# Complexity of Algorithms

*Lecture Notes, Spring 1999*

Peter Gács
Boston University

and

László Lovász
Yale University

# Contents


**0 Introduction and Preliminaries** **1**
- 0.1 The subject of complexity theory . . . 1
- 0.2 Some notation and definitions . . . 2

**1 Models of Computation** **3**
- 1.1 Introduction . . . 3
- 1.2 Finite automata . . . 4
- 1.3 The Turing machine . . . 6
- 1.4 The Random Access Machine . . . 16
- 1.5 Boolean functions and Boolean circuits . . . 21

**2 Algorithmic decidability** **29**
- 2.1 Introduction . . . 29
- 2.2 Recursive and recursively enumerable languages . . . 30
- 2.3 Other undecidable problems . . . 33
- 2.4 Computability in logic . . . 38

**3 Computation with resource bounds** **46**
- 3.1 Introduction . . . 46
- 3.2 Time and space . . . 46
- 3.3 Polynomial time I: Algorithms in arithmetic . . . 48
- 3.4 Polynomial time II: Graph algorithms . . . 53
- 3.5 Polynomial space . . . 58

**4 General theorems on space and time complexity** **61**
- 4.1 Space versus time . . . 67

**5 Non-deterministic algorithms** **68**
- 5.1 Non-deterministic Turing machines . . . 68
- 5.2 Witnesses and the complexity of non-deterministic algorithms . . . 70
- 5.3 General results on nondeterministic complexity classes . . . 72
- 5.4 Examples of languages in NP . . . 75
- 5.5 NP-completeness . . . 81
- 5.6 Further NP-complete problems . . . 85

**6 Randomized algorithms** **95**
- 6.1 Introduction . . . 95
- 6.2 Verifying a polynomial identity . . . 95
- 6.3 Prime testing . . . 99
- 6.4 Randomized complexity classes . . . 104

**7 Information complexity: the complexity-theoretic notion of randomness 108**
7.1 Introduction . . . 108
7.2 Information complexity . . . 108
7.3 The notion of a random sequence . . . 113
7.4 Kolmogorov complexity and data compression . . . 115

**8 Pseudo-random numbers 120**
8.1 Introduction . . . 120
8.2 Introduction . . . 120
8.3 Classical methods . . . 121
8.4 The notion of a psuedorandom number generator . . . 123
8.5 One-way functions . . . 126
8.6 Discrete square roots . . . 129

**9 Parallel algorithms 131**
9.1 Parallel random access machines . . . 131
9.2 The class NC . . . 134

**10 Decision trees 139**
10.1 Algorithms using decision trees . . . 139
10.2 The notion of decision trees . . . 142
10.3 Nondeterministic decision trees . . . 143
10.4 Lower bounds on the depth of decision trees . . . 147

**11 Communication complexity 151**
11.1 Communication matrix and protocol-tree . . . 151
11.2 Some protocols . . . 155
11.3 Non-deterministic communication complexity . . . 156
11.4 Randomized protocols . . . 160

**12 The complexity of algebraic computations 162**

**13 Circuit complexity 163**
13.1 Introduction . . . 163
13.2 Lower bound for the Majority Function . . . 164
13.3 Monotone circuits . . . 166

**14 An application of complexity: cryptography 168**
14.1 A classical problem . . . 168
14.2 A simple complexity-theoretic model . . . 168
14.3 Public-key cryptography . . . 169
14.4 The Rivest-Shamir-Adleman code . . . 171

# 0 Introduction and Preliminaries

## 0.1 The subject of complexity theory

The need to be able to measure the complexity of a problem, algorithm or structure, and to obtain bounds and quantitive 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 particualr 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 concrete 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. for the proof 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 concerining it; in particular, the decidability or undecidablity of most concrete 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.

## 0.2 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 $\Sigma$ 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 $\Sigma$ is denoted by $\Sigma^n$, the set of all words (including the empty word) over $\Sigma$ is denoted by $\Sigma^*$. A subset of $\Sigma^*$, 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 $\Sigma$ is given. In the *lexicographic ordering* of the elements of $\Sigma^*$, a word $\alpha$ precedes a word $\beta$ if either $\alpha$ is a prefix (beginning segment) of $\beta$ or the first letter which is different in the two words is smaller in $\alpha$. (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 up the natural numbers in the binary number system.

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

0.1 **Exercise** Is it true that $1 + 2 + \cdots + n = O(n^3)$? Can you make this statement *sharper*? ♢

# 1 Models of Computation

## 1.1 Introduction

In this section, we will treat the concept of "computation" or algorithm. This concept is fundamental for 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 program in the Pascal (or any other) 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*, whci 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 programable computers. The essence of this model is a central part that is bounded (with a structure independent of the input) and an infinite storage (memory). (More exactly, 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 inves-

tigations. 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 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 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 $N$.

## 1.2 Finite automata

A *finite automaton* is a very simple and very general computing device. All we assume 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 $\Sigma$,

— an *output alphabet*, which is another finite set $\Sigma'$, and

— a set $\Gamma$ of internal states, which is also finite.

To describe a finite automaton, we need to specify, for every input $a \in \Sigma$ 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.

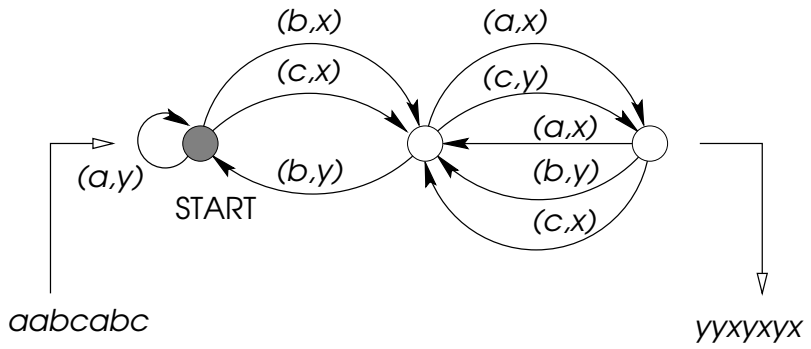



Figure 1.1: A finite automaton

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 $\Sigma, \Sigma', \Gamma$ are finite sets, $\alpha : \Sigma \times \Gamma \to \Sigma'$ and $\omega : \Sigma \times \Gamma \to \Gamma$ are arbitrary mappings, and $\text{START} \in \Gamma$.

**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 $\Sigma'$ contains the *blank symbol* $*$; in other words, we allow that the automaton does not give an output at certain steps.

2. Your favorite PC can be modelled 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, floppy drives 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'll 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 $\Gamma$, and there is an edge labelled $(a, b)$ 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).

(1.1) **Example** 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

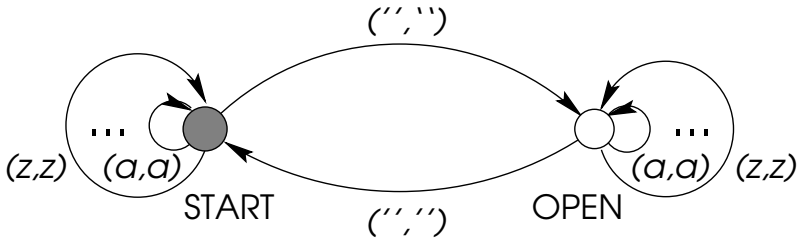



Figure 1.2: An automaton correcting quotation marks

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). ♢

1.1 **Exercise** 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. If we get past the first bit of one of the inputs numbers, a special symbol • is passed to the automaton instead of a bit; the input stops when two consecutive • symbols are occur. ♢

1.2 **Exercise** 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 YES if the number is divisible by 7 and NO if it is not. ♢

1.3 **Exercise** (a) For a fixed positive integer $n$, construct a finite automaton that reads a word of length $2n$, and its last output is YES if the first half of the word is the same as the second half, and NO otherwise. (b) Prove that the automaton must have at least $2^n$ states. ♢

1.4 **Exercise** Prove that there is no finite automaton that, for an input in $\{0,1\}*$ starting with a "1", would decide if this binary number is a prime. ♢

## 1.3 The Turing machine

### 1.3.1 The notion of a 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 $\Sigma$ 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 field 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 $\Gamma$. 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);

Of course, the heads carry out these 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, $\Sigma$ and $\Gamma$ are finite sets, $* \in \Sigma$ $START, STOP \in \Gamma$, and $\alpha, \beta, \gamma$ are arbitrary mappings:

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

Here $\alpha$ specifiess the new state, $\beta$ gives the symbols to be written on the tape and $\gamma$ specifies how the heads move.

In what follows we fix the alphabet $\Sigma$ 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. Thus, the input of a $k$-tape Turing machine is an ordered $k$-tuple, each element of which is a word in $\Sigma^*$. 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. 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.

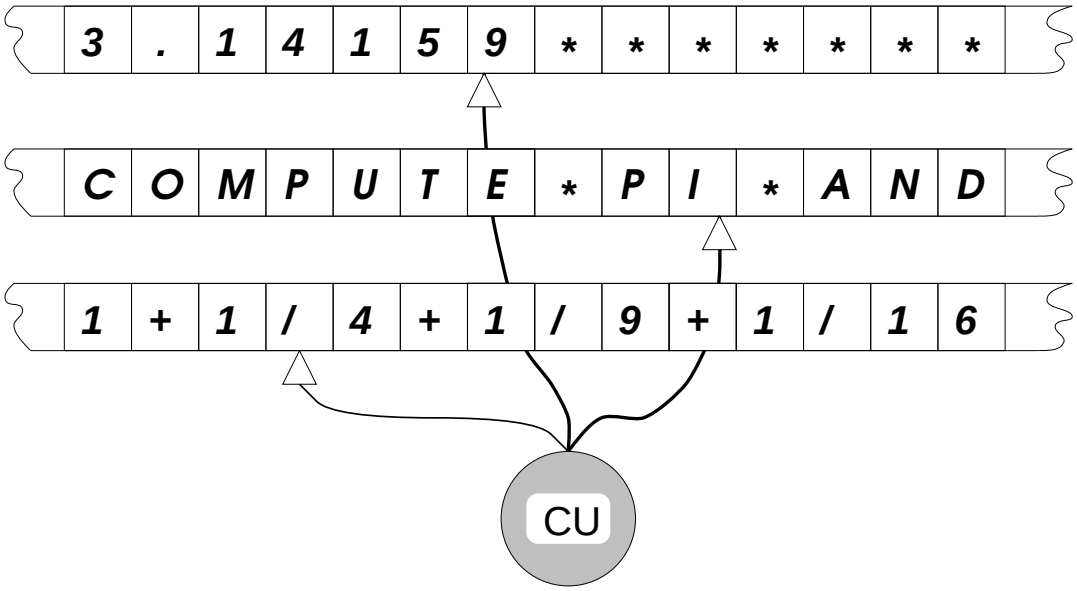



Figure 1.3: A Turing maching with three tapes

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 $\Sigma_0$. (Another solution would be to reserve a symbol for signalling "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.

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. 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.

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

(a) $x_1 \dots x_m \mapsto x_m \dots x_1$.

(b) $x_1 \dots x_m \mapsto x_1 \dots x_m x_1 \dots x_m$.

(c) $x_1 \dots x_m \mapsto x_1 x_1 \dots x_m x_m$.

(d) for an input of length $m$ consisting of all 1's, the binary form of $m$; for all other inputs, for all other inputs, it never halts.

$\diamondsuit$

1.6 **Exercise** 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 $f \circ g$. $\diamondsuit$

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

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

### 1.3.2 Universal Turing machines

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 programable 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 $\Sigma$, there is a word (program) $p$ with which $T$ simulates $S$.

(1.1) **Theorem** *For every number $k \geq 1$ and every alphabet $\Sigma$ 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 its head" the states of $S$). 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 $\Sigma$ 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 $\Gamma_S \setminus \{\mathrm{STOP}\}$ with a word of length $r$ over the alphabet $\Sigma_0^*$. Let the "code" of a given position of machine $S$ be the following word:

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

where $g \in \Gamma_S$ is the given state of the control unit, and $h_1, \ldots, h_k \in \Sigma$ 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 $k+1$; while on tape $k+2$, we write a state of machine $S$, initially the name of the START state.

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 $\alpha, \beta, \gamma$ 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 typical 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.

Now return to the proof of Theorem 1.1. We can get rid of the $(k+2)$-nd tape easily: its contents (which is always just $r$ cells) will be placed on cells $-1, -2, \ldots, -r$. 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 right half there is a 1 if the corresonding 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. ∎

1.9 **Exercise** Write a simulation of a Turing machine with a doubly infinite tape by a Turing machine with a tape that is infinite only in one direction. ♢

1.10 **Exercise** 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. ♢

1.11 **Exercise** 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. ♢

### 1.3.3 More tapes versus one tape

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

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-STATE" 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 | | |

(1.2) **Theorem** *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 $S$ halts in finitely many steps on input $x$ if and only if $T$ halts on input $x$, and at halt, the same is written on the last tape of $S$ as on the tape of $T$. Further, if $S$ makes $N$ steps then $T$ makes $O(N^2)$ steps.*

**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 $2k+2j-1$ 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, let us mark by a 0 the first even-numbered cell of the empty ends of the tapes. 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, $T$ "keeps in its head" 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 wich direction they will move. Starting backwards, for each 1 found in an odd cell, it can rewrite correspondingly the cell before 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, 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 $2ki$ 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 "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. ∎

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., $x_1 \ldots x_n$ is a palindrome iff $x_i = x_{n-i+1}$ for all $i$. Let us analyze the task of recognizing a palindrome.

(1.3) **Theorem** *(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 $k \leq j \leq 2k$. 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 deamon sitting on this, and recording the state of the central unit any time the head passes between these fields. At the end of the computation, we get a sequence $g_1 g_2 \dots g_t$ of elements of $\Gamma$ (the length $t$ of the sequence may be different for different inputs), the *$j$-log* of the given input. The key to proof is the following observation.

**Lemma.** 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 $k \leq j \leq 2k$. 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 indentical 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

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

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

$$2^k - 2(k+1)|\Gamma|^{m-1}. \tag{1.4}$$

So 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 deamons record at least $(k+1)m$ moves, so the computation takes at least $(k+1)(m+1)$ steps.

It is easy to check that the choice $m = n/(6\log|\Gamma|)$ makes (1.4) positive, and so we have found an input for which the computation takes at least $(k+1)m > n^2/(6\log|\Gamma|)$ steps. ■

1.12 **Exercise** 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. ♢

*(1.13) **Exercise** 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! (Hennie-Stearns)] ♢

1.14 **Exercise** 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.

♢

*(1.15) **Exercise** 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 writeable 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, terminated by a blank.

Find a problem that can be solved more efficiently on an online Turing machinw 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.] $\diamondsuit$

1.16 **Exercise** 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. [Hint: Store on tape 1, with each symbol of the two-dimensional tape, an arbitrary number identifying its original position and the numbers identifying its parent and children.]

(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.

$\diamondsuit$

## 1.4 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 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 assigne 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 fintie 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 machine.

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.

(1.1) **Example** [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. ♢

(1.2) **Example** [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. ♢

(1.3) **Example** [Multiple branching] Let $p_0, p_1, \ldots, p_r$ be indices of program rows, and suppose that we know that the contents 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 p0;
x[i]:=x[i]-1:
IF x[i]≤0 THEN GOTO p1;
x[i]:=x[i]-1:
⋮
IF x[i]≤0 THEN GOTO pr.
```

(Attention must be paid when including this last program segment in a program, since it changes the content of $xi$. 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 p0;
x[-1]:=x[-1]-1:
IF x[-1]≤0 THEN GOTO p1;
x[-1]:=x[-1]-1:
⋮
IF x[-1]≤0 THEN GOTO pr.
```

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. ♢

Now we show that the RAM and Turing machines 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 $x_1 \ldots x_n$ 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 $n$, $x_1, \ldots, x_n$ into the registers $x[0], \ldots, x[n]$, or we could assign to the sequence $x_1 \ldots x_n$ 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 consider the first interpretation first.

(1.4) **Theorem** *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 $\Gamma = \{1, \ldots, r\}$, 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 the state of the control unit will be determined by where we are in the program.

Our program will be composed of parts $Q_{ij}$ simulating the action of the Turing machine when in state $i$ and reading symbol $j$ ($1 \leq i \leq r-1, 0 \leq j \leq 2$) and lines $P_i$ that jump to $Q_{i,j}$ if the Turing machine is in state $i$ and reads symbol $j$. Both are easy to realize. $P_i$ is simply

`GOTO` $Q_{i,x[1]}$`;`

for $1 \leq i \leq i-1$; the program part $P_r$ consists of a single empty program line. The program parts $Q_{ij}$ are only a bit more complicated:

`x[x[1]]:=` $\beta(i,j)$`;`
`x[1]:=x[1]+2`$\gamma(i,j)$`;`
`GOTO` $P_{\alpha(i,j)}$`;`

The program itself looks as follows.

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

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 anything in at most $O(N)$ registers, so in each step of the Turing machine we work with numbers of length $O(\log N)$. ■

Another interpretation of the input of the Turing machine is, as mentioned above, to view the input as a single natural number, and to enter it into the RAM as such. This number $a$ is thus in register $x[0]$. In this case, what we can do is to compute the digits of $a$ with the help of a simple program, write these (deleting the 1 in the first position) into the registers $x[0], \ldots, x[n-1]$, and apply the construction described in Theorem 1.4.

(1.5) **Remark** In the proof of Theorem 1.4, we did not use the instruction $x[i] := x[i] + x[j]$; this instruction is needed when computing the digits of the input. 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 running time 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.

(1.6) **Theorem** *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 content of registers $x[i]$ (in binary, and with sign if it is negative). We could represent the content of all registers (representing, say, the content 0 by the symbol "*"). 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 wich possibly the whole tape must be searched. Since the length of the tape is $N$, the total number of steps is $O(N^2)$. ■

1.17 **Exercise** 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;

$\diamondsuit$

1.18 **Exercise** 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 $K = \max\{|a_0|, \ldots, |a_n|\}$. $\diamondsuit$

1.19 **Exercise** 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.
$\diamondsuit$

1.20 **Exercise** 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,..., followed by a symbol # and then by 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 $R(u, \langle p, x\rangle) = R(p, x)$. $\diamondsuit$

1.21 **Exercise** [Pointer Machine.] After having seen finite-dimensional tapes and a tree tape, we may want to consider a machine with a more general directed graph its storage medium. Each cell $c$ has a fixed number of edges, numbered $1, \ldots, r$, leaving it. When the head scans a certain cell it can move to any of the cells $\lambda(c, i)$ $(i = 1, \ldots, r)$ reachable from it along outgoing edges. Since it seems impossible to agree on the best graph, we introduce a new kind of elementary operation: to change the structure of the storage graph locally, around the scanning head. Arbitrary transformations can be achieved by applying the following three operations repeatedly (and ignoring nodes that become isolated): $\lambda(c, i) :=$ New, where New is a new node; $\lambda(c, i) := \lambda(\lambda(c, j))$ and $\lambda(\lambda(c, i)) := \lambda(c, j)$. A machine with this storage structure and these three operations added to the usual Turing machine operations will be called a Pointer Machine.

Let us call RAM' the RAM from which the operations of addition and subtraction are omitted, only the operation $x[i] := x[i] + 1$ is left. Prove that the Pointer Machine is equivalent to RAM', in the following sense.

For every Pointer Machine there is a RAM' program computing for each input the same output. If the Pointer Machine has running time $N$ then the RAM' runs in $O(N)$ steps.

For every RAM' program there is a Pointer Machine computing for each input the same output. If the RAM' has running time $N$ then the Pointer Machine runs in $O(N)$ steps.

Find out what Remark 1.5 says for this simulation. $\diamondsuit$

## 1.5 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*

*variables*, *Boolean variables* or *bits*. 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 $x_{ij}$ $(1 \leq i < j \leq N)$ is 1 if $i$ and $j$ are connected and 0 if they are not. The value of the function $f(x_{12}, x_{13}, \ldots, x_{n-1,n})$ is 1 if there is a Hamilton cycle in $G$ and 0 if there is not. Our problem is the computation of the value of this (implicitly given) Boolean function.

There are only four one-variable Boolean functions: the identically 0, 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 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 of XOR)

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

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:

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

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

$$\begin{aligned} x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) \\ x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) \end{aligned}$$

and

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

The DeMorgan identities connect negation with conjunction and disjunction:

$$\begin{aligned} \overline{x \wedge y} &= \overline{x} \vee \overline{y}, \\ \overline{x \vee y} &= \overline{x} \wedge \overline{y}. \end{aligned}$$

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

(1.1) **Lemma** *Every Boolean function is expressible as a Boolean polynomial.*

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

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

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 "$\wedge$". (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 "$\vee$". 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. As we see, a nontrivial disjunctive normal form is always satisfiable.

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

(1.2) **Example** 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 $x?y : z = (x \wedge y) \vee (\neg x \wedge z)$. It is possible to construct the disjunctive normal form of an arbitrary Boolean function by the repeated application of this example. $\diamondsuit$

Interchanging the role of the operations "$\wedge$" and "$\vee$", 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 found 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. We have seen that 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.

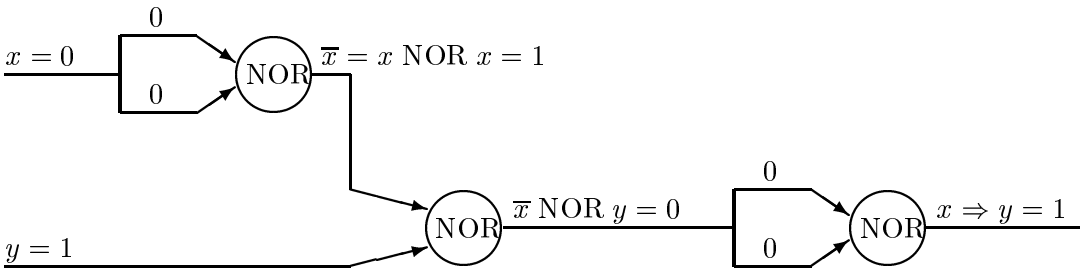



Figure 1.7: A NOR circuit computing $x \Rightarrow y$, with assignment on edges

(1.3) **Example** [Majority Function] Let $f(x_1, \ldots, x_n) = 1$ if and only if at least half of the variables are 1. $\diamondsuit$

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 that does not contain any directed cycle (i.e. is *acyclic*). 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. those of its nodes without outgoing edges, will be called *output nodes*. (In what follows, we will deal most frequently with circuits that have a single output node.)

To each node $v$ of the graph that is not a source, i.e. which has some degree $d = d^+(v) > 0$, a "gate" is given, i.e. 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; its *depth* is the maximal length of paths leading from input nodes to output nodes.

Every circuit $H$ determines a Boolean 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 to 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 values at the sinks give the *output* of the computation. We will say about the function defined this way that it is *computed* by the circuit $H$.

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

(1.4) **Example** A NOR circuit computing $x \Rightarrow y$. We use the formulas

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

If the states of the input lines of the circuit are $x$ and $y$ then the state of the output line is $x \Rightarrow y$. The assignment can be computed in 3 stages, since the longest path has 3 edges. See Figure 1.7. $\diamondsuit$

(1.5) **Example** 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 1.1) 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

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

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

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 DeMorgan 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 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.

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 wich 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.

We can consider each Boolean circuit as an algorithm serving to compute some Boolean function. It can be seen immediately, however, that circuits "can do" 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.

(1.6) **Theorem** *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, let us 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 $v[r, g', p+\varepsilon]$ and $w[r, p+\varepsilon, h']$ ($g' \in \Gamma$, $h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$). 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:

$$v[0, g, p] = 1 \quad \Leftrightarrow \quad g = START \wedge p = 0,$$

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

$$w[0, p, h] = 1 \Leftrightarrow ((p < 0 \vee p \geq n) \wedge h = *) \vee (0 \leq p \leq n-1 \wedge h = x_p).$$

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

$$\begin{aligned} v[t+1, g, p] = 1 \quad &\Leftrightarrow \quad \exists g' \in \Gamma,\ \exists h' \in \Sigma : \alpha(g', h') = g \wedge v[t, g', p - \gamma(g', h')] = 1 \\ &\quad \wedge w[t, p - \gamma(g', h'), h'] = 1. \\ w[t+1, p, h] = 1 \quad &\Leftrightarrow \quad (\exists g' \in \Gamma,\ \exists h' \in \Sigma : v[t, g', p] = 1 \wedge w[t, p, h'] = 1 \wedge \beta(g', h') = h) \\ &\quad \vee (w[t, p, h] = 1 \wedge \forall g' \in \Gamma : w[t, g', p] = 0). \end{aligned}$$

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. ∎

(1.7) **Remark** Our construction of a universal Turing machine in Theorem 1.1 is inefficient and unrealistic. For most commonly used transition functions $\alpha, \beta, \gamma$, 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 $\alpha, \beta, \gamma$ and then proceed as before. $\diamondsuit$

1.23 **Exercise** 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. $\diamondsuit$

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

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

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

$\diamondsuit$

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

1.26 **Exercise** 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. $\diamondsuit$

1.27 **Exercise** Prove that for every Boolean 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. $\diamondsuit$

1.28 **Exercise** Prove that the Boolean polynomials are in one-to-one correspondence with those Boolean circuits that are trees. $\diamondsuit$

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

1.30 **Exercise** Universal circuit. For each $n$, construct a Boolean circuit whose gates have indegree $\leq 2$, which has size $O(2^n)$ with $2^n + n$ inputs and which is universal in the following sense: that 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 table is given by $p$. [Hint: use the decoder circuit of Example 1.5.] $\diamondsuit$

1.31 **Exercise** Circuit size. The gates of the Boolean circuits in this exercise are assumed to have indegree $\leq 2$.

(a) Prove the existence of a constant $c$ such that for all $n$, there is a Boolean function such that 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 $O(2^n/n)$.

$\diamondsuit$

# 2 Algorithmic decidability

## 2.1 Introduction

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 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. According to this, 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. Gödel proved 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 that 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 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 7 on Information Complexity.)

## 2.2 Recursive and recursively enumerable languages

Let $\Sigma$ 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. (We have seen in the previous section that we can assume without loss of generality that 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 $\Sigma_0^*$, 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 $\Sigma_0^*$. Therefore even though we define these notions only over $\Sigma_0^*$, we sometimes use them in connection with function 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.

(2.1) **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. We will see some concrete languages that are non-recursive. $\diamondsuit$

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 $\mathcal{L}$: $\mathcal{L} = \{f(w_0), f(w_1), \ldots\}$, when $\Sigma_0^* = \{w_0, w_1, \ldots\}$. Here, the elements of $\mathcal{L}$ do not necessarily occur in increasing order and repetition is also allowed.

We give an alternative definiton of recursively enumerable languages through the following lemma.

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

**Proof** Let $\mathcal{L}$ be recursively enumerable. We can assume that it is nonempty. Let $\mathcal{L}$ be the range of $f$. We prepare a Turing machine which on input $x$ calculates $f(y)$ in increasing order of $y \in \Sigma_0^*$ and it stops whenever it finds a $y$ such that $f(y) = x$.

On the other hand, let us assume that $\mathcal{L}$ contains the set of 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, when the natural number $i$ is its input it simulates $T$ on input $x$ which is the $(i - \lfloor\sqrt{i}\rfloor)^2$-th word of $\Sigma_0^*$, for $i$ steps. If the simulated $T$ stops then $T_0$ ouputs $x$. Since every word of $\Sigma_0^*$ 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 assumes 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.

(2.3) **Lemma** *Every recursive language is recursively enumerable.*

**Proof** This is clear if the language $\mathcal{L}$ is empty. We can change the Turing machine that decides $f$ to output the input if the intended output is 1, and to output some arbitrary fixed $a \in \mathcal{L}$ if the intended output is 0. ∎

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

(2.4) **Theorem** *A language $\mathcal{L}$ is is recursive iff 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, it is 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. ∎

Now, we will show that there are languages that are recursively enumerable (and hance rather explicit), but not recursive.

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

(2.5) **Theorem** *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 follows from Lemma 2.3. We prove the second statement, for simplicity, 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 iff $x \notin \mathcal{L}_T$. 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 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.

(2.6) **Theorem** *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 (0.2) (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, however simple it is, 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 $\Sigma$, $\Gamma$ (where, as before, the elements of $\Gamma$ are coded by words over the set $\Sigma_0$), and the table of the functions $\alpha, \beta, \gamma$.

(2.7) **Corollary** *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 $x$. ∎

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

**Proof** For a given 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 emptyness, we cannot decide any other property $P$ of of $\mathcal{L}_T$ either if the empty language has it and $\Sigma_0^*$ has not, or vice versa. Even a more general negative result is true. We call a property of a language **trivial** if either all languages have it or none.

(2.9) **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. ∎

## 2.3 Other undecidable problems

First we mention a problem of geometrical nature. A **prototile**, or domino is a square shape and has a natural number written on each side. A **tile** is an exact copy of some prototile. (To avoid trivial solutions, let us require that the copy must be positioned in the same orientation as the prototile, without rotation.) A **kit** is a finite set of prototiles, one of which is a distinguished "initial domino". Given a kit $K$, a **tiling** of whole plane with $K$ (if it exists) assigns to each position with integer coordinates a tile which is a copy of a prototile in $K$, in such a way that

— neighbor dominoes have the same number on their adjacent sides;

— the initial domino occurs.

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 number on each side) and also a kit with which this is impossible (e.g., a single square that has a different number on each side). It is, however, a surprising fact that the tiling problem is algorithmically undecidable!

For the exact formulation, let us describe each kit by a word over $\Sigma_0 = \{0, 1, +\}$, e.g. in such a way that we write up the numbers written on the sides of the prototiles in binary, separated

by the symbol "+", beginning at the top side, clockwise, then we join the so obtained number 4-tuples starting with the initial domino. (The details of the coding are not interesting.) Let $\mathcal{L}_{\mathrm{TLNG}}$ [resp. $\mathcal{L}_{\mathrm{NTLNG}}$] the set of codes of those kits with which the plane is tileable [resp. not tileable].

(2.1) **Theorem** *The tiling problem is undecidable, i.e. the language $\mathcal{L}_{\mathrm{TLNG}}$ is not recursive.*

Accepting, for the moment, this statement, according to Theorem 2.4, either the tiling or the nontiling kits must form a language that is not recursively enumerable. Which one? For the first look, we might think that $\mathcal{L}_{\mathrm{TLNG}}$ is recursive: the fact that the plane is tileable by a kit can be proved by supplying the tiling. This is, however, not a finite proof, an actually the truth is just the opposite:

(2.2) **Theorem** *The language $\mathcal{L}_{\mathrm{NTLNG}}$ is recursively enumerable.*

Taken together with Theorem 2.1, we see that $\mathcal{L}_{\mathrm{TLNG}}$ can not even be recursively enumerable.

In the proof of Theorem 2.2, the following lemma will play important role.

(2.3) **Lemma** *The plane is tileable by a kit if an only if for all $n$, the square $(2n+1)\times(2n+1)$ is tileable by it with the initial domino is in its 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 converges to infinity. We will construct a tiling of the plane. Without loss of generality, we can assume that the center of each square is at the origin. Let us consider first the $3 \times 3$ square centered at the origin. This is tiled by the kit somehow in each $N_i$. Since it can only be tiled in finitely many ways, there is an infinite number of $N_i$'s in which it is tiled in the same way. With an appropriate thinning of the sequence $N_i$ we can assume that this square is tiled in the same way in each $N_i$. These nine tiles can already be fixed.

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

Every tile covering some integer vertex unit square of the plane 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 tiles dominoes, the tiles will be correctly matched in the final tiling, too. ∎

**Proof of Theorem 2.2** Let us construct a Turing machine doing the following. For a word $x \in \Sigma_0^*$, it first of all decides whether it codes a kit (this is easy); if not then it goes into an infinite cycle. If yes, then with this set, it tries to tile one after the other the squares $1\times1$, $2\times2$,

| | | | | |
|---|---|---|---|---|
| $*$<br><br><br>$*$ | 1<br><br><br>1 | 2<br><br><br>2 | 1<br>$g_1$<br>$*g_2$ | $*g_1$<br>$g_1$<br>$*$ |
| $*$<br><br><br>$*$ | 1<br><br><br>1 | 2<br>$g_2$<br>$*g_1$ | $*g_2$<br>$g_2$<br>$*$ | $*$<br><br><br>$*$ |
| $*$<br><br><br>$*$ | 1<br>$g_1$<br>$*$START | $*g_1$<br>$g_1$<br>$*$ | $*$<br><br><br>$*$ | $*$<br><br><br>$*$ |
| $*$<br>N N<br>$*$ | $*$START<br>N P<br>START$*$ | $*$<br>P P<br>$*$ | $*$<br>P P<br>$*$ | $*$<br>P P<br>$*$ |

Figure 2.1: Tiling resulting from a particular computation

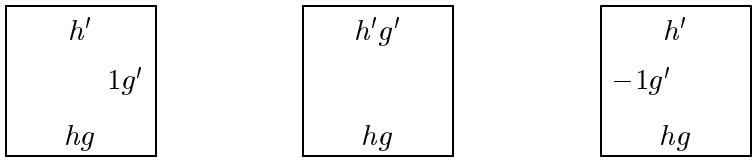
$3 \times 3$, etc. For each concrete square, it is decidable in a finite number of steps, whether it is tileable, since the sides can only be numbered in finitely many ways by the numbers occurring in the kit, and it is easy to verify whether among the tilings obtained this way there is one for which every tile comes from the given kit. If the machine finds a square not tileable by the given kit then it halts.

It is obvious that if $x \in \mathcal{L}_{\mathrm{TLNG}}$, 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 $x \in \mathcal{L}_{\mathrm{NTLNG}}$, i.e. $x$ codes a kit that does not tile the plane then according to Lemma 2.3, for a large enough $k$ already the square $k \times k$ is not tileable either, and therefore the Turing machine stops after finitely many steps. Thus, according to Lemma 2.2, the language $\mathcal{L}_{\mathrm{NTLNG}}$ is recursively enumerable. ■

**Proof of Theorem 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 start on the empty input. This is, however, undecidable due to Corollary 2.7, so it is also undecidable whether the constructed kit can tile the plane.

In defining the kit, we will write symbols, rather than numbers on the sides of the tiles; these are easily changed to numbers. For simplicity, assume $k = 1$. It is also convenient to assume (and achievable by a trivial modification of $T$) that the machine $T$ is in the starting state only before the first step.

Let us subdivide the plane into unit squares whose centers are points with integer coordinates. Assume that $T$ does not halt on empty input. Then from the machine's computation, let us construct a tiling as follows (see Figure 2.1): if the content of the $p$-th cell of the machine after step $q$ is symbol $h$ then let us write symbol $h$ on the top side of the square with center




$\alpha(g,h) = g'$ $\beta(g,h) = h'$ $\gamma(g,h) = 1$ | $\alpha(g,h) = g'$ $\beta(g,h) = h'$ $\gamma(g,h) = 0$ | $\alpha(g,h) = g'$ $\beta(g,h) = h'$ $\gamma(g,h) = -1$

Figure 2.2: Tiles for currently scanned cell

point $(p,q)$ an on the bottom side of the square with center point $(p,q+1)$. If after step $q$, the head scans cell $p$ and the control unit is in state $g$ then let us write the symbol $g$ on the top side of the square with center point $(p,q)$ an on the bottom side of the square with center point $(p,q+1)$. If the head, in step $q$, moves right [left], say from cell $p-1$ $[p+1]$ to cell $p$ and is in state $g$ after the move then let us write symbols $-1,g$ $[1,g]$ on the left [right] side of the square with center $(p,q)$ and on the right [left] side of the cell with center $(p-1,q)$ $[(p+1,q)]$. Let us write symbol "N" on the vertical edges of the squares in the bottom row if the edge is to the left of the origin and the letter "P" if the edge is to the right of the origin. Let us reflect the tiling with respect to the $x$ axis, reversing also the order of the labels on each edge. Figure 2.1 shows the construction for the simple Turing machine which steps right on the empty tape and writes on the tape 1's and 2's alternatingly.

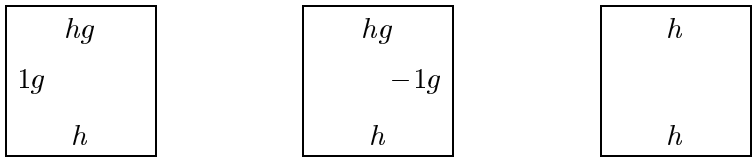
Let us determine what kind of tiles occur in the tiling obtained. In the upper half-plane, there are basically four kinds. If $q > 0$ and after step $q$, the head rests on position $p$ then the square with center $(p,q)$ comes from one of the prototiles shown in Figure 2.2. If $q > 0$ and after step $q-1$ the head scans position $p$ then the square with center $(p,q)$ comes from one of the prototiles on Figure 2.3a-b. If $q > 0$ and the head is not at position $p$ either after step $q-1$ or after step $q$ then the square with position $(p,q)$ has simply the form of Figure 2.3c. Finally, the squares of the bottom line are shown in Figure 2.4. We obtain the tiles occurring in the lower half-plane by reflecting the above ones across the horizontal axis and reversing the order of the labels on each edge.

Now, figures 2.2–2.4 can be constructed from the description of the Turing machine; we thus arrive at a kit $K_T$ whose initial domino is the middle domino of Figure 2.4. The above reasoning shows that if $T$ runs an infinite number of steps on empty input then the plane can be tiled with this kit. Conversely, if the plane can be tiled with the kit $K_T$ then the initial domino covers (say) point $(0,0)$; to the left and right of this, only the two other dominos of Figure 2.4 can stand. Moving row-by-row from here we can see that the covering is unique and corresponds to a computation of machine $T$ on empty input.

The only not completely obvious case is when a tile of the form shown in the first two examples of Figure 2.4 occurs. We must show that it can occur indeed only if the head moves onto the corresponging tape square. The symbols $\pm 1$ in front of $g$ exclude all other possibilities.

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

(2.4) **Remark** The tiling problem is undecidable even if we do not distinguish an initial




a) b) c)

Figure 2.3: Tiles for previously scanned or unscanned cell

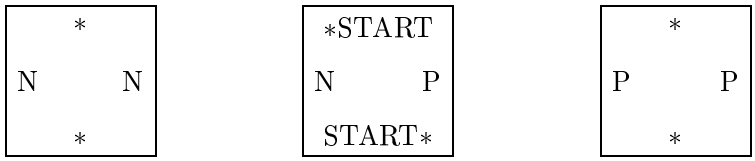



Figure 2.4: Tiles for the bottom line

domino. But the proof of this 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:

(2.5) **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. ♢

(An equation is called Diophantine if we are looking for its 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 MAT'IYASEVICH in 1970. It was found therefore that the problem of solvability of Diophantine equations is algorithmically undecidable.

We mention also an important problem of 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.

(2.6) **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 $\beta$. Is $\beta$ in the subgroup generated by $\alpha_1, \ldots, \alpha_n$? $\diamondsuit$

Finally, 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 $P$:

(2.7) **Contractability of polyhedra** Can a given polyhedron be contracted to a single point (continuously, staying within itself)? $\diamondsuit$

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 teared". 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 "contractable". For example, a triangle, taken with the area inside it, is contractable. The perimeter of the triangle (the union of the three sides without the interior) is not contractable. (In general, we could say that a polyhedron is contractable 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 contractability turns out to be algorithmically undecidable.

## 2.4 Computability in logic

### 2.4.1 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 ninetieth 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 $\varphi$, an other sentence $\psi$ called its **negation**.

(2.1) **Example** 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. $\diamondsuit$

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 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 **F** is called a *theorem* of the theory **F**.

(2.2) **Example** Here is a simple theory $T_1$ based on the language $L_1$ of the above Example 2.1. 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 $a < b$. $\diamondsuit$

A theory is called **consistent** if for no sentence can both it 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 2.2 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 $a, b$ are natural numbers and $a \geq b$.

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 desireable goal with 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 a desirable algorithmic property, as shown by the theorem below: this shows that if there are no (logically) undecidable sentences in a theory then the truth of all sentences (with respect to that theory) is algorithmically decidable.

(2.3) **Theorem** *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 a 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 r.e. set of integers that is not recursive. An arithmetical theory $\mathcal{T}$ is called **minimally adequate** if for numbers $n$, the theory contains a sentence $\varphi_n$ 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 subsection, 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:

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

**Proof** If the theory were complete then, according to Theorem 2.3 it would give a procedure to decide all sentences of the form $n \in L$, which is impossible. ∎

(2.5) **Remark** 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. ♢

(2.6) **Remark** Historically, Gödel's theorems preceded the notion of computability by 3-4 years. ♢

### 2.4.2 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 *interpretation* 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.

(2.7) **Example** Consider the language with the 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 $\forall x \forall y\, f(x, y) = f(y, x)$. $\diamondsuit$

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 2.7, 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. This 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 subsection. 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.

(2.8) **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 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

$$\begin{aligned} (t_1 = u_1 \wedge \cdots \wedge t_n = u_n) &\Rightarrow f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n), \\ (t_1 = u_1 \wedge \cdots \wedge t_n = u_n) &\Rightarrow (P(t_1, \ldots, t_n) \Leftrightarrow P(u_1, \ldots, u_n)) \end{aligned}$$

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 $A \Rightarrow \forall x\, B$.

(2.9) **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. $\diamondsuit$

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

(2.10) **Theorem** *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 in the proof system* **P** *of $T$ from the axioms of* **B**.

**A simple theory of arithmetic; 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).

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

(2.11) **Theorem** *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.

(2.12) **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 up a formula $Q(n)$ saying $N \Rightarrow (n \in L)$. There is a proof for "$n \in L$" in $N$ if and only if there is a proof for $Q(n)$ in $\mathcal{P}$ If we had a decision procedure for $\mathcal{P}$ we could decide, $Q(n)$; since we cannot, there is no decision procedure for $\mathcal{P}$. ∎

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

2.2 **Exercise**

(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.

♢

In the exercises below, we will sometimes use the following notion. A function $f$ defined on a subset of $\Sigma_0^*$ 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.

2.3 **Exercise** 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. $\diamondsuit$

2.4 **Exercise** 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. $\diamondsuit$

2.5 **Exercise** 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. $\diamondsuit$

2.6 **Exercise** 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. $\diamondsuit$

2.7 **Exercise** Let $\#E$ denote the number of elements of the set $E$. 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? $\diamondsuit$

2.8 **Exercise** A rooted tree is a set of "nodes" in wich 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. $\diamondsuit$

2.9 **Exercise** 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. $\diamondsuit$

2.10 **Exercise** Show that there is a kit of dominoes with the property that it tiles the plane but does not tile it periodically. $\diamondsuit$

2.11 **Exercise** Let $T$ be a one-tape Turing machines that never overwrites a nonblank symbol by a blank one. 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. $\diamondsuit$

*(2.12) **Exercise** 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 2.11. Use the kit assigned to it by the proof of the tiling problem. Again, we will only consider the prototiles associated with the upper 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 $\Sigma_0$, we make a new prototile $P_h$ by adding add $h$ 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 $P'$.] $\diamondsuit$

2.13 **Exercise** 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 a 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. $\diamondsuit$

2.14 **Exercise** 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. $\diamondsuit$

2.15 **Exercise** Our proof of Gödel's theorem does not seem to give a specific sentence $\varphi_T$ 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. $\diamondsuit$

# 3 Computation with resource bounds

## 3.1 Introduction

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 Chapter 4). 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 our constructions in section 1.3 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 os 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 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 contraints. 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'll 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 solvable in polynomail space (but most probably not in polynomial time) are *games* like chess or GO. We give a detailed description of this connection.

We end with a briefer discussion of other typical complexity classes.

## 3.2 Time and space

Let us fix some finite alphabet $\Sigma$, inluding 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 and output tape and $k \geq 1$ work tapes. At start, there is a word in $\Sigma_0^*$ written on the input tape.

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

Similarly, the function $\text{space}_T(n)$ is defined as the maximum number, over all inputs of length $n$, of all cells on all tapes to which the machine writes. (This way, the cells occupied by the input are not counted in the space requirement.) We assume that $\text{space}_T(n) \geq 1$ (the machine has some output).

A Turing machine $T$ is called *polynomial*, if there is a polynomial $f(n)$ such that $\text{time}_T(n) = O(f(n))$. This is equivalent to saying that there is a constant $c$ such that the time demand of $T$ is $O(n^c)$. We can define exponential Turing machines similarly (for which the time demand is $O(2^{n^c})$ for some $c > 0$), and also Turing machines 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 $\mathcal{L} \subseteq \Sigma_0^*$.

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 $\text{DTIME}(f(n))$ 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 laguage has *space complexity at most* $f(n)$, and also the language classes $\text{DSPACE}(f(n))$ and PSPACE (polynomial space).

(3.1) **Remark** 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. Section 3.3 contains a theorem that provides such examples. $\diamondsuit$

(3.2) **Remark** 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 $\text{DTIME}(f(n))$, $\text{DSPACE}(f(n))$, 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 is $x \cdot y$ 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. $\diamondsuit$

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 $n$ steps on numbers with at most $k$ bits". Similarly, the space requirement is best characterized by saying that "the machine stores most $n$ 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 occurring 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.

## 3.3 Polynomial time I: Algorithms in arithmetic

### 3.3.1 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 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 $a$ (say). 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 $r < a$, tis 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).

(3.1) **Lemma** *The Euclidean algorithm takes polynomial time. More exactly, it carries out of* $O(\log a + \log b)$ *arithmetical operations carried out 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 one of them 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 not only gives the value of the greatest common divisor, but also delivers 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 $p'a + q'b$.

(3.2) **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 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! It means that given integers $a, b$ and $m$, where $0 \leq a, b \leq m - 1$ and $b \neq 0$, we can compute 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 solution is to apply the Euclidean algorithm to compute the greatest common divisor of the numbers $b, m$. 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 $b(ap) \equiv a \pmod{m}$. So the quotient $x$ we are looking for is the remainder of the product $ap$ 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 as the number of binary digits of $n$), so of course, this number is not computable in polynomial time. The situation changes, however, if we want to carry out the exponentiation modulo $m$: then $a^b$ is also a residue class modulo $m$, and hence it 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.

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

**Algorithm** Let us write $b$ in binary:

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

where $0 \le r_1 < \cdots < r_k$. It is obvious that $r_k \le \log b$ and therefore $k \le \log b$. Now, the numbers $a^{2^t} \pmod{m}$ for $0 \le t \le \log b$ 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 $m$. ∎

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

### 3.3.2 Gaussian elimination

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

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 written down 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

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

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.

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 $1 \leq i \leq t$, 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 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 $i$ column for all $i = t+2, \ldots, n$, in order to get 0's in the elements $(t+1, t+2), \ldots, (t+1, n)$. These subtractions do not change 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 that 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 $(1, 1), \ldots, (n, n)$, i.e., we do not have to permute

the rows and columns. Let $(a_{ij}^{(k)})$ $(1 \leq i, j \leq n)$ be the matrix obtained after $k$ iterations. Let us denote the elements in the main diagonal of the final matrix, for simplicity, by $d_1, \ldots, d_n$ (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 $k+1 \leq i, j \leq n$, 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, $\det(D^{(k)}) = d_{kk}^{(k-1)}$.

(3.5) **Lemma**

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

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

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

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.

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 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 don't 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 quite easy to select such a large prime (see the section on randomized algorithms). 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 be 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.)

(3.6) **Remark** 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. $\diamondsuit$

## 3.4 Polynomial time II: 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'll group our examples around two algorithmic methods: searching a graph and augmenting paths.

### 3.4.1 How is a graph given?

### 3.4.2 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 look if there is an edge between the nodes of $T$ and the rest

of the node. 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 $T$ grom the graph and proceed recursively.

It is costumary to call the nodes of $T$ *labelled*. The usual way to look for an edge between a labelled and an unlabelled node is to look at the labelled 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 labelled node goes to an unlabelled node, then we don't need to investigate this node at any other later stage (since only an unlabelled node can become labelled, 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 labelled but not scanned.

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

**b. Breadth-First-Search and shortest paths.** Specializing the order in which we investigate edges out of the labelled nodes, we get special search algorithms that are good for solving different kinds of graph problems. Perhaps the simplest such specialization is *depth-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 the problem in the 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 unlabelled 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.) 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 labelled with a 0. In the course of the algorithm we maintain a set $T$ of vertices, each of which we have confidently labelled 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 3-connected components.**

**d. Depth-First-Search and planarity.**

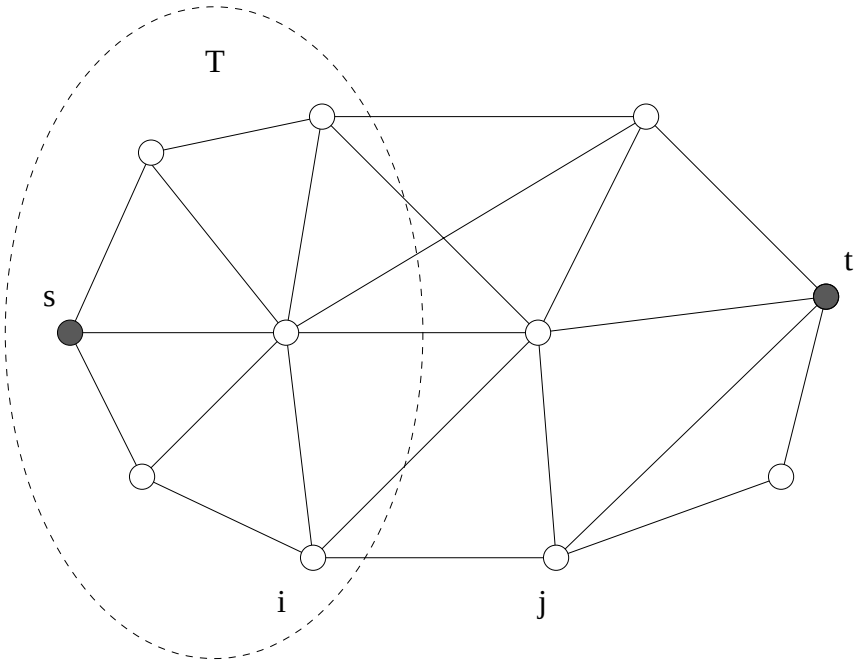



Figure 3.1: Dijkstra's shortest path algorithm.

### 3.4.3 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 subset of the edges having mutually no endpoint in common). A matching is *perfect* if it mmets 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 alterning path is *augmenting* if it starts and ends with an edge not in $M$, and moreover, its endpoints are not incindent with any edge of $M$. (An augmenting path has necessarily odd length.)

(3.1) **Lemma** *A matching in $G$ is maximum 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 coponent 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.

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).

**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 during this lecture).

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,

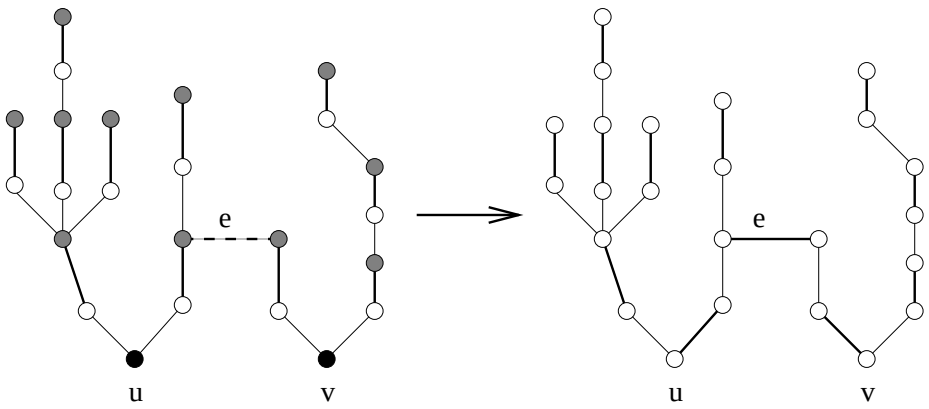



Figure 3.2: Extending a bipartite graph matching

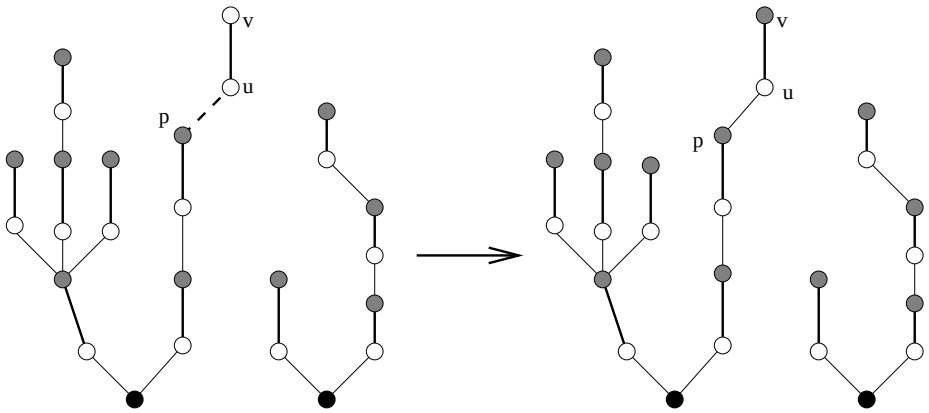



Figure 3.3: Extending an alternating forest

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 it must 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. Simple considerations enable to reduce the running time to $O(n^3)$. It is possible, but quite difficult, to reduce it further to $O(n^{2.5})$.

### 3.5 Polynomial space

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 CU, position of heads, contents of tape) is repeated, then we are in cycle).

For example, the space requirement of the trivial graph coloring algorithm treated in 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 Hamilton 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 $\Sigma$ (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 it 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

[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.

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

— — 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 labelled by the starting position. The children of the root are labelled by the positions that can be reached from the starting position by a legal move. Similarly, each node $i$ of the tree is labelled by a position $x_i$ in the game, and its children are labelled 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 moree 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.

**Algorithm to find an optimal move.** Roughly speaking, we search the game-tree in a depth-first-search manner, and label positions "winning' and "loosing".

At a given step, the machine analyzes all possible continuations of some partial game $x_0, x_1, \ldots, x_k$. 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 legal move from this position, that this position is terminal, and the algorithm marks it "winning" or "loosing" 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 loosing. 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 labelled, and that tells is who wins in this game.

3.1 **Exercise** 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 $(F_k, F_{k-1})$? $\diamondsuit$

3.2 **Exercise** (a) Show that for computing $a/b \pmod{m}$, we don't need that $m$ is a prime; it suffices to assume that the greatest common divisor of $b$ and $m$ is 1.

(b) More generally, show that given integers $a$, $b$ and $m$, we can find find in polynomial time an integer $x$ such that $bx \equiv a \pmod{m}$, or conclude that no such integer exists. $\diamondsuit$

3.3 **Exercise** 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 $a_1, \ldots, a_n$ are the row vectors of $A$ then $|\det(A)| \leq |a_1| \cdots |a_n|$ (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).] $\diamondsuit$

3.4 **Exercise** Let $A$ be an $n \times n$ matrix with 2's in the main diagonal, 1's below it and 0's above it. Show that if Gaussian elimination without simplification is applied to this matrix then after $k$ iterations, each of the denominators will be $2^{2^k-1}$. $\diamondsuit$

# 4 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 recognizing $L$ 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).

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

**Proof** For simplicity, let us also assume that $T$ has a single work tape (the proof would be similar for $k$ tapes). We can assume that $c = 1/p$ 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 $p$ 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 wich 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) makes only a $p$th of the number of steps of $T$ and decides the same language. ∎

*(4.1) **Exercise** 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 $\text{time}_S(n) \leq c \cdot \text{time}_T(n) + n$ (here, we allow the extension of the alphabet.♢

4.2 **Exercise** 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 $\mathcal{L} \in \text{DTIME}(f(n))$ for a language then there is a constant $k$ (depending on the language) that $\mathcal{L} \in \text{DSPACE}(k \cdot f(n))$. (If extending the alphabet is allowed and $f(n) > n$ then $\text{DSPACE}(k \cdot f(n)) = \text{DSPACE}(f(n))$ and thus it follows that $\text{DTIME}(f(n)) \subset \text{DSPACE}(f(n))$.) 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 $c \cdot f(n)^k m^{f(n)}$ where $m$ is the size of the alphabet.

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 $\mathcal{L} \in \text{DSPACE}(f(n))$ then $\mathcal{L} \in \text{DTIME}((m+1)^{f(n)})$.

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

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

**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 $\mathcal{L} \in \text{DTIME}(f(n))$. 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 $n$ using 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 work tapes, 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 $1 \ldots 1$ of length $n$ and we want as output a sequence $1 \ldots 1$ 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:

(4.3) **Lemma**

(a) *To every well-computable function $f(n)$, there is a fully time-constructible function $g(n)$ such that $f(n) \leq g(n) \leq \mathrm{const} \cdot f(n)$.*

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

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

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.

(4.4) **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.

4.3 **Exercise** 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 $t'(n)$. $\diamondsuit$

4.4 **Exercise** 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 in time $S'(n)$. $\diamondsuit$

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.

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

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

Thus, there is a recursive function $f$ with

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

moreover, there is even one with

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

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

1. Claim 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.

2. 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, $|p| \cdot \varphi(h(n))$. 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.

(4.6) **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 $|y| \leq |x|$ and $\tau(x, y) \leq h(|x| - |y|)$.

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 time $h(|x_i| - |y_i|)$.

This last step itself is already too much if $|y_i| \leq k$; therefore we make a list of all pairs $(x_i, y_i)$ with $|y_i| \leq k$ (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 $k < |y| \leq |x|$ for "fastness" and this needs only $(m^2 n + 1)h(n-k-1)$ (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 $y$ accepts the language $\mathcal{L}$ on on the machine $T_0$ (i.e. stops for all $\Sigma_0^*$ 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 don't 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 $n$).

## 4.1 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 4.2.

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 speeded 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 (inverting). 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).

# 5 Non-deterministic algorithms

## 5.1 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, $\Sigma$ and $\Gamma$ are finite sets, $* \in \Sigma$, START, STOP $\in \Gamma$ (so far, everything is as with a deterministic Turing machine) and

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

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 consisting of $t$ steps, with this input, 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 don't 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 arbitarily 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$(f(n))$, co$-$NSPACE$(f(n))$ are defined analogously.

(5.1) **Remark**

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

2. 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 object of a later section. In contrast to non-deterministic Turing machines, randomized Turing machines model computing processes that are practically important.

♢

(5.2) **Theorem** *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 $\mathcal{L}^{\#}$ contains, in an appropriate encoding, of a sequence of *configurations*, or *instantaneous descriptions*, as they follow in time. Each configuration shows the internal state and the symbol found in each tape cell at the given instant, as well as the positions of the tape heads. The set $\mathcal{L}^{\#}$ is obviously recursive since given two configurations, it can be decided whether the second one can be obtained in one computation step of $T$ from the first one. 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 $x$. The range of values of the recursive function defined by $S$ is obviously just $\mathcal{L}$. ■

## 5.2 Witnesses and the complexity of non-deterministic algorithms

Let us fix a finite alphabet $\Sigma_0$ 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 $\mathcal{L}$.

Let $f$ and $g$ be two functions with $g(n) \geq n$. We 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 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 $y$.)

(5.1) **Theorem**

(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 5.2, 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 computes (deterministically) the value of $f(|x|)$ and writes this many 1's on the second tape. Then it writes symbol & at the end of $x$ and makes a transition into its only state in which its behavior is nondeterministic. While staying in this state it writes a word $y$ of lengt at most $f(|x|)$ after the word $x\&$. This happens as follows: while it reads a 1 on the second tape it has $|\Sigma_0| + 1$ legal moves: either it writes some symbol of the alphabet on the first tape, moves right on the first tape and left on the second tape or it writes nothing and makes a transition into state START2.

From state START2, on the first tape, the machine moves the head on the starting cell, erases the second tape 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 $y \in \Sigma_0^*$ of length at most $f(|x|)$ for which $S$ accepts word $x\&y$, i.e. if $x \in \mathcal{L}$. The running time of this computation is obviously at most $O(f(|x|)) + g(|x| + 1 + f(|x|)) = O(g(|x| + 1 + f(x)))$. ∎

(5.2) **Corollary** *For an arbitrary language $\mathcal{L} \subset \Sigma_0^*$, 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.*

(5.3) **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 $|x|$. $\diamondsuit$

We denote the class of languages having the property stated in Corollary 5.2 by NP. The languages $\mathcal{L}$ for which $\Sigma_0^* \setminus \mathcal{L}$ 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 soluton. 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.

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 *witness language* $\mathcal{L}$. 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

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

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.

## 5.3 General results on nondeterministic complexity classes

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

(5.1) **Theorem** *Let $f$ be a well-computable function. Then*

(a) $\mathrm{NTIME}(f(n)) \subset \mathrm{DSPACE}(f(n))$

(b) $\mathrm{NSPACE}(f(n)) \subset \bigcup_{c>0} \mathrm{DTIME}(2^{cf(n)})$.

**Proof**

(a): 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 where we are in the trieout of of the cases.

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, 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 choose first always the lexicographically first action (we fix some ordering of $\Sigma$ and $\Gamma$, 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 $O(f(n))$).

(b): 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 inputs. 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 number of configurations is at most $|\Gamma|(2f(n) + 1)m^{2f(n)+1} = 2^{O(f(n))}$. Every configuration can be coded by a word of length $O(f(n))$ over $\Sigma$.

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.

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

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

**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 above graph $G$; 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. An *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.

(5.3) **Lemma** *Suppose that a directed graph $G$ is given on the set of 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 storage at most $O(qt)$, 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 $t \leq q$ 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 wether 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 a 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 $y$. 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 corrollary of Savitch's theorem shows that this would not lead to any new notion:

(5.4) **Corollary** PSPACE = NPSPACE.

## 5.4 Examples of languages in NP

### 5.4.1 Graphs

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.

(5.1) GRAPH-CONNECTIVITY Certificate: a set of $\binom{n}{2}$ paths, one for each pair of nodes. $\diamondsuit$

(5.2) GRAPH NON-CONNECTIVITY Certificate: a proper subset of the set of nodes that is not connected by any edge to the rest of the nodes. $\diamondsuit$

(5.3) GRAPH 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 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:

(5.4) **Theorem** *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. In this case, it is enough to check whether every edge is 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.) $\diamondsuit$

(5.5) NON-PLANARITY Let us review the following facts.

1. Let $K_5$ be the graph obtained by connecting five nodes in every possible way. This graph is also called a "complete pentagon". Let $K_3^3$ be 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:

(5.6) **Kuratowski's Theorem** *A graph is nonplanar if and only if it contains a subgraph that is a topological version 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. ♢

(5.7) Existence of perfect matching A certificate is the perfect matching itself. ♢

(5.8) 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 $G$ consisting with bipartition classes $U$ and $W$. Recall the following theorem:

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

Hence, if in some bipartite graph there is no perfect matching then this can be certified either by noting that $U$ and $W$ have different cardinality, or by a subset of $W$ 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.

(5.10) **Theorem** *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.

(5.11) Existence of a Hamiltonian cycle
Certificate: 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.

(5.12) 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. ♢

Most of the properties listed above, up to (and including) the non-existence of a perfect matching, 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).

### 5.4.2 Arithmetic and algebra

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 a word in $\{0,1\}^*$ (representing the number in binary). We start with the problem of deciding whether a natural number is a prime.

(5.13) COMPOSITENESS OF AN INTEGER Certificate of compositeness: a proper divisor. ♢

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

(5.15) **Theorem** *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 also check the condition

$$a^{n-1} \equiv 1 \pmod{n} \tag{5.16}$$

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

$$a^m \not\equiv 1 \pmod{n} \qquad (1 \leq m < n-1). \tag{5.17}$$

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 (5.16) holds then the smallest $m = m_0$ violating (5.17) (if there is any) is a divisor of $n-1$. It is also easy to see that then (5.17) 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

(5.17) 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 (5.17) 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). Now only the problem remains to check whether this is a prime decomposition indeed, 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 recursion 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

$$k_1 + \cdots + k_t < k + t.$$

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

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

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

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:

(5.18) 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. $\diamondsuit$

The complementary language is also in NP:

(5.19) NONEXISTENCE OF A BOUNDED DIVISOR 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 certificateof the primality of every prime factor.

It is not known whether the problem of compositeness (even less, the existence of a bounded divisor) is in P. Extending the notion of algorithms and using random numbers, it is decidable in polynomial time about a number whether it is a prime (see the section on randomized algorithms). At the same time, the corresponding search problem (the search for a proper divisor), or, equivalently, deciding the existence of bounded divisors, is significantly harder; for this, a polynomial algorithm was not yet found even when the use of random numbers is allowed. ♢

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.

(5.20) REDUCIBILITY OF A POLYNOMIAL OVER THE RATIONAL FIELD
Certificate: 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. ♢

### 5.4.3 Systems of linear inequalities

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:

(5.21) EXISTENCE OF SOLUTION The solution offers itself as an obvious witness of solvability but we must be careful: we must be convinced 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. ♢

(5.22) NONEXISTENCE OF SOLUTION Witnesses for the non-existence of solution can be found using the following fundamental theorem known from linear programming:

(5.23) **Farkas 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 other system of inequalities given in the lemma (the nonnegative coefficients) is a witness of the nonexistence of a solution. ♢

(5.24) 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 more difficult here. ♢

It is not known wheter 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 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 (as shown by experience) 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 observatons 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 $\mathrm{NP} \cap \mathrm{co{-}NP}$ 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.

— With 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 $\mathrm{P} = \mathrm{NP}$ holds); but still, one can prove a certain exact property of these problems that shows that they are hard. We will turn to this later.

— 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 whether there is a perfect matching in a certain graph then we can search for 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 (at least, in some sense) whether a number is a prime was not applicable to the problem of finding a proper divisor.

— A number of NP-problems has a related *optimization problem* which is easier to state, even if it is not an NP-problem by its form. For example, instead of the general matching problem, it is easier to say that the problem is to find out 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* $R(G)$ belonging to $G$ is the smallest $N > 0$ for which it is the case 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 says that such a finite number exists, which is non-trivial.) 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 algoritm, which is, unfortunately, the best known, goes over all colorings of the edges of the $N$-node complete graph, and the number of these is $2^{N(N-1)/2}$.

## 5.5 NP-completeness

We say that a language $\mathcal{L}_1 \subset \Sigma_1^*$ is **polynomially reducible** to a language $\mathcal{L}_2 \subset \Sigma_2^*$ 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

$$x \in \mathcal{L}_1 \Leftrightarrow x \in \mathcal{L}_2.$$

It is easy to verify from the definition that this relation is transitive:

(5.1) **Proposition** *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 $\{0, 1\}$.

(5.2) **Proposition** *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 word "completeness" suggests that the solution of the decision problem of a complete language contains, in some sense, the solution to the decision problem 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.

(5.3) **Proposition** *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 5.3) 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.

(5.4) Satisfiability Problem For a given Boolean polynomial $f$, decide whether it is satisfiable. We consider the problem, in general, in the case when the Boolean polynomial is a conjunctive normal form. $\diamondsuit$

We can consider each conjunctive normal form as a word over the alphabet consisting of the symbols "$x$", "0", "1", "+", "$\neg$", "$\wedge$" 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.

(5.5) **Theorem** **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:

$$\begin{aligned} x[n, g] \quad & (0 \leq n \leq N,\ g \in \Gamma), \\ y[n, p] \quad & (0 \leq n \leq N,\ -N \leq p \leq N), \\ z[n, p, h] \quad & (0 \leq n \leq N,\ -N \leq p \leq N,\ h \in \Sigma). \end{aligned}$$

If a legal computation of the machine $T$ is given then let us assign to these variables the following values: $x[n, g]$ is true if after the $n$-th step, the control unit is in state $g$; $y[n, p]$ is true if after the $n$-th step, the head is on the $p$-th tape cell; $z[n, p, h]$ is true if after the $n$-the step, the $p$-th tape cell contains symbol $h$. The variables $x, y, z$ obviously determine the computation of the Turing machine.

However, not each 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:

$$\bigvee_{g \in \Gamma} x[n, g] \qquad (0 \leq n \leq N);$$

and it should not be in two states:

$$\neg x[n, g] \vee \neg x[n, g'] \qquad (g, g' \in \Gamma,\ 0 \leq n \leq N).$$

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 $h_1 \cdots h_n$ and finally the symbol 1 on cell 0:

$$\begin{aligned} z[0, i-1, h_i] &= 1 \qquad (1 \leq i \leq n) \\ z[0, i-1, *] &= 1 \qquad (i < 0 \text{ or } i > n) \\ z[N, 0, 1] &= 1. \end{aligned}$$

We must further express the computation rules of the machine, i.e., that for all $g, g' \in \Gamma$, $h, h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$ and $-N \leq p \leq N$ we have

$$(x[n, g] \wedge y[n, p] \wedge z[n, p, h]) \Rightarrow \neg(x[n+1, g'] \wedge y[n+1, p+\varepsilon] \wedge z[n+1, p, h'])$$

and that where there is no head the tape content does not change:

$$\neg y[n, p] \Rightarrow (z[n, p, h] \Leftrightarrow z[n+1, p, h]).$$

For the sake of clarity, the the last two formulas are not in conjunctive normal form but it is easy to bring them to such form. 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 $h_1 \cdots h_n$. 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.

(5.6) **Theorem** *The language* $k$-SAT *is* NP*-complete.*

**Proof** Let $B$ be a Boolean circuit with inputs $x_1, \ldots, x_n$ (a conjunctive normal form is a special case of this). We will find a 3-normal form that is satisfiable if and only if the function computed by $B$ is not identically 0. Let us introduce a new variable $y_i$ for each node $i$ of the circuit. The meaning of these variables is that in a satisfying assignment, these are the values computed by the corresponding nodes. Let us write up all the restrictions for $y_i$. For each input node $i$, with node variable $y_i$ and input variable $x_i$ we write

$$y_i \Leftrightarrow x_i \qquad (1 \leq i \leq n).$$

If $y_i$ is the variable for an $\wedge$ node with inputs $y_j$ and $y_k$ then we write

$$y_i \equiv y_j \wedge y_k.$$

If $y_i$ is the variable for a $\vee$ node with inputs $y_j$ and $y_k$ then we write

$$y_i \equiv y_j \vee y_k.$$

If $y_i$ is the variable for a $\neg$ node with input $y_j$ then we write

$$y_i \equiv \neg y_j.$$

Finally, if $y_i$ is the output node then we add the clause

$$y_i.$$

Each of these statements involves only three variables and is therefore expressible as a 3-normal form. The conjunction of all these is satisfiable if and only if $B$ is satisfiable. ∎

it is natural to wonder at this point why have we considered just 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.)

(5.7) **Theorem** *The language* 2*-*SAT *is in* P*.*

**Proof** Let $B$ be a 2-normal 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 $x_i^\varepsilon$ to node $x_j^\delta$ 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.

(5.8) **Lemma** *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 5.8** Let us note first that if an assignment of values satisfies formula $B$ and $x_i^\varepsilon$ is "true" in this assignment then every $x_j^\delta$ is "true" to which an edge leads from $x_i^\varepsilon$: 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 components, which is a contradiction. ∎

(5.9) **Theorem** *The language* SAT-3 *is* NP*-complete.*

**Proof** Let $B$ be a Boolean formula of the variables $x_1, \dots, x_n$. 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, $y_j^1 \Rightarrow y_j^2 = \neg y_j^1 \vee y_j^2$.) 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. ∎

## 5.6 Further NP-complete problems

### 5.6.1 NP-complete problems for graphs

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 about a problem that it 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:

(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 $A_i$? $\diamondsuit$

We have met a special case of this problem, the Blocking Set Problem for the edges of a bipartite graph. This special case was polynomial time solvable. In contrast to this, we prove:

(5.2) **Theorem** *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 5.9 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 Satisfiablity Problem to the case when the input is a conjunctive normal form in which every variable occurs at least once negated and at least one 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):

(5.3) Covering problem: Given a system $\{A_1, \ldots, A_m\}$ 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 $S$? $\diamondsuit$

According to the discussion above, this problem is NP-complete already even when each of the given subsets has at most 3 elements. has only 2 elements, the problem becomes polynomially solvable, as the following exercise shows:

5.1 **Exercise** Prove that the covering problem, if every set in the set system is restricted to have at most 2 elements, is reducible to the following matching problem: given a graph $G$ and a natural number $k$, is there a matching of size $k$ in $G$? $\diamondsuit$

For set systems, the following pair of problems is also important:

(5.4) $k$-PARTITION PROBLEM: Given a system $\{A_1, \ldots, A_m\}$ of subsets of a finite set $V$ and a natural number $k$. Can a subsystem of $k$ sets $\{A_{i_1}, \ldots, A_{i_k}\}$ be selected that gives a **partition** of the underlying set (i.e. consists of disjoint sets whose union is the whole set $V$)? $\diamondsuit$

(5.5) PARTITION PROBLEM: Given a system $\{A_1, \ldots, A_m\}$ of subsets of a finite set $S$. Can a subsystem (of any size) be selected that gives a partition of the underlying set? $\diamondsuit$

If all the $A_i$ 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.

(5.6) **Theorem** *The $k$-partition problem and the partition problem are* NP*-complete.*

**Proof** We reduce the 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 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 as most $2^3 = 9$-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 the sets of our new set system be 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 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 Set Partition Problem is just the perfect matching problem and can therefore be solved in polynomial time. The argument above shows that, on the other hand, the Set 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:

(5.7) **Theorem** *The coloring of graphs with three colors is an* NP*-complete problem.*

**Proof** Let us be given a 3-form $B$; 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 5.1).

The following remark, which is 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*

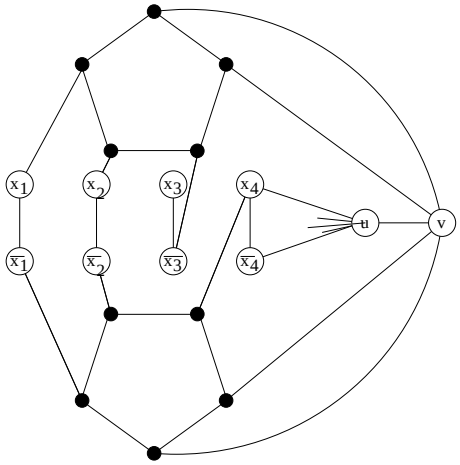



Figure 5.1: The graph whose 3-coloring is equivalent to satisfying the expression $(\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$

*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. Let us color red those (negated or unnegated) variables that are "true", and blue the others. Let us 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 two of them.

(5.8) Independent node set problem: Given a graph $G$ and a natural number $k$, are there $k$ independent nodes in $G$? ♢

(5.9) **Theorem** *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 5.2 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. Formulating it as a yes-no question:

(5.10) **Remark** The independent vertex set problem (and similarly, the Blocking set problem) are only NP-complete 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. ♢

From the NP-completeness of the Independent 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

(5.11) **Corollary** *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.

(5.12) **Corollary** *The problem of deciding whether a graph has a clique of size $k$ is* NP-*complete.*

Very many other important combinatorial and graph-theoretical problems are NP-complete:

— Does a given graph have a Hamiltonial 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.

### 5.6.2 NP-complete problems in arithmetic and algebra

A number of NP-complete problems is 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.

(5.13) LINEAR DIOPHANTINE INEQUALITIES Given a system $Ax \leq b$ of linear inequalities with integer coefficients, decide whether it has a solution in integers. $\diamondsuit$

(Recall that the epithet "Diophantine" indicates that we are looking for the solution among integers.)

(5.14) **Theorem** *The solvability of a Diophantine system of linear inequalities is an* NP*-complete problem.*

**Proof** Let a 3-form $B$ be given over the variables $x_1, \ldots, x_n$. Let us write up the following inequalities:

$$\begin{aligned}
0 \leq x_i \leq 1 \qquad & \text{for all } i, \\
x_{i_1} + x_{i_2} + x_{i_3} \geq 1 \qquad & \text{if } x_{i_1} \vee x_{i_2} \vee x_{i_3} \text{ is in } B, \\
x_{i_1} + x_{i_2} + (1 - x_{i_3}) \geq 1 \qquad & \text{if } x_{i_1} \vee x_{i_2} \vee \overline{x}_{i_3} \text{ is in } B, \\
x_{i_1} + (1 - x_{i_2}) + (1 - x_{i_3}) \geq 1 \qquad & \text{if } x_{i_1} \vee \overline{x}_{i_2} \vee \overline{x}_{i_3} \text{ is in } B, \\
(1 - x_{i_1}) + (1 - x_{i_2}) + (1 - x_{i_3}) \geq 1 \qquad & \text{if } \overline{x}_{i_1} \vee \overline{x}_{i_2} \vee \overline{x}_{i_3} \text{ is in } B.
\end{aligned}$$

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:

(5.15) SUBSET SUM PROBLEM: Given natural numbers $a_1, \ldots, a_m$ and $b$. Does the set $\{a_1, \ldots, a_m\}$ have a subset whose sum is $b$? (The empty sum is 0 by definition.) $\diamondsuit$

(5.16) **Theorem** *The subset sum problem is* NP*-complete.*

**Proof** We reduce the set-partition problem to the subset sum problem. Let $\{A_1, \ldots, A_m\}$ be a family of subsets of the set $S = \{0, \ldots, n-1\}$, 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

$$a_{i_1} + \cdots + a_{i_k} = \sum_j d_j q^j.$$

Since the representation of the integer $b$ with respect to the number base $q$ is unique, it follow 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 of the algorithms will be smaller in comparison with the length of the input.

(5.17) **Theorem** *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 5.14 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 $b \in T_p$ holds.

We must see yet that this simple algorithm is polynomial. This follows immediately from the observation that $T_p \subset \{0, \ldots, \sum_i a_i\}$ 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**.

(5.18) **Remark** 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 Hamilton cycle with minimum cost (the cost of a Hamilton cycle is the sum of costs of its edges);

— the Steiner problem (under the previous conditions, find a connected graph containing given vertices with minimum cost);

— 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, Hamilton circuits or legal colorings). ♢

(5.19) **Remark** 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:

PRIMALITY. Is a given natural number a prime?

BOUNDED DIVISOR. Does a given natural number $n$ have a proper divisor not greater than $k$?

GRAPH ISOMORPHISM. Are two given graph isomorphic?

The primality problem is probably in P (this is proved using an old number-theoretical conjecture, the so-called Generalized Riemann Hypothesis; see the section on "Randomized algorithms"). For the problems of bounded divisor and isomorphism, it is rather expected that they are neither in P nor NP-complete. ♢

(5.20) **Remark** 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 is 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 occuring 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 the section on cryptography. ♢

5.2 **Exercise** A *quantified Boolean expression* is a Boolean expression in which the quantifiers $\forall x$ and $\exists x$ can also be used. Prove that the problem of deciding about a given quantified Boolean expression whether it is true is in PSPACE. $\diamondsuit$

5.3 **Exercise** Let $f$ be a length-preserving one-to-one function over binary strings computable in polynomial time. We define the language $\mathcal{L}$ of those strings $y$ for which there is an $x$ with $f(x) = y$ such that the first bit of $x$ is 1. Prove that $\mathcal{L}$ is in NP $\cap$ co$-$NP. $\diamondsuit$

5.4 **Exercise** We say that a quantified Boolean formula is in class $F_k$ if all of its quantifiers are in front and the number of alternations between existential and universal quantifiers is at most $k$. Let $L_k$ be the set of true closed formulas in $F_k$. Prove that if P = NP then for all $k$ we have $L_k \in$ P. [Hint: induction on $k$.] $\diamondsuit$

5.5 **Exercise** Show that the following *general matching problem* is in both NP and co-NP: Given a graph $G$ and a natural number $k$, does there exist a $k$-edge matching in $G$? $\diamondsuit$

5.6 **Exercise** Give a polynomial algorithm to decide whether a disjunctive normal form is satisfiable. $\diamondsuit$

5.7 **Exercise** Show that the Satisfiablity Problem can be reduced to the special case when each variable occurs at least once unnegated and at least one negated. $\diamondsuit$

5.8 **Exercise** Given a graph $G$ and a variable $x_v$ for each vertex $v$ of $G$. Write up a conjunctive normal form that is true if and only if the values of the variables give a legal coloring of the graph $G$ with 2 colors. (I.e. the normal form is satisfiable if and only bif the graph is colorable with 2 colors.) $\diamondsuit$

5.9 **Exercise** Given a graph $G$ and three colors, 1,2 and 3. Let us introduce, to each vertex $v$ and color $i$ a logical value $x[v, i]$. Write up a conjunctive normal form $B$ for the variables $x[v, i]$ which is satisfiable if and only if $G$ can be colored with 3 colors.

[Hint: Let $B$ be such that it is true if and only if there is a coloring of the vertices with the given 3 colors for which $x[v, i]$ is true if and only if vertex $v$ has color $i$.] $\diamondsuit$

5.10 **Exercise** Define the language 3-SAT-3 and show that it is NP-complete. $\diamondsuit$

5.11 **Exercise** Define the language SAT-2 and show that it is in P. $\diamondsuit$

5.12 **Exercise** Prove that it is also NP-complete to decide whether in a given $2n$-vertex graph, there is an $n$-element independent set. $\diamondsuit$

5.13 **Exercise** In the GRAPH EMBEDDING PROBLEM, what is given is a pair $(G_1, G_2)$ of graphs. We ask whether $G_2$ has a subgraph isomorphic to $G_1$. Prove that this problem is NP-complete. $\diamondsuit$

5.14 **Exercise** 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. $\diamondsuit$

5.15 **Exercise** 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 with for this system is polynomial time solvable.

[Hint: reduce it to the general matching problem.] ♢

5.16 **Exercise** 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? ♢

5.17 **Exercise** An instance of the problem of 0-1 Integer Programming is defined as follows. The input of the problem is the 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 an immediate reduction of the 0-1 Integer Programming problem to the Subset Sum Problem. ♢

5.18 **Exercise** The SUM PARTITION PROBLEM is the following. Given a set $A = \{a_1, \ldots, a_n\}$ of integers, find a subset $B$ of $A$ such that $\sum_{i \in B} a_i = \sum_{i \notin B} a_i$. Prove that this problem is NP-complete. [Hint: use the NP-completeness of the subset sum problem.] ♢

5.19 **Exercise** Consider the following problem: given a finite set of integers in binary form, one has to partition this set into three subsets such that the sum of numbers within each subset is the same. Prove that this problem is NP-complete. You can use the fact that the problem of partitioning into two sets is known to be NP-complete. ♢

5.20 **Exercise** 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 and $n$ represents 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. ♢

5.21 **Exercise** Consider the following tiling problem $T$. Given is a finite set of tile types with a distinguished initial tile $I$ among them and a number $n$ in binary. It is to decide whether an $n \times n$ square can be tiled tiles of these types, when the lower left corner must be the initial tile. Prove that if $T$ is solvable in time polynomial in $n$ then NEXPTIME=EXPTIME. ♢

# 6 Randomized algorithms

## 6.1 Introduction

We cited Church's Thesis in Section: every "algorithm" (in the heuristic meaining 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., the 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 later section); 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.

## 6.2 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 an 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:

(6.1) **Theorem** *Schwartz's Lemma If $f$ is a not identically 0 polynomial in $n$ variables with degree at most $k$, and the values $\xi_i$ $(i = 1, \ldots, n)$ are chosen in the interval $[0, N-1]$ independently of each other according to the uniform distribution then*

$$\text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0\} \leq \frac{k}{N}.$$

(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 $k$ 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 $f_0, \ldots, f_t$ are polynomials of the variables $x_2, \ldots, x_n$, the term $f_t$ is not identically 0, and $t \leq k$. Now,

$$\begin{aligned}
&\text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0\} \\
&\quad \leq \text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0 \mid f_t(\xi_2, \ldots, \xi_n) = 0\}\text{Prob}\{f_t(\xi_2, \ldots, \xi_n) = 0\} \\
&\quad + \text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0 \mid f_t(\xi_2, \ldots, \xi_n) \neq 0\}\text{Prob}\{f_t(\xi_2, \ldots, \xi_n) \neq 0\} \\
&\quad \leq \text{Prob}\{f_t(\xi_2, \ldots, \xi_n) = 0\} + \text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0 \mid f_t(\xi_2, \ldots, \xi_n) \neq 0\}.
\end{aligned}$$

Here 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 $\xi_1$ is independent of the variables $\xi_2, \ldots, \xi_n$, 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 $\xi_1$ is its root is at most $t/N$). Hence

$$\text{Prob}\{f(\xi_1, \ldots, \xi_n) = 0\} \leq \frac{k-t}{N} + \frac{t}{N} \leq \frac{k}{N}.$$

■

This suggests the following a **randomized** algorithm to decide whether a polynomial $f$ is identically 0:

(6.2) **Algorithm** *We compute $f(\xi_1, \ldots, \xi_n)$ with integer values $\xi_i$ chosen randomly and independently of each other according to the uniform distribution in the interval $[0, 2kn]$. If we don't get the value 0 we stop: $f$ is not identically 0. If we get 0 value we repeat the computation. If we get 0 value 100 times we stop and declare that $f$ is identically 0.*

If $f$ is identically 0 then this algorithm will determine this. If $f$ is not identically 0 then in every separate iteration—according to Schwartz's 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 Chapters 3 and 5). 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 $x_{ij}$. Let us 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 with the matchings of the graph $G$ as Dénes Kőnig noticed while analyzing a work of Frobenius:

(6.3) **Theorem** *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 $\pi$ 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 contain 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 a later section.

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 $v_i v_j$ (where $i < j$) a variable $x_{ij}$ and construct an asymmetric $n \times n$ matrix $T = (t_{ij})$ as follows:

$$t_{ij} = \begin{cases} x_{ij} & \text{if } v_i v_j \in E(G) \text{ and } i < j, \\ -x_{ij} & \text{if } v_i v_j \in E(G) \text{ and } i > j, \\ 0 & \text{otherwise.} \end{cases}$$

The following analogue of the above cited Frobenius-Kőnig theorem comes from Tutte and we formulate it here without proof:

(6.4) **Theorem** *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 the bipartite graph, a randomized algorithm for deciding whether there is a perfect matching in $G$.

## 6.3 Prime 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.

(6.1) **Theorem** *"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—for a given $m$—the integer $a^{m-1}-1$ is divisible by $m$ 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:

(6.2) **Lemma** *An integer $m > 0$ is a prime if and only 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 5.

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 as random $a$ and check 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 miniscule compared with 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 pseudo-primes) 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 pseudo-prime is e.g. $561 = 3 \cdot 11 \cdot 17$.)

But at least of $m$ is not a pseudo-prime, then the random choice for $a$ works. This is guaranteed by the following lemma.

(6.3) **Lemma** *If $m$ is not a prime and not a pseudo-prime 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 $ab$ mod $m$ 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 $ab$ mod $m$) 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 pseudo-prime 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 if 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 finds 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 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 just $a^{m-1} - 1$. Hence every number satisfying the Miller 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 condition. If $m$ is composite then, however, it could happen that some $a$ satisfies the Fermat condition but not the Miller condition ($m$ can be a divisor of a product without being the divisor of any of its factors).

Thus the Miller 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.

(6.4) **Lemma** *Every pseudoprime $m$ is*

(a) *Odd*

(b) *Squarefree (is 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 $-(m-1)(m/p) + 1 \equiv m/p + 1 \not\equiv 1 \pmod{p^k}$ (all earlier terms are divisible by $p^k$) and if an integer is not divisible by $p^k$ then it is not divisible by $m$ either. ■

(6.5) **Lemma** *Let $m = p_1 p_2 \cdots p_t$ where the $p_i$'s are different primes. The relation $a^k \equiv 1 \pmod{m}$ holds for all $a$ relatively prime to $m$ if and only if $p_i - 1$ divides $k$ for all $i$ with $1 \leq i \leq t$.*

**Proof** If $p_i - 1$ divides $k$ for all $i$ with $1 \leq i \leq t$ then $a^k - 1$ is divisible by $p_i$ according to the little Fermat Theorem and then it is also divisible by $m$. Conversely, suppose that $a^k \equiv 1 \pmod{m}$ for all $a$ relatively prime to $m$. If e.g. $p_1 - 1$ would not divide $k$ then let $g$ be a primitive root modulo $p_1$ (the existence of primitive roots was stated in Chapter 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 $i \geq 2$. Then $(h, m) = 1$ and $p_1 \not| h^k - 1$, so $m \not| h^k - 1$. ■

(6.6) **Corollary** *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, $t \geq 2$, and $(p_i - 1)$ divides $(m - 1)$ for all $i$ with $1 \leq i \leq t$.*

(6.7) **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 test better than the Fermat test.

(6.8) **Theorem** *If $m$ is a composite number then at least half of the numbers $1, \ldots, m-1$ violate the Miller 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 $p_1 \cdots p_t$ $(t \geq 2)$ 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 $1 \leq i \leq t$.

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 $0 \leq l < k$. 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 is a Miller accomplice 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 $2^l M$, Lemma 6.5 implies that there is a number $a$ not divisible by $p_i$ for which $a^{2^l M} - 1$ is not divisible by $p_i$. The reasoning of Lemma 6.3 shows that then at most half of the mod $p_i$ residue classes will satisfy the Fermat comdition 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. ■

(6.9) **Lemma** *For a given $m$ and $a$, it is decidable in polynomial time whether $a$ satisfies the Miller 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:

(6.10) **Algorithm** We choose a number between 1 and $m-1$ randomly and check whether it satisfies the Miller condition. If it does not then $m$ is composite. If it does then we choose a new $a$. If the Miller 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 condition with probability 1/2. After hundred independent experiments the probability will therefore be at most $2^{-100}$ that the Miller condition is not violated even once, i.e., that the algorithm asserts that $m$ is a prime.

(6.11) **Remarks**

1. If $m$ is found composite by the algorithm then — interestingly enough — we see this not from its finding a divisor but from the fact that one of the residues violates the Miller 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 is significantly more difficult also in practice than the testing of primality. We will see in the section on cryptography that this empirical fact has important applications.

2. For a given $m$, we can try to find an integer $a$ violating the Miller 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.

$\diamondsuit$

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 cca. $1.44/n$. 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 $[1, 2^n]$.

## 6.4 Randomized complexity classes

In the previous subsections, we treated algorithms that used random numbers. Now we define a class of problems solvable by such algorithms.

First we define the corresponding machine. Let $T = (k, \Sigma, \Gamma, \Phi)$ be a non-deterministic Turing machine and let us be given a probability distribution for every $g \in \Gamma$, $h_1, \ldots, h_k \in \Sigma$ on the set

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

(It is useful to assume that the probabilities of events are rational numbers, since then events with such probabilities are easy to generate, provided that we can generate mutually independent bits.) A non-deterministic Turing machine together with these distributions is called a **randomized Turing machine**.

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 distunguish 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 $\Delta$RP. Obviously, BPP $\supset$ RP $\supset$ $\Delta$RP $\supset$ 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 namely 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 $\Delta$RP: the restriction of the longest running time would lead here already to a deterministic algorithm, and it is not known whether $\Delta$RP 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).

(6.1) **Remark** A Turing machine using randomness could also be defined in a different way: we could consider 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). It is easy to see that such a deterministic Turing machine fitted with a random tape and the non-deterministic Turing machine fitted with a probability distribution can replace each other in all definitions.

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. $\diamondsuit$

It can be seen that every language in RP is also in NP. It is trivial that the classes BPP and $\Delta$RP 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.

(6.2) **Theorem** *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*

(a) $\mathcal{L} = \{ x \in \Sigma^* : y \in \Sigma^{f(|x|)}\ x\&y \in \mathcal{L}' \}$ *and*

(b) *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. ■

The connection of the classes RP and $\Delta RP$ is closer than it could be expected on the basis of the analogy to the classes NP and P:

(6.3) **Theorem** *The following properties are equivalent for a language* $\mathcal{L}$*:*

**(i)** $\mathcal{L} \in \Delta\mathrm{RP}$*;*

**(ii)** $\mathcal{L} \in RP \cap \mathrm{co{-}RP}$*;*

**(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 $\tau$ and the probability of giving it up is $p$ then on this same input, the expected number of steps of the modified machine is

$$\sum_{t=1}^{\infty} p^{t-1}(1-p)t\tau = \frac{\tau}{1-p} \leq 2\tau.$$

■

We have seen in the previous subsection that the "language" of composite numbers is in RP. Even more is true: Adleman and Huang have shown that this language is also in $\Delta$RP. For our other important example, the not identically 0 polynomials, it is only known that they are in RP. Among the algebraic (mainly group-theoretical) problems, there are many that are in RP or $\Delta$RP but no polynomial algorithm is known for their solution.

(6.4) **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. ♢

6.1 **Exercise** 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 Tshebysheff's Inequality.] ♢

6.2 **Exercise** 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 $2^{-t}$. ♢

6.3 **Exercise** Suppose that somebody gives you three $n \times n$ matrices $A, B, C$ (of integers of maximimum length $l$) and claims $C = AB$. You are too busy to verify this claim exactly and 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.

(a) How large must $N$ be chosen to make the probability of false acceptance smaller than 0.01?

(b) Compare the time complexity the probabilistic algorithm to the one of the deterministic algorithm computing $AB$.

♢

6.4 **Exercise** 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. ♢

6.5 **Exercise** Show that the Turing machine equipped with a random tape and the non-deterministic Turing machine equipped with a probability distribution are equivalent: if some language is accepted in polynomial time by the one then it is also accepted by the other one. ♢

6.6 **Exercise** 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. ♢

6.7 **Exercise** 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. ♢

# 7 Information complexity: the complexity-theoretic notion of randomness

## 7.1 Introduction

The mathematical foundation of probability theory appears among the famous problems of Hilbert formulated in 1900 (mentioned before). 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 other 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 probabity 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. Then we discuss the notion of an *informatically random sequence*, and show that such sequences behave like "usual" random sequences: they obeu the Laws of Large Numbers. Finally, we discuss the problem of *optimal encoding* of various structures.

## 7.2 Information complexity

Fix an alphabet $\Sigma$. Let $\Sigma_0 = \Sigma \setminus \{*\}$ and consider a two-tape universal Turing machine over $\Sigma$. It will be convenient to identify $\Sigma_0$ with the set $\{0, 1, \ldots, m-1\}$. Consider a 2-tape, universal Turing machine $T$ over $\Sigma$. We say that the word (program) $q$ over $\Sigma$ **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 the machine is started so that its program tape contains a word of the form $x$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.

(7.1) **Lemma** *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 $p_0$DATA$x$ will print the word $x$ and stop. Thus the constant $c_T = |p_0| + 4$ satisfies the conditions. ■

In what follows we assume, to be specific, that $c_T \leq 100$.

(7.2) **Remark** We had to be a little careful since we did not want to restrict what symbols can occur in the word $x$. In BASIC, 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.

(7.3) **Invariance Theorem** *Let $T$ and $S$ be universal Turing machines satisfying the 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 $\Sigma_0^*$ 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 $T$.

Unfortunately, the following theorem shows that the optimal code cannot be found algorithmically.

(7.4) **Theorem** *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 $\Sigma_0^*$ 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 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 explicitely 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 even taken together, the 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 with 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 $\mathbf{K}(x)$.

Thus the halting problem is not decidable.

In contrast to Theorem 7.4, 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 size order, or $p(x) = (m+1)^{-|x|-1}$ where $m$ is the alphabet size.

(7.5) **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. $\diamondsuit$

(7.6) **Theorem** *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 $\Sigma_0^*$ 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).

(7.7) **Lemma** (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 words $x$, it is easy to find the index $j$ for which $x = y_j$. Next, find the first $k \geq j$ for which

$$p(y_1) + \cdots + p(y_k) > 1 - p(y_j). \tag{7.8}$$

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 lemma, 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 it 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 $N$. To the end, write

$$\sum_{i=1}^{N} p(x_i)(|f(x_i)| - \mathbf{K}(x_i)) \tag{7.9}$$

$$= \sum_{i=1}^{N} p(x_i)(|f(x_i)| - \log_m i) + \sum_{i=1}^{N} p(x_i)(\log_m i - \mathbf{K}(x_i)). \tag{7.10}$$

We claim that both sums remain 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 (7.9) is at most $C$.

To estimate the second term in (7.9), we use the following simple but useful principle. Let $a_1 \geq a_2 \geq \ldots \geq a_m$ be a decreasing sequence and let $b_1, \ldots, b_m$ be an arbitrary sequence of real numbers. Let $b_1^* \geq \ldots \geq b_m^*$ be the sequence $b$ ordered decreasingly, and let $b_1^{**} \leq \ldots \leq b_m^{**}$ 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 $1 + m + \ldots m^k < m^{k+1}$. 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. ∎

## 7.3 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).

(7.1) **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 estimate first the complexity of "average" 0-1 sequences.

(7.2) **Lemma** *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. ∎

(7.3) **Corollary** *The complexity of 99% of the $n$-digit 0-1 sequences is greater than $n - 7$. 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 introduction to the section on randomized computation. 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 the exercise mentioned after Theorem 7.4, 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 7.4, 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:

(7.4) **Lemma** *If the number of 1's in a $0-1$-sequence $x$ of length $n$ is $k$ then*

$$\mathbf{K}(x) \le \log_2 \binom{n}{k} + \log_2 n + \log_2 k + O(1).$$

*Let $k = pn$ $(0 < p < 1)$, then this can be estimated as*

$$\mathbf{K}(x) \le (-p \log p - (1-p)\log(1-p))n + O(\log n).$$

In particular, if $k > (1/2 + \varepsilon)n$ or $k < (1/2 - \varepsilon)n$ then

$$\mathbf{K}(x) \le cn + O(\log n)$$

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 $\varepsilon$.

**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 $\log_2 \binom{n}{k} + 2\log_2 n + 2\log_2 k$ 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 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:

(7.5) **Theorem** *If $x$ is informatically random then $a_n/n \to 1/2$ $(n \to \infty)$.*

The question arises whether the definition of an algorithmically random sequence is not too strict, whether there are any algorithmically random infinite sequences at all. Let us show that not only there are such sequences but that almost all sequences have this property:

(7.6) **Theorem** *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 algorithmically random with probability 1.*

**Proof** For a fixed $\epsilon > 0$, let $S$ be the set of all those finite sequences $y$ for which $\mathbf{K}(y) < (1-\epsilon)|y|$, and let $A_n$ denote the event $x_n \in S$. Then by lemma 7.2,

$$\mathrm{Prob}(A_n) \leq \sum_{y \in S} 2^{-n} < 2^{(1-\epsilon)n} + 12^{-n} = 2^{1-\epsilon n},$$

and hence the sum $\sum_{k=1}^{\infty} \mathrm{Prob}(A_n)$ is convergent. But then, the Borel-Cantelli Lemma in probability theory implies that with probability 1, only finitely many of the events $A_k$ occur. But this just means that $\mathbf{K}(x_n)/n \to \infty$. ∎

## 7.4 Kolmogorov complexity and data compression

Let $\mathcal{L} \subset \Sigma_0^*$ 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,

$$\eta(n) \geq \log_2 |\mathcal{L}_n|.$$

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 $\log_2 t \leq \log_2 |\mathcal{L}_n|$ bits, plus a constant number of additional bits (the program for taking the elements of $\Sigma^*$ 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.

(7.1) **Example** In the proof of Lemma 7.4, 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 $\{a_1, \ldots, a_m\}$ lexicografically 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 $\{b_1, \ldots, b_n\}$ with this property is exactly $\binom{a_i}{m-i+1}$. Summing this for all $i$ we find that

$$t = 1 + \binom{a_1}{m} + \binom{a_2}{m-1} + \cdots + \binom{a_m}{1}. \tag{7.2}$$

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 $a_2 > a_3 > \ldots > a_m \geq 0$ and that there is no "remainder" after $m$ steps, i.e., that 7.2 holds. It can therefore be determined in polynomial time which subset is lexicographically the $t$-th. $\diamondsuit$

(7.3) **Example** 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:

(7.4) **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.

**(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. $2(n-1)\log_2 n$ 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 $n^{n-2}$ 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 a refinement of the procedure (c). The idea is that we order the edges $[i, \alpha(i)]$ not by the magnitude 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.

(7.5) **Claim** *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 $\alpha(i_1), \ldots, \alpha(i_{n-1}), 0$. 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 $\alpha(i_k), \ldots, \alpha(i_{n-1})$. So, $i_k$ is also uniquely determined.

(7.6) **Claim** *Every sequence $(b_1, \ldots, b_{n-2})$, where $1 \le b_i \le n$, 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 $1, \ldots, n$. 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. $\diamondsuit$

(7.7) **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. $\diamondsuit$

(7.8) **Example** 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 $n^{n-2}/n! \leq 2^{n-2}$. (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.) The information-theoretical lower bound is therefore at least $n-2$.

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 has not 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 $x_0$. $\diamondsuit$

(7.9) **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 $(a) - (b)$.

$\diamondsuit$

# 8 Pseudo-random numbers

## 8.1 Introduction

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 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 is called cryptography, to be treated in a later chapter.

The way a *pseudo-random 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.

## 8.2 Introduction

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 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 is called cryptography, to be treated in a later chapter.

The way a *pseudo-random 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.

## 8.3 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 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.

(8.1) **Example** 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 perodicity. Indeed, let

$$f(x_0, dots, x_{n-1}) = b_0x_0 + b_1x_1 + \ldots b_{n-1}x_{n-1}$$

(where $b_i \in \{0, 1\}$). Assume that we we don't 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 $b_i$:

$$\begin{aligned} b_0a_0 + b_1a_1 + \ldots b_{n-1}a_{n-1} &= a_n \\ b_0a_1 + b_1a_2 + \ldots b_{n-1}a_n &= a_{n+1} \\ &\vdots \\ b_0a_{n-1} + b_1a_n + \ldots b_na_{2n-2} &= a_{2n-1} \end{aligned}$$

Here are $n$ equations to determine these $n$ 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. $\diamondsuit$

(8.2) **Example** 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 are good for most practical applications. $\diamondsuit$

(8.3) **Example** 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. $\diamondsuit$

## 8.4 The notion of a psuedorandom number generator

In general, a pseudo-random 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 pseudo-random bit generator if no applications running only in polynomial time can distinguish $g(s)$ from truly random strings.

To make the defition 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 $|G(x)|$ depends only on $|x|$ and $|x| \leq |G(x)| < |x|^c$ for some constant $c$. 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 0 as an output as "not random", and 1 as an output 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 $1/2 + \mathrm{NEGL}(n)$.

Condition (b) 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 truely random is only negligibly larger than $1/2$ (which can of course be achieved by guessing randomly). The probability in (b) is 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 exercises). But we will see in the next section that they exist under some complexity-theoretic assumptions.

Condition (b) is very general and it 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 *unpredictible* if the following holds. Let $n \geq 1$ and let $x$ be chosen uniformly from $\{0,1\}^n$. Let $g(x) = G_1 G_2 \dots G_N$. For every randomized polynomial time algorithm $\mathcal{A}$ that accepts a string $x \in \{0,1\}^i$ $(i \leq N)$ as input and computes a bit from it,

$$\max_i \mathsf{P}\Big(\mathcal{A}(G_1 \dots G_i) = G_{i+1}\Big) = \frac{1}{2} + \mathrm{NEGL}(n). \tag{8.1}$$

Informally: we try to use $\mathcal{A}$ to predict each bit of $g_1 \dots g_N$ from the previous bits. Then at any stage, the probability that we suceed is only negligibly larger than 1/2 (which we could achive just by guessing randomly).

(8.2) **Theorem** *[A. Yao] A generator $g$ is a safe random number generator if and only if it is unpredictible.*

**Proof** I. Suppose that $g$ is no unpredictable. Then there is a randomized polynomial time algorithm $\mathcal{A}$, a constant $k > 0$, there are infinitely many values of $n$, and for each such value a certain $i < N$, such that

$$\mathsf{P}\Big(\mathcal{A}(G_1 \dots G_i) = G_{i+1}\Big) > \frac{1}{2} + \frac{1}{n^k}.$$

(where $x \in \{0,1\}^n$ is a uniformly chosen random string and $g(x) = G_1 \dots G_N$).

But then we can perform the following randomness test $\mathcal{B}$ on strings $y = y_1 \dots y_N$: if $\mathcal{A}(y_1 \dots y_i) = y_{i+1}$ we declare $y$ "non-random", else we declare it "random". Then if we give $\mathcal{B}$ either a truly random string $R_1 \dots R_N$ or $g(x)$ (each with probability 1/2), the probability of success is

$$\begin{aligned} &\frac{1}{2}\mathsf{P}\Big(\mathcal{A}(R_1 \dots R_i) \neq R_{i+1}\Big) + \frac{1}{2}\mathsf{P}\Big(\mathcal{A}(G_1 \dots G_i) = G_{i+1}\Big) \\ &\geq \frac{1}{2}\frac{1}{2} + \frac{1}{2}\left(\frac{1}{2} + \frac{1}{n^k}\right) = \frac{1}{2} + \frac{1}{2n^k}. \end{aligned}$$

Since this is non-negligibly larger than 1/2, the generator is not safe.

II. Assume that there exists an 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 some bit of $G_1 \dots G_N$.

The success probability of $\mathcal{A}$ is

$$\frac{1}{2}\mathsf{P}(\mathcal{A}(r) = 1) + \frac{1}{2}\mathsf{P}(\mathcal{A}(g(x)) = 0)$$

(since success means that $\mathcal{A}$ accepts $r$, if $r$ is given to it, but rejects $g(x)$, if $g(x)$ is given to it). In other words, this probability is

$$\frac{1}{2} + \frac{1}{2}\Big(\mathsf{P}(\mathcal{A}(r) = 1) - \mathsf{P}(\mathcal{A}(g(x)) = 1)\Big).$$

So it follows that

$$\mathsf{P}(\mathcal{A}(r) = 1) - \mathsf{P}(\mathcal{A}(g(x)) = 1) > \frac{2}{n^c}. \tag{8.3}$$

The trick is to consider the mixed sequences

$$y^i = G_1 \dots G_i R_{i+1} \dots R_N$$

and subject them to $\mathcal{A}$. We have $y^0 = r$ and $y^N = g(x)$, and hence

$$\mathsf{P}(\mathcal{A}(y^0) = 1) - \mathsf{P}(\mathcal{A}(y^N) = 1) > \frac{2}{n^k}.$$

Hence it follows that there is an index $i$, $1 \leq i \leq N$, such that

$$\mathsf{P}(\mathcal{A}(y^{i-1}) = 1) - \mathsf{P}(\mathcal{A}(y^i) = 1) > \frac{2}{Nn^k}. \tag{8.4}$$

We are going to show that this $i$ is predictible with success probability non-negligibly larger than $1/2$. The prediction algorithm is simple: we guess that

$$X = R_{i+1} \oplus \mathcal{A}(y^i)$$

is the value of $G_{i+1}$ (this value is indeed computed by a randomized polynomial time algorithm from $G_1, \dots, G_i$).

For simplicity of notation, write $G = G_i$, $R = R_i$, and

$$A_0 = \mathcal{A}(G_1 \dots G_{i-1} 0 R_{i+1} \dots R_N), A_1 = \mathcal{A}(G_1 \dots G_{i-1} 1 R_{i+1} \dots R_N).$$

Note that $G$, $X$, $R$, $A_0$ and $A_1$ are random bits. These are not independent in general, but $R$ is independent of the rest. Moreover, it is easy to see that we can write

$$X = R(1 - A_1) + (1 - R)A_0$$

Hence $G = X$ (in other words, our prediction is successful) is equivalent to saying that

$$G(R(1 - A_1) + (1 - R)A_0) + (1 - G)(1 - R(1 - A_1) - (1 - R)A_0) = 1,$$

or

$$\begin{aligned} GR(1 - A_1) \quad &+ \quad G(1 - R)A_0 \; + \; (1 - G) \\ &- \quad (1 - G)R(1 - A_1) - (1 - G)(1 - R)A_0 = 1. \end{aligned}$$

Since the left hand side is always 0 or 1, the probability that this happens is the expectation of the left hand side. But this is

$$\begin{aligned} \mathsf{E}\Big(GR(1 - A_1) \quad &+ \quad G(1 - R)A_0 + (1 - G) - (1 - G)R(1 - A_1) \\ &- \quad (1 - G)(1 - R)A_0\Big) \end{aligned}$$

$$\begin{aligned} = \mathsf{E}(GR(1 - A_1)) \quad &+ \quad \mathsf{E}(G(1 - R)A_0) \\ &+ \quad \mathsf{E}(1 - G) - \mathsf{E}((1 - G)R(1 - A_1)) \\ &- \quad \mathsf{E}((1 - G)(1 - R)A_0). \end{aligned}$$

Since $R$ is independent of the other variables, we can pull out $\mathsf{E}(R) = 1/2$, and get

$$\frac{1}{2}\Big(\mathsf{E}(G(1-A_1)) + \mathsf{E}(GA_0) + 2\mathsf{E}(1-G) - \mathsf{E}((1-G)(1-A_1)) - \mathsf{E}((1-G)A_0)\Big)$$

$$= \frac{1}{2}\mathsf{E}(1 + 2GA_0 - 2GA_1 - A_0 + A_1).$$

We can also express (8.4) in terms of these bits. In fact, $\mathcal{A}(y^i) = RA_1 + (1-R)A_0$, and hence

$$\mathsf{P}(\mathcal{A}(y^i) = 1) = \frac{1}{2}\mathsf{E}(A_0 + A_1).$$

Similarly, $\mathcal{A}(y^{i+1}) = GA_1 + (1-G)A_0$, and hence

$$\mathsf{P}(\mathcal{A}(y^{i+1}) = 1) = \mathsf{E}(GA_1 + (1-G)A_0).$$

Thus (8.4) implies that

$$\frac{1}{2}\mathsf{E}(A_0 + A_1 - 2GA_1 - 2(1-G)A_0) = \frac{1}{2}\mathsf{E}(A_1 - A_0 - 2GA_1 + 2GA_0) > \frac{2}{Nn^k}.$$

But then

$$\mathsf{P}(G = X) = \frac{1}{2}\mathsf{E}(1 + 2GA_0 - 2GA_1 - A_0 + A_1) > \frac{1}{2} + \frac{1}{Nn^k} > \frac{1}{2} + \frac{1}{n^{k+c}},$$

which shows that our prediction is non-negligibly better than the trivial. ∎

The existence of pseudo-random bit generators can be derived from some (unproved) complexity theoretic assumptions that are nevertheless rather plausible. These are discussed in the next section.

## 8.5 One-way functions

A one-way function is a function that is easy to compute but difficult to invert. The exact definition is as follows.

(8.1) **Definition** A function $f : \{0,1\}^* \to \{0,1\}^*$ is called a *one-way function* if

(a) there is a constant $c \geq 1$ such that $|x|^{1/c} < |f(x)| < |x|^c$;

(b) $f(x)$ is polynomial time computable;

(c) 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 $\{0,1\}^n$,

$$\mathsf{P}(f(\mathcal{A}(f(y))) = f(y)) = NEG(n). \tag{8.2}$$

♢

The third condition needs explanation. We can read it as follows: 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

$$\mathsf{P}(f(\mathcal{A}(z)) = z) = NEG(n) \tag{8.3}$$

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 (8.2) 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, (8.3) is equivalent to (8.2).

(8.4) **Theorem** *Let $g$ be a safe random number generator, and assume that $|g(x)| \geq 2|x|$. 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 $\{0,1\}^n$,

$$\mathsf{P}(g(\mathcal{A}(g(y))) = g(y)) > \frac{1}{n^k}.$$

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 $r = R_1 \ldots R_N$ or $g(x)$ (each with probability $1/2$), the probability of success is

$$\frac{1}{2}\mathsf{P}(g(\mathcal{A}(r)) \neq r) + \frac{1}{2}\mathsf{P}(g(\mathcal{A}(g(x))) = g(x))$$

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 os $2^{n-N} < 2^{-n}$ (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. For this, we need the following lemma due to Goldreich and Levin. For two 0-1 strings $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_n$, define $b(u,v) = u_1 v_1 \oplus \ldots \oplus u_n v_n$.

(8.5) **Lemma** *Let $f$ be a one-way permutation. Then for every randomized polynomial time algorithm $\mathcal{A}$, and for $x, y$ chosen randomly and uniformly from $\{0,1\}^n$,*

$$\mathsf{P}\Big(\mathcal{A}(x,y) = b(f^{-1}x, y)\Big) = \frac{1}{2} + \mathrm{NEGL}(n).$$

Informally, selecting a random subset of the bits of $f^{-1}$, their sum modulo 2 cannot be computed with success probability non-negligibly larger than 1/2. the proof of this lemma is quite involved, and we have to state it without proof.

(8.6) **Theorem** *Suppose that a one-way permutation exists. Then there exists a safe random number generator.*

**Proof** We describe a random number generator. Let $f$ be a one-way permutation and $k > 0$ a constant. We stretch a random sequence $x = (x_1, \ldots, x_{2n})$of length $2n$ (the parity of the length is even for convenience) to a pseudorandom sequence of length $N = n^k$ as follows. Compute the strings

$$y_1^t \ldots y_n^t = f^t(x_1, \ldots, x_n)$$

for $t = 1, \ldots, N$ (here $f^t$ is the function $f$ iterated $t$ times), and let

$$g_t = y_1^t x_{n+1} \oplus \ldots \oplus y_n^t x_{2n}.$$

We show that $g(x) = g_N g_{N-1} \ldots g_1$ defines a safe random number generator (note the trick of writing the bits in the reverse order!). Using theorem 8.2, it suffices to show that for every $1 \leq i \leq N$, and for every randomized polynomial time algorithm $\mathcal{A}$ computing a bit from $g_N \ldots g_{i+1}$, 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 $f^t(x_1, \ldots, x_n)$ for $t \geq i+1$ and also the string $x_{n+1} \ldots x_{2n}$ (from which $g_N \ldots g_{i+1}$ is easily computed). But then, we don't have to give $f^{i+2}(x_1, \ldots, x_n) \ldots, f^N(x_1, \ldots, x_n)$; these are easily computed from $f^{i+1}(x_1, \ldots, x_n)$. Thus we may assume that the algorithm gets $z = f^{i+1}(x_1, \ldots, x_n)$ and $u = x_{n+1} \ldots x_{2n}$, and has to guess $g_i = b(f^{-1}(z), u)$. But by lemma 8.5, the probability of succeeding is only negligibly larger than 1/2, as claimed. ∎

### 8.5.1 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 $n$.

**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 the still, a large fraction of the instances remains difficult.

**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*, of *discrete logarithm*, of $y$ with respect to $p$.

**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 $x^2 \equiv x \pmod m$. This is solvable in polynomial time by a probabilistic algorithm if $m$ is a prime but is considered difficult in the general case.

## 8.6 Discrete square roots

In this section we discuss the 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 integer $x$, 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.

(8.1) **Lemma** *A residue $x$ has a square root if and only if*

$$x^{(p-1)/2} \equiv 1 \pmod{p}. \tag{8.2}$$

**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 this square root? For some primes, this is easy:

(8.3) **Lemma** *Assume that $p \equiv 3 \pmod{4}$. Then for every quadratic residue $x$, $x^{(p+1)/4}$ is a square root of $x$.*

Indeed,

$$\left(x^{(p+1)/4}\right)^2 = x^{(p+1)/2} = x \cdot x^{(p-1)/2} \equiv x \pmod{p}.$$

The case when $p \equiv 1 \pmod{4}$ is more difficult, and the solution uses randomization. In fact, randomization is only needed in the following auxiliary algorithm:

(8.4) **Lemma** *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 8.1 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.

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

$$x^{2^j q} z^{2^{j+1} t_j} \equiv 1 \pmod p. \tag{8.5}$$

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. Show that squaring an integer is not a random number generator.

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 $rev(g(s))$.

If P = NP then no one-way function exists.

Somebody proposes the following random number generator: it takes an integer $x$ with $n$ bits as the seed, and outputs $\lfloor x^3 / 10^n \rfloor$. Show that this random number generator is not safe.

# 9 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 modelling 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 modelled 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 section, 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 parallellizability 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.

## 9.1 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 $u[i]$. Beyond this, every processor has a separate memory cell $v$ containing the serial number of the processor. The processor can read its own name $v$ and can read and write its own cells $x, y, u[i]$ 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

$$u[i] := 0; \qquad u[i] := u[i] + 1; \qquad u[i] := u[i] - 1; \qquad u[i] := u[j];$$
$$u[i] := u[i] + u[j]; \qquad u[i] := u[i] - u[j]; \qquad u[i] := u[u[j]]; \qquad u[u[i]] := u[j];$$
$$u[i] := x[u[j]]; \qquad x[u[i]] := u[j]; \qquad \textbf{if } u[i] \leq 0 \textbf{ then goto } p;$$

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 $x[t]$ is the total number of digits in $k$ and $t$, i.e., approximately $\log_2 |k| + \log_2 |t|$. 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? We referred to

this problem already in connection with the definition of the Parallel Pointer Machine. 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 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 smalles serial number succeeds: this model is called **priority concurrent-read, concurrent-write** (P-CRCW), or shortly, the **priority** model.

9.1 **Exercise**

(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 completely conflict-free model, this actually requires $\Omega(\log n)$ steps.

*(c) How many steps are needed on the other two models?

♢

9.2 **Exercise** 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. ♢

9.3 **Exercise**

(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) Perform the same on the completely conflict-free model.

♢

It is obvious that the above models are stronger and stronger since they permit more and more. It can be shown, however, that—at least if the number of processors is not too great—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. The following lemma is concerned with such a statement.

(9.1) **Lemma** *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 $O(t(\log p)^2)$.*

(9.2) **Remark** 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. $\diamondsuit$

**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 $2pz, 2pz + 1, \ldots, 2pz + p - 1$ whatever 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 $i$ leaves 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 its subordinates started to the right from them and reaches 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 $O((\log z)^2)$-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.

It is easy to convince ourselves that the following statement holds.

(9.3) **Proposition** *If a computation can be performed with $p$ processors in $t$ steps with numbers of at most $s$ bits then for all $q < p$, it can be performed with $q$ processors in $O(tp/q)$ steps with numbers of at most $O(s+\log(p/q))$ bits. In particular, it can be performed on a sequencial Random Access Machine in $O(tp)$ steps with numbers of length $O(s+\log p)$.*

The fundamental question of the complexity theory of parallel algorithms is just the opposite of this: given is a sequential algorithm with time $N$ and we would like to implement it on $p$ processors in "essentially" $N/p$ (say, in $O(N/p)$) steps. Next, we will overview some complexity classes motivated by this question.

Randomization is, as we will see, 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).

## 9.2 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 computes conflict-free with $O(|x|^{c_1})$

processors in time $O((\log|x|)^{c_2})$. (According to Lemma 9.1, it would not change this definition if we used e.g. the priority model instead.)

The class NC of languages consists of those languages $\mathcal{L} \subset \{0,1\}^*$ whose characteristic function can be computed by an NC-program.

(9.1) **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). $\diamondsuit$

Obviously, $NC \subset P$. It is not known whether equality holds here but the answer is probably no.

We define the **randomized** NC, or RNC, class of languages on the pattern of the class BPP. This 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 an 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 $1/4$.

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.

(9.2) **Proposition** *The adjacency-matrices of graphs containing a triangle form a language in* NC.

**Proof** 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.

(9.3) **Proposition** *The adjacency matrices of connected graphs form a language in NC.*

**Proof** 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.

9.4 **Exercise** Give an NC algorithm that in a given graph, computes the distance of two points. ♢

(9.4) **Proposition** *The product of two matrices (in particular, the scalar product of two vectors), and the $k$-th power of an $n \times n$ matrix ($k \leq n$) is* NC*-computable.*

**Proof** We can compute the scalar product of two vectors as follows: we multiply—parallelly—their corresponding elements; then we group the products obtained this way in pairs and form the sums; then we group these sums in pairs and form the sums, etc. Now, we can also compute the product of two matrices since each element of the product is the scalar product of two vectors, and these can be computed parallelly. Now the $k$-th power of an $n \times n$ matrix can be computed on the pattern of $a^b \pmod c$ (Lemma 3.3). ■

The next algorithm is maybe the most important tool of the theory of parallel computations.

(9.5) **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.*

**Proof** 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 $n \times n$ unit 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:

$$F(t) = \prod_{k=1}^{n} [\sum_{j=0}^{n} (- \sum_{m=1}^{n} t^m (A_k^m)_{kk})^j].$$

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 $\det B$. ■

Using Theorem 6.3 with random substitutions, we arrive at the following important application:

(9.6) **Corollary** *The adjacency matrices of the graphs with complete matchings form a language in* RNC.

No combinatorial proof (i.e. one avoiding the use of Csánky's theorem) is known for this fact. 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).

9.5 **Exercise** 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. $\diamondsuit$

# 10 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).

## 10.1 Algorithms using decision trees

Consider some simple examples.

### 10.1.1 Binary search

Perhaps the simplest situation in which decision trees are used is *binary search*. We want to compute an integer $k$, about which at the beginning we only 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 $\lceil \log_2 N \rceil$ times, we can determine $a$. We have used this method when we showed that factoring of 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 $[w + 1, v]$.

### 10.1.2 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 1g; 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 into 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 numbers of elements $\lceil |X_v|/2 \rceil$ and $\lfloor |X_v|/2 \rfloor$. These correspond to the children of $v$. Measuring the first one we learn which one contains the false coin.

### 10.1.3 Finding a false coin with a two-armed scale

Again, we are given $n$ outwardly identical 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 fals 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 to be found 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 contains the false coin.

### 10.1.4 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, wich which one in the pair is greater, and this determines the order. These comparisons are not, however, independent: often, we can infer the order of certain pairs using transitivity. Indeed, it is enough to make $\sum_{k=1}^{n} \lceil \log_2 k \rceil \sim n \log_2 n$ 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 $\lceil \log_2 n \rceil$ 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.

(10.1) **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. $\diamondsuit$

### 10.1.5 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 10.1).

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 $p_n$.

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.

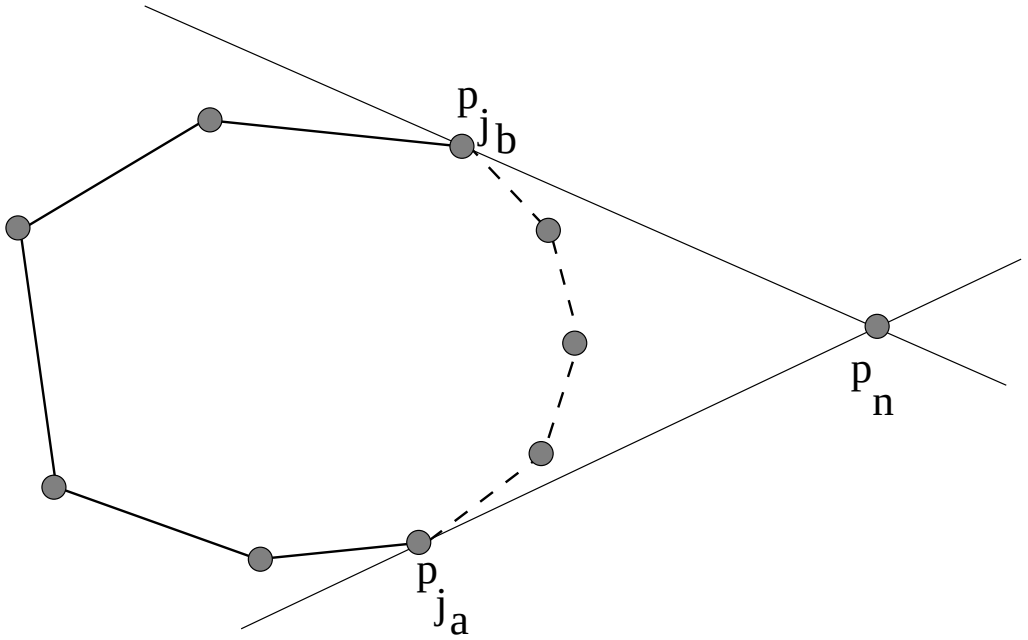



Figure 10.1: A simple convex hull algorithm

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.)

## 10.2 The notion of decision trees

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 $\Phi$ of functions defined on $A$ with values in $\{1, \ldots, d\}$, the **test-functions**. A **decision tree** is a rooted tree whose internal nodes (including the root) have $d$ children (the tree is $d$-regular), its leaves are labelled with the elements of $B$, the other nodes with the functions of $\Phi$. We assume that for every vertex, the edges leaving it are numbered in some order.

Every decision tree determines a function $f : A \to B$. Let namely $a \in A$. 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 the reading in of the value of one of the variables. In this case, we call the decision tree **simple**. Every simple decision tree is binary

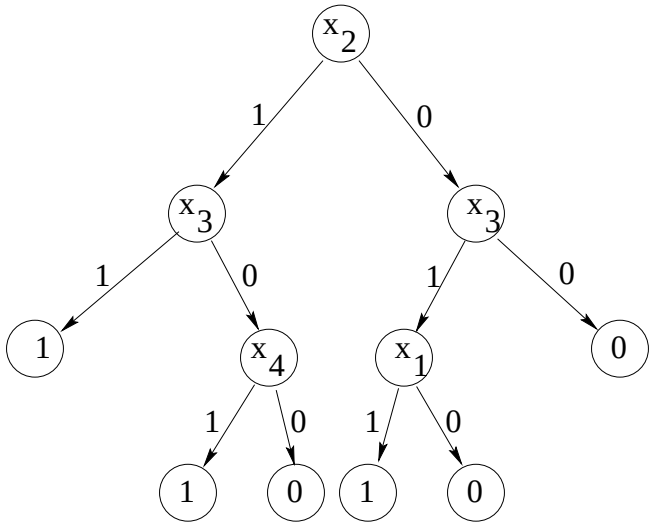



Figure 10.2: A simple decision tree

(2-regular), 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 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 $f$.

(10.1) **Example** 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 10.2. This shows that $D(f) \leq 3$. It is easy to see that we cannot compute this funtion by a decision tree of depth 2, and hence $D(f) = 3$. $\diamondsuit$

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 $\Phi$. 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.)

## 10.3 Nondeterministic decision trees

The idea 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 classe 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.

(10.1) **Example** 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 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 10.8). $\diamondsuit$

(10.2) **Example** Let now $G$ be an arbitrary but fixed $n$-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 also be simply expressed by a Boolean formula:

$$f(x_1, \ldots, x_n) = \bigvee_{ij \in E(G)} (x_i \wedge x_j).$$

If the value of this Boolean function is 1 then this will 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 $\alpha$ is the maximum number of independent points in the graph. It can also be proved (see Theorem 10.6) 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

$$D(f) = n.$$

$\diamondsuit$

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:

(10.3) **Theorem**
$D(f) \leq D_0(f)D_1(f)$.

**Proof** We use induction over the number $n$ of variables. If $n = 1$ then the inequality is trivial.

Let (say) $f(0, \ldots, 0) = 0$; 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 strenghened:

$$D_1(g) \leq D_1(f) - 1.$$

Consider an input $(a_{k+1}, \ldots, a_n)$ of $g$ with $g(a_{k+1}, \ldots, a_n) = 1$. 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 at the value $a_i$, the value of $f$ becomes 1 independently of the other variables. One

of the first $k$ variables must occur among these $m$ variables; otherwise, $f(0,\dots,0,a_{k+1},\dots,a_n)$ 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 $x_{i_1},\dots,x_{i_m}$), which is a contradiction. Thus, in the function $g$, at the position $a_{k+1},\dots,a_k$, only $m-1$ variables must be fixed to obtain the identically 1 function. From this, the claim follows. From the induction hypothesis,

$$D(g) \le D_0(g)D_1(g) \le D_0(f)(D_1(f)-1)$$

and hence

$$D(f) \le k + D(g) \le D_0(f) + D(g) \le D_0(f)D_1(f).$$

■

In Example 10.2, we could define the function by a disjunctive 2-normal form and $D_1(f) = 2$ was true. This is not an accidental coincidence:

(10.4) **Proposition** *If $f$ is expressible by a disjunctive $k$-normal form then $D_1(f) \le k$. If $f$ is expressible by a conjunctive $k$-normal form then $D_0(f) \le k$.*

**Proof** It is enough to prove the first assertion. Let $(a_1,\dots,a_n)$ 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. ■

For monotone functions, the connection expressed in the previous proposition is even tighter:

(10.5) **Proposition** *A monotonic Boolean function is expressible by a disjunctive [conjunctive] $k$-normal form if and only if $D_1(f) \le k$ [$D_0(f) \le k$].*

**Proof** According to Proposition 10.4, it is sufficient to see that if $D_1(f) = k$ then $f$ is expressible by a disjunctive $k$-normalform. Let $\{x_{i_1},\dots,x_{i_m}\}$ 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 function is called a **minterm**.) Notice that then we had to fix every variable $x_{i_j}$ necessarily to 1: due to the monotonicity, this fixing gives the identically 1 function, and if a variable could also be fixed to 0 then it would not have to be fixed to begin with.

We will show that $m \le k$. Let us namely assign the value 1 to the variables $x_{i_1},\dots,x_{i_m}$ 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 obtained function identically 1. By the above remarks, we can assume that we only fix 1's, i.e. we only fix some of the variables $x_{i_1},\dots,x_{i_m}$. But then due to the minimality of the set $\{x_{i_1},\dots,x_{i_m}\}$, we had to fix all of them, and hence $m \le k$.

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. By what was said above, we obtain a disjunctive $k$-normal form this way. It can be verified trivially that this defines the function $f$. ■

## 10.4 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**.

(10.1) **Lemma** *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 same, 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.

(10.2) **Lemma** *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. We have seen (championship by elimination) that $n-1$ comparisons are enough for this. Lemma 10.1 gives only $\log n$ for lower bound; but we can apply Lemma 10.2 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 $a' = (a'_1, \ldots a'_n)$ be the permutation containing the numbers $1, \ldots, p$ in the positions corresponding to the vertices of $G_1$ and the numbers $p+1, \ldots, n$ 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.

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 which gives a lower bound of unusual character for the depth of decision trees.

(10.3) **Theorem** *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 decision tree computing $f$ is at least $n-k$.*

**Proof** Consider an arbitrary decision tree of depth $d$ that computes the function $f$, and a leaf of this tree. Here, $m \leq d$ 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 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 $k \geq n - d$. ∎

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 10.3 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 $x_1 + \cdots + x_n$, $x_1 \vee \cdots \vee x_n$ and $x_1 \wedge \cdots \wedge x_n$ 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.

(10.4) **Proposition** *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 $1 \leq j \leq n$ 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:

(10.5) **Conjecture** *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.

(10.6) **Theorem** *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

$$M = \sum f(x_1, \ldots, x_p)(p-1)^{x_1+\cdots+x_p}, \tag{10.7}$$

where the summation extends to all systems of values $(x_1, \ldots, x_p) \in \{0,1\}^p$. By an argument that is quite similar to the proof of Theorem 10.3, we can see that if $f$ is non-evasive, then $M$ is divisible by $p$.

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 $n$. 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 $p$. ■

We get important examples of weakly symmetric Boolean functions taking any **graph property**. Consider an arbitrary property of graphs, e.g. planarity; we only assume that if a graph has this property then every graph isomorphic with 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 $i, j \subset \{1, \ldots, n\}$, we 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:

(10.8) **Conjecture** *Every non-trivial monotonic graph property is evasive, i.e., every decision tree that decides such a graph property and that 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).

10.1 **Exercise** Prove that fewer measurements do not suffice in either problem 10.1.2 or problem 10.1.3. ♢

10.2 **Exercise** Show that to pick the median of $2n+1$ numbers,
(a) at least $2n$ comparisons are needed;
(b)* $O(n)$ comparisons suffice. ♢

10.3 **Exercise** 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. ♢

10.4 **Exercise** Show that the second phase of the above algorithm, i.e. the determination of the convex hull of the points $p_1, \ldots, p_i$ for $i = 2, \ldots, n$, can be performed in $O(n)$ steps provided that the points are already sorted by their $x$ coordinates. ♢

10.5 **Exercise** Give an example showing that in Proposition 10.5, the condition of monotonicity cannot be omitted. ♢

10.6 **Exercise** Given an $n$-variable Boolean function $f$, construct the following polynomial: $\Psi_f(t) = \sum f(x_1, \ldots, x_n) t^{x_1 + \cdots + x_n}$ where the summation extends to all systems of values $(x_1, \ldots, x_n) \in \{0,1\}^n$. Prove that if $f$ can be computed by a decision tree of depth $d$, then $\Psi_f(t)$ is divisible by $(t+1)^{n-d}$. ♢

10.7 **Exercise** Prove that the connectedness of a graph is a evasive property. ♢

10.8 **Exercise**
(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 $n$. ♢

10.9 **Exercise** 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 tournamentse can be considered an $\binom{n}{2}$-variable Goolean function. Prove that the tournament property that there is a 0-degree vertex is evasive. ♢

10.10 **Exercise** (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. ♢

10.11 **Exercise** 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 ordering them by magnitude, there is a $x_j$ between every pair of $y_i$'s. Prove that this needs $\Omega(n \log n)$ comparisons. ♢

# 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 characterise 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 something from this; we will only consider the case when this something is a single bit, i.e., they want to determine some property of the (whole) input. We abstract from the time- and other cost 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 $\varepsilon_1$ to the other one; then one of them (maybe the other one, maybe the same one) sends a bit $\varepsilon_2$, 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 is in Europe and Bob in New Zeeland; 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 $\varepsilon_1, \ldots, \varepsilon_k$ were sent so far (including who sent them) whether the next turn is his/her (this can only depend on the messages $\varepsilon_1, \ldots, \varepsilon_k$ 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.

## 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 $\varepsilon_1$. 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 have the same "meaning", i.e., that it divide 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 $C_2$ (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. We will call such a decomposition into submatrices 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.

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 we stop splitting it. But sometimes, it will be more useful to pretend that we keep splitting even this one: formally, we agree that an all-0 matrix consisting of 0 rows cam be split from an all-0 matrix as well as from an all-1 matrix.)

We can make the protocol even more graphic with the help of a binary tree. 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 vertex 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

formall 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 $C$.

(11.1) **Lemma**
$\kappa(C) \geq 1 + \log \mathrm{rk}(C)$.

**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. ∎

(11.2) **Corollary** *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.

(11.3) **Example** 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 $2^n \times 2^n$ unit 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:

(11.4) **Theorem** *Consider an arbitary communication protocoll deciding about two 0-1-sequence of length $n$ whether they are identical, and let $h > 0$. Then the number of sequences $a \in \{0,1\}^n$ for 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 $(b,b)$ 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 Adel'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 a solution for an earlier exercise. A **palindrome** is a string with the property that it is equal to its reverse.

(11.5) **Theorem** *Every 1-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 1-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 $\log|\Gamma|$ bits (where $\Gamma$ 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 11.4, we have therefore at most $2^{n/2}$ palindroms 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 palindrom $\alpha$ of length $2h$ and write in front of it a sequence $\beta$ of length $n-h$ and behind it a sequence $\gamma$ of length $n-h$. The sequence obtained this way is a palindrome if and only if $\beta = \gamma^{-1}$ where we denoted by $\gamma^{-1}$ the inversion of $\gamma$. By Theorem 11.4 and the above reasoning, for every $\alpha$ there are at most $2^{n/2}$ strings $\beta$ 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 $\alpha$'s is $2^h$ the assertion follows.

If we add up this estimate for all $h$ with $0 \leq h \leq n/2$ 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 $(n/2) \cdot (cn) = \Omega(n^2)$. ■

11.1 **Exercise** 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 one subset each of an $n$-element set, $X$ and $Y$. They must decide whether

(a) $X$ and $Y$ are disjoint;

(b) $|X \cap Y|$ is even.

♢

## 11.2 Some protocols

In our examples until now, even the smartest communication protocol could not outperform the trivial one (when Alice sends the complete information to Bob). In this subsection, we show a few protocols that solve their problems surprisingly cheaply by the tricky organization of communication.

(11.1) **Example** Alice and Bob know a subtree each of a (previously fixed) tree $T$ with $n$ nodes. Alice has subtree $T_A$ and Bob has subtree $T_B$. They wan to decide whether the subtrees have a common point.

The trivial protocol uses obviously $\log M$ bits where $M$ is the number of subtrees; $M$ can even be greater than $2^{n-1}$ if e.g. $T$ is a star. (For different subtrees, Alice's message must be different. If Alice gives the same message for polygons $T_A$ and $T'_A$ and, say, $T_A \not\subset T'_A$ 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 looks up the point of $T_B$ closest to $x$ (let this be $y$) and sends it to Alice. If this is in $T_A$ then Alice knows that the two trees have a common point; if $y$ is not in the tree $T_A$ then the two trees have no common points at all. This protocol uses only $1 + 2\lceil \log(n+1) \rceil$ bits. ♢

11.2 **Exercise** Prove that in Example 11.1, any protocol requires at least $\log n$ bits. ♢

*(11.3) **Exercise** Refine the above protocol to use only $\log n + \log\log n + 1$ bits. ♢

(11.2) **Example** Given is a graph $G$ with $n$ points. Alice knows a point set $S_A$ spanning a complete subgraph and Bob knows an independent $S_B$ point set in the graph. They want to decide whether the two subgraphs have a common point.

If Alice wants to give the complete information to Bob about the point 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 it 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 point 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 points 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 $\lceil \log n \rceil (2 + \lceil \log n \rceil)/2$. $\diamondsuit$

## 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 that on their part, they accept the proof. This non-deterministic protocol consists therefore of certain possible "revelations" $x_1, \ldots, x_n \in \{0,1\}^*$ all of which are acceptable for certain inputs of Alice and Bob. For a given pair of inputs, there is an $x_i$ acceptable for both of them if and only if for this pair of inputs, the answer to the communication problem is 1. The length of the longest $x_i$ 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 $\kappa_{ND}(C)$

(11.1) **Example** Suppose that Alice and Bob know a 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.) $\diamondsuit$

(11.2) **Example** In Example 11.3, 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. $\diamondsuit$

Let $x$ be a possible revelation of the superior being and let $H_x$ be the set of all possible pairs $(i, j)$ for which $x$ "convinces" the players that $c_{ij} = 1$. We note that if $(i_1, j_1) \in H_x$ and $(i_2, j_2) \in H_x$ then $(i_1, j_2)$ and $(i_2, j_1)$ also belong to $H_x$: since $(i_1, j_1) \in H_x$, Alice, possessing $i_1$, accepts the revelation $x$; since $(i_2, j_2) \in H_x$, Bob, possessing $j_2$, accepts the revelation $x$; thus, when they have $(i_1, j_2)$ both accept $x$, hence $(i_1, j_2) \in H_x$.

We can therefore also consider $H_x$ 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:

(11.3) **Lemma** *$\kappa_{ND}(C)$ is the smallest natural number $t$ for which the 1's of the matrix can be covered with $2^t$ all-1 submatrices.*

In the negation of Example 11.2, the matrix $C$ is the $2^n \times 2^n$ unit matrix. Obviously, only the $1 \times 1$ 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 the nondeterministic communication complexity of $C$ is therefore at most $s - 1$. Hence

$$\kappa_{ND}(C) \leq \kappa(C) - 1.$$

Example 11.2 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 11.2 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 (Aho, Ullman and Yannakakis) shows that here, already, the difference between the two sides of the inequality cannot be too great.

(11.4) **Theorem**

$$\kappa(C) \leq (2 + \kappa_{ND}(C)) \cdot (2 + \kappa_{ND}(\overline{C})).$$

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 implies

$$\log \varrho(C) \leq \kappa_{ND}(C).$$

The following inequality therefore implies theorem 11.4.

(11.5) **Theorem**
$\kappa(C) \leq 1 + \log \varrho(C)(\kappa_{ND}(\overline{C}) + 2)$.

**Proof** We use induction on $\log \varrho(C)$. If $\varrho(C) \leq 1$ 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:

(11.6) **Claim**
$\varrho(A_i) + \varrho(B_i) \leq \varrho(C)$.

11.4 **Exercise** Prove this claim. ♢

Now, we can prescribe 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 and the first phase of 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 phas 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):

(11.7) **Corollary**
$\kappa(C) \leq 1 + \log(1 + \mathrm{rk}(C))(2 + \kappa_{ND}(\overline{C}))$.

11.5 **Exercise** Show that in Theorems 11.4 and 11.5 and in Corollary 11.7, with more careful planning, the factor $(2 + \kappa_{ND})$ can be replaced with $(1 + \kappa_{ND})$. ♢

To show the power of Theorems 11.4 and 11.5 consider the examples treated in Subsection 11.2. If $C$ is the matrix corresponding to Example 11.1 (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 the theorems 11.4 and 11.5 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 subsection 11.2.

Let now $C$ be the matrix corresponding to Example 11.2. 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 11.5, 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 know for the value of $\kappa_{ND}(C)$, is only the estimate $\kappa_{ND} = O((\log n)^2)$ coming from the inequality $\kappa_{ND} \leq \kappa$.

(11.8) **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 $\{ \neg C \in \mathcal{H} \}$ is in $\mathrm{NP}^{\mathrm{comm}}$. Then Example 11.2 shows that

$$\mathrm{P}^{\mathrm{comm}} \neq \mathrm{NP}^{\mathrm{comm}},$$

and Theorem 11.4 implies

$$\mathrm{P}^{\mathrm{comm}} = \mathrm{NP}^{\mathrm{comm}} \cap \mathrm{co{-}NP}^{\mathrm{comm}}.$$

♢

## 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 $n$.

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.

(11.1) **Protocol** Alice chooses a random prime number $p$ in the interval $1 \leq p \leq N$ and divides $x$ by $p$ with remainder. Let the remainder by $r$; then Alice sends Bob the numbers $p$ and $r$. Bob checks whether $y \equiv r \pmod{p}$. If not then he determines that $x \neq y$. If yes then he concludes that $x = y$. $\diamondsuit$

First we note that this protocol uses only $2 \log N$ 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.

(Now we will use some number-theoretical facts. For those who are unfamiliar with them but feel the need for completeness we include a proof of some weaker but still satisfactory versions of these facts in the next section.)

It is a known number-theoretical fact (see the next section for the proof) that for large enough $q$ we have, say,

$$2 \cdot 3 \cdot 5 \cdot \cdots \cdot q > e^{\frac{3}{4}q} > 2^q$$

(Lovász has $2^{q-1}$.)

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:

$$\mathrm{Prob}(p \mid d) = \frac{k}{\pi(N)} \leq \frac{\pi(n)}{\pi(N)}.$$

Now, according to the prime number theorem, we have $\pi(n) \asymp n/\log n$ 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 $2 \log N = 2 \log n +$ constant.

(11.2) **Remark** The use of randomness does not help in every communication problem this much. We have seen in one of 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. $\diamondsuit$

# 12 The complexity of algebraic computations

(To be written)

# 13 Circuit complexity

## 13.1 Introduction

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

$$\text{PARITY}(x_1, \ldots, x_n) = x_1 + x_2 + \ldots + x_n \pmod 2.$$

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.

At this point, we have to make an important remark: if we restrict the fan-in (indegree) of the Boolean circuit to 2, then it is easy to see that we need at least $\log n$ depth, just to get an output node which is dependent on all the inputs (see exercise). However, we will allow arbitrary fan-in and fan-out (this is analogous to the concurrent-read-concurrent-write model of parallel computation).

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 such a 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 the 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).

(13.1) **Theorem** *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 truely interested.

## 13.2 Lower bound for the Majority Function

Let us start with the exact statement of the theorem.

(13.1) **Theorem** *If $C$ is a circuit of depth $d$, with* AND, OR, XOR, *and* NOT *gates that computes the majority function of $n$ input bits, then $C$ has at least $2^{n^{(1/2d)}}/10\sqrt{n}$ gates.*

We would not really need all these types of gates (just like we don't need both AND and OR gates in a Boolean circuit). We can assume without loss of generality that the circuit uses only NOT, XOR and OR gates, since these can be used to simulate AND gates within constant depth and polynomial size (see exercise).

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. It is easy to see that 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 (see exercise). 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 $h$.

Each Boolean function can be expressed as a polynomial over the two-element field $GF(2)$. 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 $p_1 p_2$ 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.

(13.2) **Lemma** *Let $g_1, \ldots, g_m$ be Boolean functions of degree at most $h$. 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 $\alpha$; 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. ∎

Next we show that any function of low degree has to differ from the $k$-threshold function on a significant fraction of the inputs.

(13.3) **Lemma** *Let $n/2 \leq k \leq n$. 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 $x_{i_1} \ldots x_{i_r}$ 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 $\mathcal{A}$, 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 $\mathcal{B}$,

$$\sum_{\substack{b \le a_1 \\ b \in \mathcal{B}}} m_{a_2 b} = \sum_{\substack{b \le a_1 \wedge a_2 \\ b \in \mathcal{B}}} 1 = \sum_{u \le a_1 \wedge a_2} (f_k(u) + g(u)) = \sum_{u \le a_1 \wedge a_2} f_k(u) + \sum_{u \le a_1 \wedge a_2} g(u).$$

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. 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 13.2 with $r = \lfloor n^{1/(2d)} \rfloor$ to approximate the OR 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 13.3 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 $s 2^{n-r} \ge \binom{n}{k}$. From this, routine calculations yield that

$$s \ge \binom{n}{k} 2^{r-n} > \frac{2^r}{\sqrt{\pi n}},$$

which establishes the desired exponential lower bound.

## 13.3 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 $x_{ij}$, $1 \le i < j \le n$, 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.

13.1 **Exercise** 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 $\log n$. $\diamondsuit$

13.2 **Exercise** (a) Show that the negation of a variable can be computed by using only XOR gates, if we can use the constant 1. (b) Show that the AND of $n$ variables can be computed by a circuit using NOT, OR and XOR gates, with constant depth and polynomial size. $\diamondsuit$

13.3 **Exercise** 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. $\diamondsuit$

# 14 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 8. 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.

## 14.1 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 "code" 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 interceptor 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).

(14.1) **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. $\diamondsuit$

## 14.2 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 imposter 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!

(14.1) **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 wich 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.

♢

## 14.3 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

$$f(f(x, e_i), d_i) = f(f(x, d_i), e_i) = x. \tag{14.1}$$

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 useable, 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.

(14.2) *Only $j$ can decode a message addressed to $j$.*

**Proof** Assume that a band $k_1, \ldots, k_r$ of unauthorized participants finds 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 $k_1, \ldots, k_r$ and $i$ together could compute $x$ also from $f(x, e_j)$. Let, namely, $z = f(x, e_j)$; then $k_1, \ldots, k_r$ and $i$ knows the message $f(x, e_j) = f(f(z, d_i), e_j)$ and thus using the method of $k_1, \ldots, k_j$, 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:

(14.3) *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 $i$.*

(14.4) *$j$ 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.*

(14.5) *$j$ 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 function have been proposed; many of the proposed systems turned out insecure later on—the corresponding complexity conditions were not true.) In the next subsection, 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).

## 14.4 The Rivest-Shamir-Adleman 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 $1 \leq e_i < m$ 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 $1 \leq d_i < m$ 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 $0 \leq x < m$ (if it is longer then it will be cut into pieces). The encoding function is defined by the formula

$$f(x, e) = x^e \pmod{m} \qquad 0 \leq f(x, e) < m.$$

The same formula serves for decoding, only with $d$ in place of $e$.

The inverse relation between coding and decoding (formula 14.1) 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 $x$ and $e_i$ 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 of $e_i$ and $e_j d_j - 1$, then the greatest commond divisor $v_2$ of $e_k$ and $(e_j d_j - 1)/v_1$, then the greatest commond divisor $v_3$ of $(e_j d_j - 1)/(v_1 v_3)$, 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 $x$.

14.1 **Exercise** 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 descibe 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

$$e_i d_i \equiv 1 \pmod{(p_i - 1)(q_i - 1)}$$

(here, $(p_i - 1)(q_i - 1) = \varphi(m_i)$, the number of positive integers smaller than $m_i$ and relatively prime to it). 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.

**Literature**


[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullmann. *Design and Analysis of Computer Algorithms.* Addison-Wesley, New York, 1974.

[2] Peter Bürgisser, Michael Clausen and M. Amin Shokrollahi, *Algebraic Complexity Theory.* Springer Verlag, Berlin–Heidelberg, 1997.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Algorithms.* Mc Graw-Hill, New York, 1990.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[5] Raymond Greenlaw and H. James Hoover. *Fundamentals of the Theory of Computation* Morgan Kauffmann Pub. Inc., San Francisco, 1998.

[6] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, New York, 1979.

[7] Donald E. Knuth. *The Art of Computer Programming, I-III.* Addison-Wesley, New York, 1969-1981.

[8] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, New York, 1981.

[9] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.

[10] Christos H. Papadimitriou and K. Stieglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, New York, 1982.

[11] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley, New York, 1986.

[12] Robert Sedgewick. *Algorithms.* Addison-Wesley, New York, 1983.

[13] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Co. Boston, 1996.

[14] Klaus Wagner and Gert Wechsung. *Computational Complexity.* Reidel, New York, 1986.