycle, since it can be predicted: if the top sheet is removed, what is shown is a re-drawing of the graph G, with its nodes in random positions; if the two top sheets are removed, what is shown is a polygon with the right number of nodes, randomly drawn in the plane.

But the fact that the audience sees what is expected is an evidence that I know a Hamiltonian cycle in G! For suppose that G contains no Hamiltonian cycle, or at least I don't know such a cycle. How can I prepare the two transparencies? Either I cheat by drawing something different on the bottom transparency (say, a polygon with

fewer nodes, or missing edges), or by having a different graph on the two transparencies together.

Of course, I may be lucky (say, I draw a polygon with fewer edges and the audience opts to view the two transparencies together) and I will not be discovered; but if I cheat, I only have a chance of 1/2 to get away with it. We can repeat this 100 times (each time with a brand new drawing!); then my chance for being lucky each time is less than 2^{-100}, which is much less than the probability that a meteorite hits the building during this demonstration. So if a 100 times in a row, the audience sees what it expects, this is a *proof* that I know a Hamiltonian cycle in the graph!

To make this argument precise, we have to get rid of non-mathematical notions like overhead projectors and transparencies. The reader may figure out how to replace these by keyboxes, which have been mathematically defined in Section 14.1.

Exercise 14.3.1. *Give a formal description of the above proof using envelopes.*

Remark. The most interesting aspect of the scheme described above is that it extends the notion of a *proof*, thought (at least in mathematics) to be well established for more than 2000 years. In the classical sense, a proof is written down entirely by the author (whom we call the *Prover*), and then it is verified by the reader (whom we call the *Verifier*). Here, there is *interaction* between the Prover and the Verifier: the action taken by the Prover depends on "questions" by the Verifier. The notion of interactive proof systems was introduced independently by Goldwasser, Micali and Rackoff and by Babai around 1985, and has lead to many deep and surprising results in computer science and mathematical logic.

The kind of interactive proof described in this section, where the Verifier gets no information other than a single bit telling that the claim of the Prover is correct, is called a *zero-knowledge proof*. This notion was introduced by Goldwasser, Micali and Rackoff. We will not elaborate on it in these notes.

Interactive proofs are however interesting from many other aspects, and we discuss them in detail in the rest of this Chapter.

Remark. There is another feature which distinguishes this scheme from a classical proof: it also makes use of *lack of information*, namely, that I cannot possibly know in advance the choice of the audience. (If I am telling, say, Gauss' proof that the regular 7-gon cannot be constructed by compass and ruler, then it remains a valid proof even if I anticipate the questions from the audience, or even have a friend "planted" in the audience asking the questions I wish.) In a sense, a password scheme is also an interactive proof: my password is a proof that I have the right to access the account — and again, a certain "lack of information" is a key assumption, namely, that no unauthorized person can possibly know my password.

Exercise 14.3.2. *Give a zero-knowledge proof for the* 3*-colorability of a graph.*

14.4 How to prove non-existence?

It is easy to prove that a graph is Hamiltonian: it suffices to exhibit a Hamiltonian cycle (the language of Hamiltonian graphs is in NP). But how to prove it if it is *not* Hamiltonian?

There are many problems (e.g, the existence of a perfect matching, or embeddability in the plane), for which there is also an easy way to prove non-existence; in other words, these problems are in NP∩co-NP. For these problems, the existence of such a proof often depends on rather deep theorems, such as the Marriage Theorem of Frobenius and König, or Tutte's theorem, or Kuratowski's Theorem on planarity. But for the Hamiltonian cycle problem no similar result is known; in fact, if NP≠co-NP then no NP-complete problem can have such a "good characterization".

It turns out that with interaction, things get easier, and a Prover (being computationally very powerful himself) can convince a Verifier (who works in polynomial time and, in particular, is able to follow proofs only if they have polynomial length relative to the input size) about nonexistence. We start with the relatively simple interactive protocol for the graph isomorphism problem.

Problem 14.4.1. *Given two graphs G_1 and G_2, are they isomorphic?*

Trivially, the problem is in NP, and it is not known whether it is in co-NP (it is probably not). However, there is a simple interactive protocol by which the Prover can convince the verifier that the two graphs are not isomorphic.

The Verifier selects one of the two graphs at random (each with probability 1/2, and randomly relabels its nodes, to get a third graph G_3. She then asks the Prover to guess which of G_1 and G_2 it comes from.

If the two graphs are not isomorphic, the Prover can run any simple isomorphism test to tell which of G_1 and G_2 is isomorphic to G_3. (This may take exponential time, but the Prover is very powerful, and can do this.) On the other hand, if the two graphs are isomorphic, the Prover only sees 3 isomorphic graphs, and has only a chance of 1/2 to guess the answer correctly. (If this probability of being cheated is too high for the Verifier, she can repeat it a hundred times, so that the probability that the Prover succeeds in proving a false thing is less than 2^{-100}.)

One can design an interactive protocol to prove that a given graph has no Hamiltonian cycle, is not 3-colorable, etc. In fact, it suffices to design such a protocol for the negation of any NP-complete problem; then all problems in NP can be reduced to it. We shall sketch the protocol, due to Nisan, that works for the following NP-complete problem:

Problem 14.4.2. *Given a polynomial $p(x_1, \ldots, x_n)$ with integral coefficients (say, of degree n), is there a point $y \in \{0,1\}^n$ such that $f(y) \neq 0$?*

Trivially, the problem is in NP, and in fact it is NP-complete. (See Exercise 14.4.1.) But how to prove if the polynomial vanishes on $\{0,1\}^n$?

The property of f the Prover wants to prove is equivalent to saying that

$$\sum_{x_1,\ldots,x_n \in \{0,1\}} p^2(x_1, \ldots, x_n) = 0. \tag{14.1}$$

Of course, the sum on the left hand side has an enormous number of terms, so the

Verifier cannot check it directly. The Prover suggests to consider the one-variable polynomial

$$f_1(x) = \sum_{x_2,\dots,x_n\in\{0,1\}} p^2(x, x_2, \dots, x_n),$$

and offers to reveal its explicit form:

$$f_1(x) = a_0 + a_1 x + \cdots + a_d x^d, \tag{14.2}$$

where $d = 2n$. The Verifier checks, using this form, that $f_1(0) + f_1(1) = 0$; this is easy. But how to know that (14.2) is the true form of $f_1(x)$? To verify the coefficients one-by-one would mean to reduce an instance in n variables to $d+1$ instances in $n-1$ variables, which would lead to a hopelessly exponential protocol with more than d^n steps.

Here is the trick: the Verifier chooses a random value $\xi \in \{0, \dots, 2n^3\}$, and requests a proof of the equality

$$f_1(\xi) = a_0 + a_1\xi + \cdots + a_d\xi^d. \tag{14.3}$$

Note that if the polynomials on both sides of (14.2) are distinct, then they agree on at most $d = 2n$ places, and hence the probability that the randomly chosen integer ξ is one of these places is at most $2n/2n^3 = 1/n^2$. So if the Prover is cheating, the chance that he is lucky at this point is at most $1/n^2$.

Now (14.3) can be written as

$$\sum_{x_2,\dots,x_n\in\{0,1\}} p^2(\xi, x_2 \dots, x_n) = b, \tag{14.4}$$

where the verifier easily calculates the value $b = a_0 + a_1\xi + \cdots + a_d\xi^d$. This is of the same form as (14.1) (except that the right hand side is not 0, which is irrelevant) and that now we have only $n-1$ variables. Therefore, the Prover can prove recursively that (14.4) holds. The amount of exchange is $O(d \log n)$ bits per iteration, which gives a total $O(dn \log n)$ bits; the total computational time used by the verifier is clearly also polynomial.

Since the problem solved above is NP-complete, every other problem in NP can be reduced to it, and so it follows that every problem in co-NP has polynomial time interactive proofs. But the full picture is much more interesting. It is easy to see that every polynomial time interactive proof translates into a polynomial-space algorithm: without a limitation on time, the Prover and Verifier can go through all possible choices of the random numbers, and once a run is completed, it can be erased except for adding 1 to the total number of successful or unsuccessful runs. It turns out that the protocol described above can be improved to show that *every problem in PSPACE has polynomial time interactive proofs* (Lund, Fortnow, Karloff and Nisan (1990); Shamir (1990)).

Exercise 14.4.1. *Prove that Problem 14.4.2 is NP-complete.*

14.5 How to verify proofs that keep the main result secret?

Our last two examples come from a field which causes much headache: editing scientific journals. These are "fun" examples and their purpose is to illuminate the logic of interactive proofs rather then propose real-life applications (although with the fast development of e-mail, electronic bulletin boards, on-line reviewing, and other forms of electronic publications—who knows?).

A typical impasse situation in journal editing is caused by the following letter: "I have a proof for P≠NP, but I won't send you the details because I am afraid you'd steal it. But I want you to announce the result." All we can do is to point out that the policy of the Journal is to publish results only together with their proofs (whose correctness is verified by two referees). There is no way to guarantee that the editor and/or the referee, having read the proof, does not publish it under his own name (unfortunately, this does seem to happen occasionally).

The result sketched in the Section 14.3, however, can resolve the situation. The author has to convince the editor that he has a proof, without giving any information about the details of the proof. Assume that the theorem itself can be stated with n characters, and we assume that we have an upper bound k for the proof (the proof has to be written out formally, with all details included). The set of pairs (x, y), where x is mathematical statement that has a proof of length at most $|y|$ in NP. (Note that we bound the length of the proof in unary, through the length of y rather than by the integer encoded by y. We may assume that y is of the form $11\ldots1$.)

So NP-completeness of the Hamiltonian cycle problem gives the following: for every mathematical statement x and string y one can construct a graph G such that x has a proof of length at most $|y|$ if and only if G has a Hamiltonian cycle. So the editor can come up with the following suggestion upon receiving the above letter: "Please construct the graph corresponding to Fermat's last theorem and the length of your proof, and prove me, using Blum's protocol, that this graph has a Hamiltonian cycle." This takes some interaction (exchange of correspondence), but in principle it can be done.

14.6 How to referee exponentially long papers?

Two Provers

But of course, the real difficulty with editing a Journal is to find referees, especially for longer papers. Who wants to spend months to read a paper of, say, 150 pages, and look for errors in the complicated formulas filling its pages? And, unfortunately, the devil is often hidden in the little details: a "-" instead of a "+" on the 79-th page may spoil the whole paper...

The results of Babai, Fortnow and Lund (1990) offer new possibilities. Let us give, first, an informal description of their method.

Assume that instead of one, we have two Provers. They are isolated, so that they cannot communicate with each other, only with the Verifier. It is clear that this makes it more difficult for them to cheat (see any crime drama), and so they may succeed in convincing the Verifier about some truth that they could not individually convince him about. Does this really happen? Babai et al. gave a protocol that allows two provers to give a polynomial time *interactive* proof for every property that has an "ordinary" proof of exponential length.

It is interesting to point out that this is the first result where we see that interaction really helps, without any unproven complexity hypothesis like P≠NP: it is known that there are properties whose shortest proofs grow exponentially with the input size of the instance.

The protocol of Babai et al. is a very involved extension of the protocol described in section 14.4, and we cannot even sketch it here.

Probabilistically Checkable Proofs

There is another interpretation of these results. It is commonplace in mathematics that sometimes making a proof longer (including more detail and explanation) makes it easier to read. Can this be measured? If a proof is written down compactly, without redundancy, then one has to read the whole proof in order to check its correctness. One way of interpreting the results mentioned above is that there is a way to write down a proof so that a referee can check the correctness of the theorem by reading only a tiny fraction of it. The proof becomes longer than necessary, but not much longer. The number of characters the referee has to read is only about the *logarithm* of the original proof length! To be precise, if the original proof has length N then the new proof can be checked by reading $O(\log N)$ characters; this is due to Feige, Goldwasser, Lovász, Safra and Szegedy (1991). So a 2000-page proof (and such proofs exist!), can be checked by reading a few lines! What a heaven for referees and editors!

This modified write-up of the proof may be viewed as an "encoding"; the encoding protects against "local" errors just like classical error-correcting codes protect against "local" errors in telecommunication. The novelty here is the combination of ideas from error-correcting codes with complexity theory.

Let us state the result formally. Let $\mathcal{A}$ be a randomized algorithm that for every $x, y \in \{0,1\}^*$ computes a bit $\mathcal{A}(x,y)$. We say that $\mathcal{A}$ is a *verifier* for a language $\mathcal{L} \subseteq \{0,1\}^2$ *with one-sided error* if for every positive integer n there exists a positive integer m such that for every $x \in \{0,1\}^n$,

(a) if $x \in \mathcal{L}$, then there exists a $y \in \{0,1\}^m$ such that $\mathcal{A}(x,y) = 1$ with probability 1;

(b) if $x \notin \mathcal{L}$, then for all $y \in \{0,1\}^m$, the probability that $\mathcal{A}(x,y) = 1$ is less than $1/2$.

We say that the verifier is (f,g,h)-*lazy*, if it works in $O(f(|x|))$ time, uses $O(g(|x|))$ random bits, and reads $O(h(|x|))$ bits of the string y.

Clearly NP consists of those languages which have an $(n^{\text{const}}, 0, n^{\text{const}})$-lazy verifier.

The following theorem, called the PCP Theorem, was proved by Arora, Lund, Motwani, Safra, Sudan and Szegedy in 1992. The proof is involved and not given here.

Theorem 14.6.1 (PCP theorem)**.** *For every language in NP there is an* $(n^{const}, \log n, 1)$*-lazy verifier.*

One should note that we do not need witnesses y longer than a polynomial in $|x|$. Indeed, the verifier uses at most $c_1 \log n$ bits, so the total number of outcomes for his coin flips is at most n^{c_1}. Furthermore, for any outcome of the coin flips, he reads at most c_2 bits of y. So there are at most $c_2 n^{c_1}$ bits in y that are ever read; we can dispose of the rest.

14.7 Approximability

A significant part of problems arising in the real world is NP-complete. Since we must find some kind of solutions even for these, several methods exist, see last comment of Chapter 4. In this section we discuss *approximate* solutions. These are hard to define for decision problems, where we expect a 0-1 answer. Because of this, here we deal with *optimization* problems. One part of the input specifies the set of *feasible solutions* and another gives an *objective function* that assigns a real value to each feasible solution. Our goal is to find a solution that minimizes (or maximizes) the objective function. E.g. the input is a graph, the feasible solutions are the proper colorings of its vertices and the objective function is the number of colors that we use.

For simplicity, the definitions here are only given in case the goal is to minimize the objective function, similar definitions can be given in case if the goal is to maximize it. For a given input, let OPT denote the value of the optimal solution, i.e. the infimum of the values of the objective function taken on a feasible solution. An algorithm is an $r(n)$-approximation, if for all n on all inputs of size n it returns a feasible solution (if exists) on which the value of the objective function is at most $r(n)\cdot$ OPT. An important special case is when $r(n)$ does not depend on n but is some constant c, in this case we say the algorithm is a c-approximation. A problem is $r(n)$-approximable if it has a polynomial time $r(n)$-approximation algorithm.

Example 14.7.1. In a graph we want to find the minimum vertex cover (such a vertex set that is incident to every edge). It is easy to give a 2-approximation: take a maximal M matching, and output the $2|M|$ endpoints of the edges of M. The maximality means that any edge is incident to this vertex set, so our solution is feasible. On the other hand, to cover the edges of M, we need at least $|M|$ vertices, thus OPT$\geq |M|$.

In most problems we do not know any, or just very weak, approximation results, many times an $O(\log n)$-approximation already makes us satisfied (e.g. we do not know anything better for the Blocking Set Problem (Problem 4.5.1) where we want to find a small set intersecting every set of a given a set system. However, a significant class of problems admit a c-approximation for some c, this class is denoted by *APX*.

For many NP-hard problems we can do even better. We say that a problem has a polynomial time approximation scheme (PTAS) if for every $\varepsilon > 0$ there is a $(1+\varepsilon)$-approximation. A problem has a fully polynomial time approximation scheme (FPTAS) if every $\varepsilon > 0$ there is a $(1+\varepsilon)$-approximation algorithm whose running time is a polynomial of n (the length of the input) and $1/\varepsilon$. (So the running time of a PTAS can be $n^{2^{1/\varepsilon}}$ but not of a FPTAS.) An intermediate, recently defined class is efficient

polynomial time approximation scheme (EPTAS) where the running time must be $f(\varepsilon) \cdot n^c$, so the power of n cannot depend on ε but $f(\varepsilon) = 2^{1/\varepsilon}$ is allowed.

We can also define the class of *APX-complete* problems. For the exact definition we would need a more sophisticated notion of reduction, so here we only mention the corollary that if a problem in APX is APX-complete, if there is a $c > 1$ such that if it were c-approximable, then this would imply P=NP. So if we suppose P≠NP, then APX-complete problems cannot have a PTAS.

Limits of approximability

The main application of the PCP Theorem is that it implies lower bounds on how well combinatorial optimization problems can be approximated. We illustrate this by the following. Recall that the clique number $\omega(G)$, the maximum number of nodes in a clique of G, is NP-hard to compute.

Theorem 14.7.1. *Suppose that there is a polynomial time algorithm that computes an approximation $f(G)$ of the clique number such that for every graph G, $\omega(G) \le f(G) \le 2\omega(G)$. Then P=NP.*

Proof. Let $\mathcal{L} \subseteq \{0,1\}^*$ be any NP-complete language, say 3-SAT. By the PCP Theorem, it has an $(n^{\text{const}}, \log n, 1)$-lazy verifier $\mathcal{A}$. For a fixed length $n = |x|$, let $\mathcal{A}$ use $k \le c_1 \log n$ random bits and read b bits of y.

For a given string $x \in \{0,1\}^*$, we construct a graph G_x as follows. The nodes of G_x will be certain pairs (z,u), where $y \in \{0,1\}^k$ and $u \in \{0,1\}^b$. To decide if such a pair is a node, we start the algorithm $\mathcal{A}$ with the given x and with z as its random bits. After some computation, it tries to look up b entries of y; we give it the bits $u_1, \dots, u_b$. At the end, it outputs 0 or 1; we take (z,u) as a node of G_x if it outputs 1.

To decide if two pairs (z,u) and (z',u') are adjacent, let us also remember which positions in y the algorithm $\mathcal{A}$ tried to read when starting with (x,u), and also when starting with (x,u'). If there is one and the same position read both times, but the corresponding bits of u and u' are different, we say that there is a *conflict*. If there is no conflict, then we connect (z,u) and (z',u') by an edge. Note that the graph G_x can be constructed in polynomial time.

Suppose that $x \in \mathcal{L}$. Then this has a proof (witness) $y \in \{0,1\}^m$. For every sequence $z = z_1 \dots z_k$ of random bits, we can specify a string $u \in \{0,1\}^b$ such that $(z,u) \in V(G_x)$, namely the string of bits that the algorithm reads when started with inputs x, y and z. Furthermore, it is trivial that between these there is no conflict, so these 2^k nodes form a clique. Thus in this case $\omega(G_x) \ge 2^k$.

Now suppose that $x \notin \mathcal{L}$, and assume that the nodes $(z,u), (z',u'), \dots, (z^{(N)}, u^{(N)})$ form a clique in G_x. The strings $z, z', \dots$ must be different; indeed, if (say) $z = z'$, then $\mathcal{A}$ tries to look up the same positions in y in both runs, so if there is no conflict, then we must have $u = u'$.

We create a string y as follows. We start with m empty positions. We run $\mathcal{A}$ with input x and random bits z, and we insert the bits of u in the b positions which the algorithm tries to look up. Then we repeat this with random bits z'; if we need to write in a position we already have a bit in, we do not have to change it since (z,u)

and (z', u') are connected by an edge. Similarly, we can enter the bits of $u'', \dots u^{(N)}$ in the appropriate positions without having to overwrite previously entered bits.

At the end, certain positions in y will be filled. We fill the rest arbitrarily. It is clear from the construction that for the string y constructed this way,

$$\Pr(\mathcal{A}(x, y) = 1) \geq \frac{N}{2^k},$$

and so by condition (b), $N < 2^{k-1}$. So if $x \notin \mathcal{L}$ then $\omega(G_x) < 2^{k-1}$.

Now if a polynomial time algorithm exists that computes a value $f(G_x)$ such that $\omega(G_x) \leq f(G_x) \leq 2\omega(G_x)$, then we have $f(G_x) \geq \omega(G_x) \geq 2^k$ if $x \in \mathcal{L}$, but $f(G_x) \leq 2\omega(G_x) < 2^k$ if $x \notin \mathcal{L}$, so we can decide whether $x \in \mathcal{L}$ in polynomial time. Since $\mathcal{L}$ is NP-complete, this implies that P=NP. □

The above method (using stronger forms of the PCP-theorem) have been significantly extended, e.g. the following strengthening of the above theorem was proved. If P≠NP, then there is no $n^{0.999}$-approximation for ω (Håstad, Zuckerman). Another example is that if P≠NP, then there is no $(c \log n)$-approximation for the Blocking Set Problem (Raz and Safra). Famous APX-complete problems include Maximal Cut (even restricted to 3-regular graphs), Vertex Cover and the Smallest (size) Maximal (unextendable) Matching.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullmann. *Design and Analysis of Computer Algorithms*. Addison-Wesley, New York, 1974.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Algorithms*. Mc Graw-Hill, New York, 1990.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, New York, 1979.

[5] Donald E. Knuth. *The Art of Computer Programming, I-III*. Addison-Wesley, New York, 1969-1981.

[6] L. A. Levin. Fundamentals of computing theory. Technical report, Boston University, Boston, MA 02215, 1996. Lecture notes.

[7] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, New York, 1981.

[8] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994. ISBN 0-201-53082-1.

[9] Christos H. Papadimitriou and K. Stieglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New York, 1982.

[10] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, New York, 1986.

[11] Robert Sedgewick. *Algorithms*. Addison-Wesley, New York, 1983.

[12] Klaus Wagner and Gert Wechsung. *Computational Complexity*. Reidel, New York, 1986.