Computation Complexity

László Lovász
translation, additions, modifications by
Péter Gács

March 15, 1994

Lecture notes for a one-semester graduate course. Part of it is also suitable for an undergraduate course, at a slower pace. Mathematical maturity is the main prerequisite.

Contents

1 Introduction

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). 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 important classification even for the problems arising in classical areas of mathematics; this classification reflects well the practical and theoretical difficulty of problems. The relation of different machine models to each other 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).

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 some introduced 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 related to it. 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 result.

It is, finally, worth remarking that if a problem turns out to have only "difficult" solutions, this is not necessarily a negative result. More and more areas (random number generation, communication protocols, secret communication, 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 cryptography, the theory of secret communication.

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

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

Let us define some orderings of the set of words. Suppose that an ordering of the elements of Σ is given. In the *lexicographic ordering* of the elements of Σ^*, a word α precedes a word β if either α is a prefix (beginning segment) of β or the first letter which is different in the two words is smaller in α. (E.g., 35244 precedes 35344 which precedes 353447.) The lexicographic ordering does not order all words in a single sequence: for example, every word beginning with 0 precedes the word 1 over the alphabet $\{0,1\}$. The *increasing order* is therefore often preferred: here, shorter words precede longer ones and words of the same length are ordered lexicographically. This is the ordering of $\{0,1\}^*$ we get when we write 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 $|f(n)| \le c|g(n)|$. 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 $c_1 g(n) \le f(n) \le c_2 g(n)$. 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 $O(n^2) \ne O(n)$. When such formulas become too complex it is better to go back to some more explicit notation.

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

2 Models of computation

In this section, we will treat the concept of algorithm. This concept is fundamental for our topic but still, we will not define it. Rather, we consider it an intuitive notion which is amenable to various kinds of formalization (and thus, investigation from a mathematical point of view). 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 convince you that a general agreement can be achieved on these matters. This agreement is often formulated as *Church's thesis*. A program in the Pascal programming language is a good example of an algorithm specification.

Since the "mechanical" nature of an algorithm is the most important one, instead of the notion of algorithm, we will introduce various concepts of a *mathematical machine*. The mathematical machines that we will consider will be used to compute some *output* from some *input*. The input and output can be e.g. 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 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: Goodman, Zeldovich, Fekete. The output is also a string: Fekete, Goodman, Zeldovich. The problem is to compute a *function* assigning an output string to each input string. Both input and output have unbounded size.
>
> In this course, we will concentrate on this type of problem, though there are other possible ones. If e.g. a database must be designed then several functions must be considered simultaneously. Some of them will only be computed once while others, over and over again while new inputs arrive.

In general, a typical generalized algorithmic problem is not just one problem but an infinite family of problems where the inputs can have arbitrarily large size. Therefore we either must consider an infinite family of finite computers of growing size or some ideal, infinite computer. The latter approach has the advantage that it avoids the questions of what infinite families are allowed. (The usual model of finite automata, when used to compute functions with arbitrary-size inputs, is too primitive for the purposes of complexity theory since it is able to compute only very simple functions. The example problem (sorting) above cannot be solved by such a finite automaton, since the intermediate memory needed for computation is unbounded.)

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 program-controlled 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 investigations. It is less appropriate for the definition of concrete algorithms since its description is awkward, and mainly since it differs from existing computers in several important aspects.

The most essential clumsiness distinguishing the Turing machine from real computers is that its memory is not accessible immediately: in order to read a "far" memory cell, all intermediate cells must also be read. This is bridged over 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 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.

Maybe the simplest idea for an infinite machine is to connect an infinite number of similar automata into an array (say, a one-dimensional one). Such automata are called **cellular automata**.

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.

2.1 The Turing machine

The notion of a Turing machine One tendency of computer development is to build larger and larger finite machines for larger and larger tasks. Though the detailed structure of the larger machine is different from that of the smaller ones certain general principles guarantee some uniformity among computers of different size. For theoretical purposes, it is desirable to pursue this uniformity to the limit and to design a computer model infinite in size, accomodating therefore tasks of arbitrary size. A good starting point seems to be to try to characterize those functions over the set of natural numbers computable by an idealized human calculator. Such a calculator can hold only a limited amount of information internally but has access to arbitrary amounts of scratch paper. This is the idea behind Turing machines. In short, a Turing machine is a finite-state machine interacting with an infinite tape through a device that can read the tape, write it or move on it in unit steps.

Formally, a **Turing machine** consists of the following.

(a) $k \geq 1$ tapes 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 can be written from a finite alphabet Σ. With the exception of finitely many cells, this symbol must be a special symbol $*$ of the alphabet, denoting the "empty cell".

(b) A **read-write head**, positioned in every step over a cell of the tape, belongs to every tape.

(c) A control unit with a set of possible states from a finite set Γ. There is a distinguished starting state "START" and ending state "STOP".

Initially, the control unit is in the "START" state, and the heads rest over the starting cells of the tapes. In every step, each head reads the symbol found in the given cell of the tape; depending on the symbols read and its own state, the control unit does one of the following 3 things:

— makes a transition into a new state (this can be the same as the old one, too);

— directs each head to overwrite the symbol in the tape cell it is scanning (in particular, it can give the direction to leave it unchanged);

— directs each head to move one step right or left, or to stay in place.

The machine halts when the control unit reaches the state "STOP".

Mathematically, the following data describe the Turing machine: $T = \langle k, \Sigma, \Gamma, \alpha, \beta, \gamma \rangle$ where $k \geq 1$ is a natural number, Σ and Γ are finite sets, $* \in \Sigma$ and $START, STOP \in \Gamma$, further

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

are arbitrary mappings. Among these, α gives the new state, β the symbols to be written on the tape and γ shows how much the head moves. In what follows we fix the alphabet Σ and assume that it consists, besides the symbol $*$, of at least two symbols, say it contains 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 understand the words initially written on the tapes. We always assume that these are written on the tapes starting from 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 Σ^*. Most frequently, we write a non-empty word only on the first tape for input. If we say that the input is a word x then we understand that the input is the k-tuple $(x, \emptyset, \dots, \emptyset)$.

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 without any previous agreement, we refer to a single word as output, then we understand the word on the last tape.

It is practical to assume that the input words do not contain the symbol $*$. Otherwise, it would not be possible to know where is the end of the input: a simple problem like "find out the length of the input" would not be solvable, it would be useless for the head to keep stepping right, it would not know whether the input has already ended. We denote the alphabet $\Sigma \setminus \{*\}$ by Σ_0. (We could 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.

2.1 **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, "WINNIEPOOH".

$\diamondsuit$

2.2 **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 $f \circ g$. $\diamondsuit$

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

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

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 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 some intuition will be provided.

2.5 **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. $\diamondsuit$

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 show now 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 program-controlled computers but they will also play an important role in many proofs.

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

We say that a $(k+1)$-tape Turing machine is **universal** (with respect to k-tape Turing machines) if for every k-tape Turing machine S over Σ, there is a word (program) p with which T simulates S.

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

Proof The basic idea of the construction of a universal Turing machine is that on tape $k+1$, we write a table describing the work of the Turing machine S to be simulated. Besides this, the universal Turing machine T writes it up for itself, which state of the simulated machine S it 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.

We give the exact construction by first using $k+2$ tapes. For the sake of simplicity, assume that Σ contains the symbols "0", "1", "–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 Σ_0^*. Let the "code" of a given position of machine S be the following word:

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

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 will 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 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-STATE, MOVE and AGAIN. Let $h(i)$ denote the symbol read by the i-th head ($1 \le i \le 3$). We describe the functions α, β, γ by the table in Figure 2.1 (wherever we do not specify a new state the control unit stays in the old one). In the typical run in Figure 2.2, 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.

We can get rid of tape $k+2$ easily: its contents (which is always just r cells) will be placed on cells –1,–2,..., $-r$. It is a problem, however, that we still need two heads on this tape: one moves on its positive half, and one on the negative half. We solve this by doubling each cell: the symbol written into it originally stays in its left half, and in its right half there is a 1 if the head would rest there, and a 0 if two heads would rest 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. ∎

2.6 **Exercise** Show that if we simulate a k-tape machine on the above constructed $(k+1)$-tape 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. ♢

2.7 **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. ♢

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-STATE:

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 2.1: A universal Turing machine

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

Figure 2.2: Example run of the universal Turing machine

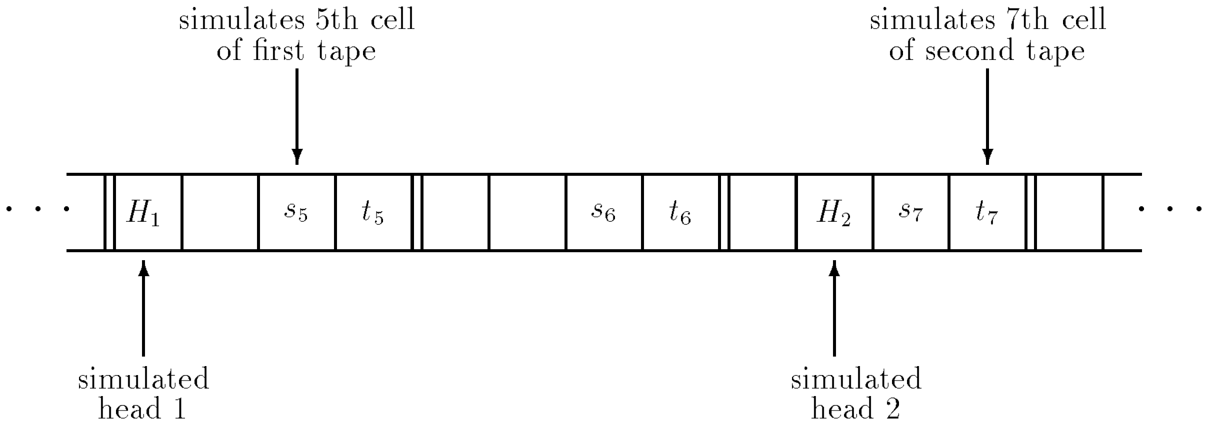


Figure 2.3: One tape simulating two tapes

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

(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 $1, 2, \dots, 2k-1$. Then starting from the last symbol, it moves every symbol in the last block of nonblanks $2k$ positions to right, etc.

Now, position $2ki + 2j - 2$ $(1 \le j \le k)$ 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. ■

2.8 **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. ♢

*(2.9) **Exercise** Show that if we allow a two-tape Turing machine in Theorem 2.1.2 then the number of steps increases less: every k-tape Turing machine can be replaced 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)] ♢

2.10 **Exercise** Let the language $\mathcal{L}$ consist of "palindromes":

$$\mathcal{L} = \{\, x_1 \dots x_n x_n \dots x_1 : x_1 \dots x_n \in \Sigma_0^* \,\}.$$

(a) There is a 2-tape Turing machine deciding about a word of length N in $O(N)$ steps, whether it is in $\mathcal{L}$.

*(b) Every 1-tape Turing machine needs $\Omega(N^2)$ steps to decide this.

♢

2.11 **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.

♢

*(2.12) **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.] $\diamondsuit$

2.13 **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.

$\diamondsuit$

2.2 The Random Access Machine

The Random Access Machine is a hybrid machine model close to real computers and often convenient for the estimation of the complexity of certain algorithms. The main point where it is more powerful is that it can reach its memory registers immediately (for reading or writing).

Unfortunately, these advantageous properties of the RAM machine come at a cost: in order to be able to read a memory register immediately, we must address it; the address must be stored in another register. Since the number of memory registers is not bounded, the address is not bounded either, and therefore we must allow an arbitrarily large number in the register containing the address. The content of this register must itself change during the running of the program (indirect addressing). This further increases the power of the RAM; but by not bounding the number contained in a register, we are moved again away from existing computers. If we do not watch out, "abusing" the operations with big numbers possible on the RAM, we can program algorithms that can be solved on existing computers only harder and slower.

The RAM machine has a program store and a memory. The memory is an infinite sequence $x[0], x[1], \ldots$ of memory registers. Each register can store an arbitrary integer. (We get bounded versions of the Random Access Machine by restricting the size of the memory and the size of the numbers stored.) At any given time, only finitely many of the numbers stored in memory are different from 0.

The program storage consists also of an infinite sequence of registers called **lines**. We write here a program of some finite length, in a certain programming language similar to the machine language of real machines. It is enough, e.g., to permit the following instructions:

$$x[i] := 0; \qquad x[i] := x[i] + 1; \qquad x[i] := x[i] - 1;$$
$$x[i] := x[i] + x[j]; \qquad x[i] := x[i] - x[j];$$
$$x[i] := x[x[j]]; \qquad x[x[i]] := x[j];$$
if $x[i] \le 0$ **then goto** p.

Here, i and j are the number of some memory register (*i.e.* an arbitrary integer), p is the number of some program line (*i.e.* an arbitrary natural number). The instruction before the last one guarantees the possibility of immediate addressing. With it, the memory behaves like an array in a conventional programming language like Pascal. What are exactly the basic instructions here 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 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.

The number of steps of the Random Access Machine is not the best measure of the "time it takes to work". Due to the fact that the instructions operate on natural numbers of arbitrary size, tricks are possible that are very far from practical computations. For example, we can simulate vector operations by the adddition of two very large natural numbers. One possible remedy is to permit only operations even more primitive than addition: it is enough to permit $x[i] := x[i] + 1$ (see the exercise on the Pointer Machine below). The other possibility is to speak about the **running time** of a RAM instead of its number of steps. We define this by counting as the time of a step not one unit but as much as the number of binary digits of the natural numbers occurring in it (register addresses and their contents). (Since these are essentially base two logarithms, it is also usual to call this model **logarithmic cost** RAM.)

Sometimes, the running time of some algorithms is characterized by two numbers. We would say that "the machine makes at most n steps on numbers with at most k (binary) digits"; this gives therefore a running time of $O(nk)$.

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

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

(b) computes its base 2 representation;

$\diamondsuit$

2.15 **Exercise** Let $p(x) = a_0 + a_1x + \cdots + a_nx^n$ 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 $K = \max\{|a_0|, \ldots, |a_n|\}$. $\diamondsuit$

Now we show that the RAM and Turing machines can compute essentially the same 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 be 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 first the first interpretation.

(2.2.1) **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 P_i $(1 \le i \le r)$ and Q_{ij} $(1 \le i \le r-1, 0 \le j \le 2)$. The program part P_i below $(1 \le i \le r-1)$ simulates the event when the control unit of the Turing machine is in state i and the machne reads out what number is on cell $x[1]/2$ of the tape. Depending on this, it will jump to different places in the program:

> $x[3] := x[x[1]]$;
> **if** $x[3] \le 0$ **then goto** [the address of Q_{i0}];
> $x[3] := x[3] - 1$;
> **if** $x[3] \le 0$ **then goto** [the address of Q_{i1}];
> $x[3] := x[3] - 1$;
> **if** $x[3] \le 0$ **then goto** [the address of Q_{i2}];

Let the program part P_r consist of a single empty program line. The program part Q_{ij} below overwrites the $x[1]$-th register according to the rule of the Turing machine, modifies $x[1]$ according to the movement of the head, and jumps to the corresponding program part P_i.

$$
\begin{aligned}
&x[3] := 0;\\
&\qquad \left.\begin{aligned} &x[3] := x[3]+1;\\ &\vdots\\ &x[3] := x[3]+1; \end{aligned}\right\} \beta(i,j)\textbf{-times}\\
&x[x[1]] := x[3];\\
&x[1] := x[1] + \gamma(i,j);\\
&x[1] := x[1] + \gamma(i,j);\ x[3] := 0;\ \textbf{if}\ x[3] \le 0\ \textbf{then goto}\ [\text{the address of } P_{\alpha(i,j)}];
\end{aligned}
$$

(Here, instruction $x[1] := x[1] + \gamma(i,j)$ must be understood in the sense that we take instruction $x[1] := x[1] + 1$ or $x[1] := x[1] - 1$ if $\gamma(i,j) = 1$ or -1 and omit it if $\gamma(i,j) = 0$.) The program itself looks as follows.

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

With this, we have described the "imitation" of the Turing machine. To estimate the running time, 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 $O(\log N)$. ∎

(2.2.2) **Remark** The language of the RAM is similar to (though much simpler than) the machine language of real computers. If more advanced programming constructs are desired a compiler must be used. ♢

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 compute the digits of a with the help of a simple program, write these (deleting the 1 found in the first position) into the registers $x[0], \ldots, x[n-1]$ (see Exercise 2.14) and apply the construction described in Theorem 2.2.1.

(2.2.3) **Remark** In the proof of Theorem 2.2.1, we did not use the instruction $x[i] := x[i] + x[j]$; this instruction is needed only in the solution of Exercise 2.14. Moreover, even this exercise would be solvable 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, moreover, 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 few steps; its running time, even in case of logarithmic costs, is only approx. $\log_2|a| + \log_2|b|$. But if we exclude the instruction $x[i] := x[i] + x[j]$ then the time it needs is at least $\min\{|a|, |b|\}$ (since every other instruction increases the absolute value of the largest stored number by at most 1). $\diamondsuit$

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.

(2.2.4) **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 "*"). It would cause a problem, however, that the RAM can write even into the register with number 2^N using only time N, according to the logarithmic cost. 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 $x[i] \le 0$ 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 $O(N^2)$. ∎

2.16 **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 $R(u,\langle p,x\rangle) = R(p,x)$. $\diamondsuit$

2.17 **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 $1,\dots,r$, leaving it. When the head scans a certain cell it can move to any of the cells $\lambda(c,i)$ $(i = 1,\dots,r)$ 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 2.2.3 says for this simulation. $\diamondsuit$

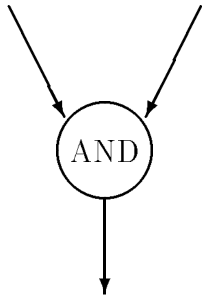


Figure 2.4: A node of a logic circuit

2.3 Boolean functions and logic circuits

Let us look inside a computer, (actually inside an integrated circuit, with a microscope). Discouraged by a lot of physical detail irrelevant to abstract notions of computation, we will decide to look at the blueprints of the circuit designer, at the stage when it shows the smallest elements of the circuit still according to their computational functions. We will see a network of lines that can be in two states, *high* or *low*, or in other words True or False, or, as we will write, 1 or 0. The nodes at the junction of the lines have the forms like in Figure 2.4 and some others. These **logic components** are familiar to us. Thus, at the lowest level of computation, the typical computer processes **bits**. Integers, floating-point numbers, characters are all represented as strings of bits, and the usual arithmetical operations are *composed* of bit operations. Let us see, how far the concept of bit operation gets us.

A **Boolean function** is a mapping $f : \{0,1\}^n \to \{0,1\}$. The values 0,1 are sometimes identified with the value False, True and the variables in $f(x_1,\dots,x_n)$ are sometimes called **logic variables**, **Boolean variables** or **bits**. In many algorithmic problems, there are n input logic 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 $\binom{N}{2}$ logic variables: the nodes are numbered from 1 to N and x_{ij} $(1 \le i < j \le N)$ is 1 if i and j are connected and 0 if they are not. The value of the function $f(x_{12}, x_{13}, \dots, x_{n-1,n})$ 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 $\neg x$. We mention only the following 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**, or logical OR

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

the **binary addition**

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

The mentioned operations are connected by a number of useful identities. All three mentioned binary operations 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**.

(2.3.1) **Lemma** *Every Boolean function is expressible using 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 degenerated 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.

(2.3.2) **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. ♢

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

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 a nontrivial conjunctive normal form is never a tautology.

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.

2.18 **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. ♢

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

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

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

♢

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

There are cases when a Boolean function can be computed fast but it can only expressed by a very large Boolean polynomial. This is because 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.

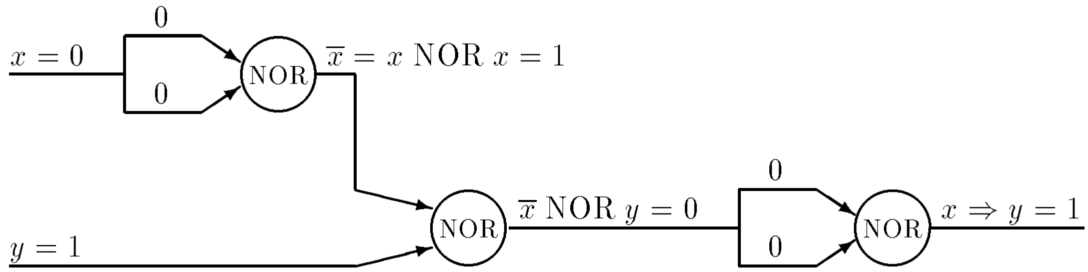


Figure 2.5: A NOR circuit computing $x \Rightarrow y$, with assignment on edges

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 logic 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 **logic 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 logic 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.

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

It is sometimes useful to assign values to the edges as well: the value assigned to an edge is the one assigned to its start node. If the longest path leaving the input nodes has length t then the assignment procedure can be performed in t **stages**. In stage k, all edges reachable from the inputs in k steps receive their values.

(2.3.3) **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 2.5. $\diamondsuit$

(2.3.4) **Example** An important Boolean circuit with many outputs. For a natural number n we can construct a circuit that will compute all the functions $E_{a_1,\dots,a_n}(x_1,\dots,x_n)$ (as defined above in the proof of Lemma 2.3.1) for all values of the vector $(a_1,\dots,a_n)$. This circuit is called the **decoder circuit** since it has the following behavior: for each input $x_1,\dots,x_n$ only one output node, namely $E_{x_1,\dots,x_n}$ 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 such a circuit is given for n. To obtain one for $n+1$, we split each output $y = E_{a_1,\dots,a_n}(x_1,\dots,x_n)$ in two, and form the new nodes

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

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 gate computes the output immediately from the input. The notion of logic circuits will be important for us 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 often restricted to 2 or to some fixed maximum. (Sometimes, bounds are also imposed on the out-degree. This means that a partial result cannot be "freely" distributed to an arbitrary number of places.)

(2.3.5) **Remark** Logic circuits (and in particular, Boolean circuits) can model two things. First, they give a combinatorial description of certain simple (feedbackless) electronic networks. Second—and this is more important from out point of view— they give a good description of the logical structure of algorithms. We will prove a general theorem (Theorem 2.3.6) later about this but very often, a logic circuit provides an immediate, transparent description of an algorithm.

The nodes correspond to the operations of the algorithm. The order of these operations is restricted only by the directed graph: the operation at the end of a directed edge cannot be performed before the one at the start of the edge. This description helps when we want to parallelize certain algorithms. If a function is described by a logical circuit of depth h and size n then one processor can compute it in time $O(n)$, but many (at most n) processors might be able to compute it even in time $O(h)$ (provided that we can solve well the connection and communication of the processors; we will consider parallel algorithms in Section 8). $\diamondsuit$

2.22 **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. $\diamondsuit$

2.23 **Exercise** Prove that for every logic 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. $\diamondsuit$

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

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

be its representation by a disjunctive normal form. This representation corresponds to a depth 2 Boolean circuit in the following manner: let its input points correspond to the variables $x_1, \dots, x_n$ and the negated variables $\overline{x}_1, \dots, \overline{x}_n$. 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.

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

2.25 **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. $\diamondsuit$

We can consider each Boolean circuit as an algorithm serving to compute some Boolean function. It can be seen immediately, however, that logic circuits "can do" less than e.g. Turing machines: a Boolean 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 Boolean 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.

(2.3.6) **Theorem** *For every Turing machine T and every pair $n, N \ge 1$ of numbers there is a Boolean circuit with n inputs, depth $O(N)$, indegree at most 2, that on an input $(x_1, \dots, x_n) \in \{0,1\}^n$ 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 $n, N \ge 1$. 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 $0 \le t \le N$, $g \in \Gamma$, $h \in \Sigma$ and $-N \le p \le N$. 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 $s_0, \dots, s_{n-1}$ 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 $0,\dots,n-1$ 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:

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

It can be seen that these recursions can be taken as logical functions which turn the graph into a logic 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. ∎

(2.3.7) **Remark** Our construction of a universal Turing machine in Theorem 2.1.1 is, in some sense, inefficient and unrealistic. For most commonly used transition functions α,β,γ, a table is namely a very inefficient way to express them. In practice, a finite control is generally given by a logic circuit (with a Boolean vector output), which 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 logic circuit computing it, along with a specific state of this circuit. Each stage of the simulation would first simulate the logic circuit to find the values of the functions α,β,γ and then proceed as before. ♢

2.26 **Exercise** Universal circuit. For each n, construct a Boolean circuit whose gates have indegree ≤ 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 2.3.4.] ♢

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

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

♢

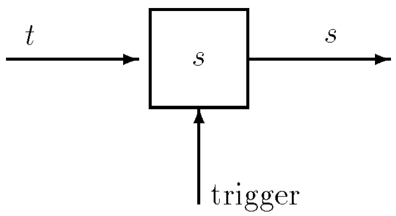


Figure 2.6: A shift register

2.4 Finite-state machines

Clocked circuits. The most obvious element of ordinary computations missing from logic circuits is *repetition.* Repetition requires *timing* of the work of computing elements and *storage* of the computation results between consecutive steps. Let us look at the drawings of the circuit designer again. We will see components with one ingoing edge, of the form like in Figure 2.6, called **shift registers**. The shift registers are controlled by one central **clock** (invisible on the drawing). At each clock pulse, the assignment value on their incoming edge jumps onto their outgoing edges and becomes the value "in" the register.

A **clocked circuit** is a directed graph with numbered nodes where each node is either a logic component, a shift register, or an input or output node. There is no directed cycle going through only logic components.

How to use a clocked circuit for computation? We write some initial values into the shift registers and input edges, and propagate the assignment using the logic components, for the given clock cycle. Now we send a clock pulse to the register, and write new values to the input edges. After this, the new assignment is computed, etc.

(2.4.1) **Example** An adder. This circuit (see Figure 2.7) computes the sum of two binary numbers x, y. We feed the digits of x and y beginning with the lowest-order ones, at the input nodes. The digits of the sum come out on the output edge. $\diamondsuit$

How to compute a *function* with the help of such a circuit? Here is a possibility. We enter the input, either parallelly or serially (in the latter case, we signal at extra input nodes when the first and last digits of the input are entered). Now we run the circuit, until it signals at an extra output node when the output can be received from the other output nodes. After that, we read out the output, again either parallelly or serially (in the latter case, one more output node is needed to signal the end).

In the adder example above, the input and output are processed serially, but for simplicity, we omitted the extra circuitry for signaling the beginning and the end of the input and output.

Finite-state machines. The clocked circuit's future responses to inputs depend only on the inputs and the present contents of its registers. Let $s_1, \dots, s_k$ be this content. To a

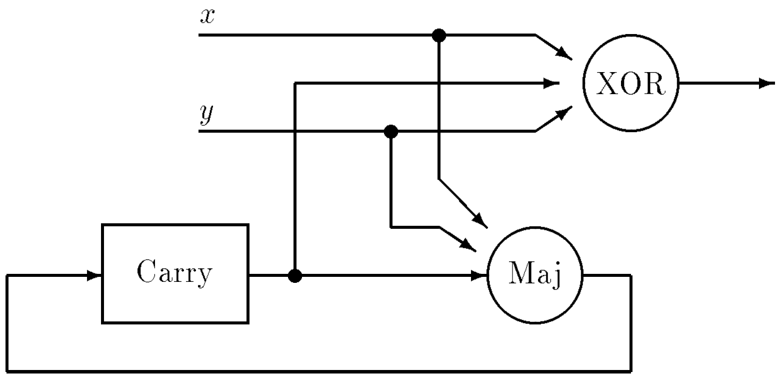


Figure 2.7: A binary adder

circuit with n binary inputs and m binary outputs let us denote the inputs at clock cycle t by $x^t = (x_1^t, \dots, x_n^t)$, the state by $s^t = (s_1^t, \dots, s_k^t)$, and the output by $y^t = y_1^t, \dots, y_k^t)$. Then to each such circuit, there are two functions

$$\begin{aligned} \lambda &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m, \\ \delta &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k \end{aligned}$$

such that the behavior of the circuit is described by the equations

$$\begin{aligned} y^t &= \lambda(s^t, x^t), \\ s^{t+1} &= \delta(s^t, x^t). \end{aligned} \tag{2.4.2}$$

From the point of view of the user, only these equations matter, since they describe completely, what outputs to expect from what inputs. A device described by equations like (2.4.2) is called a **finite-state machine**, or **finite automaton**. The finiteness refers to the finite number of possible values of the state s^t. This number can be very large: in our case, 2^k.

(2.4.3) **Example** For the binary adder, let u^t and v^t be the two input bits at time t, let c^t be the content of the carry, and w^t be the output at time t. Then the equations (2.4.2) now have the form

$$\begin{aligned} w^t &= u^t \oplus v^t \oplus c^t, \\ c^{t+1} &= \mathrm{Maj}(u^t, v^t, c^t). \end{aligned}$$

$\diamondsuit$

Not only every clocked circuit is a finite automaton, but every finite automaton can be implemented by a clocked circuit.

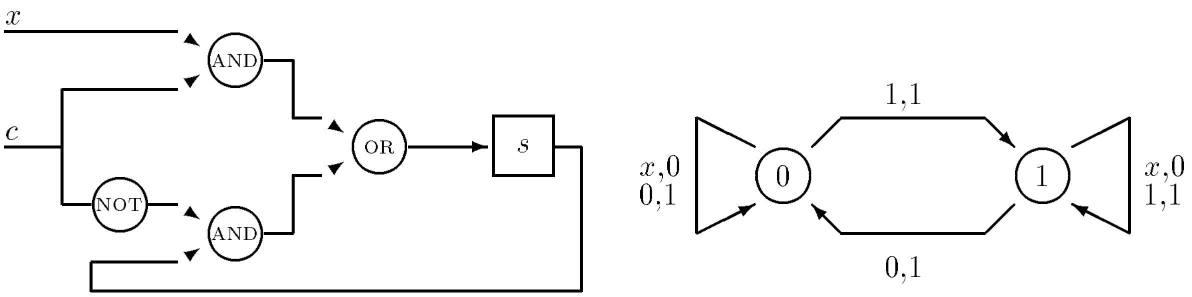


Figure 2.8: Circuit and state-transition diagram of a memory cell

(2.4.4) **Theorem** *Let*

$$\begin{aligned}\lambda &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m,\\ \delta &: \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k\end{aligned}$$

be functions. Then there is a clocked circuit with input $x^t = (x_1^t, \ldots, x_n^t)$*, state* $s^t = (s_1^t, \ldots, s_k^t)$*, output* $y^t = y_1^t, \ldots, y_k^t)$ *whose behavior is described by the equations* (2.4.2),

Proof The circuit has k shift registers, which at time t will contain the string s^t, and n input nodes, where at time t the input x^t is entered. It contains two logic circuits. The input nodes of both of these logic circuits are the shift registers and the input nodes of the whole clocked circuit. The output nodes of the first logic circuit B are the output nodes of the clocked circuit. We choose this circuit to implement the function $\lambda(s, x)$. The output nodes of the other circuit C are the inputs of the shift registers. This logic circuit is chosen to implement the function $\delta(s, x)$. Now it is easy to check that the clocked circuit thus constructed obeys the equations (2.4.2). ∎

Traditionally, the state-transition functions of a finite-state machine are often illustrated by a so-called **state-transition diagram**. In this diagram, each state is represented by a node. For each possible input value x, from each possible state node s, there is a directed edge from node s to node $\delta(s, x)$, which is marked by the pair $(x, \lambda(s, x))$.

(2.4.5) **Example** A memory cell (flip-flop, see Figure 2.8). This circuit has just one register with value s which is also the output, but it has two input lines x and c. Line x is the information, and line c is the control. The equations are

$$\lambda(s, x, c) = \delta(s, x, c) = c \wedge x \vee \neg c \wedge s.$$

Thus, as long as $c = 0$ the state s does not change. If $c = 1$ then s is changed to the input x. The figure shows the circuit and the state transition diagram. ♢

Cellular automata. All infinite machine models we considered so far were *serial*: a finite-state machine interacted with a large passive memory. Parallel machines are constructed from a large number of interacting finite-state machines. A clocked circuit is a parallel machine, but a finite one. One can consider infinite clocked circuits, but we generally require the structure of the circuit to be *simple*. Otherwise, an arbitrary noncomputable function could be encoded into the connection pattern of an infinite circuit.

One simple kind of infinite parallel machine is a one-dimensional array of finite automata A_n, $(-\infty < n < \infty)$ each with the same transition function. Automaton A_n takes its inputs from the outputs of the neighbors A_{n-1} and A_{n+1}. Such a machine is called a **cellular automaton**. In the simplest case, when inputs and outputs are the same as the internal state, such a cellular automaton can be characterized by a single transition function $\alpha(x, y, z)$ showing how the next state of A_n depends on the present states of A_{n-1}, A_n and A_{n+1}.

A Turing machine can be easily simulated by a one-dimensional cellular automaton. But tasks requiring a lot of computation can be performed faster on a cellular automaton than on a Turing machine, since the former can use each cell for computation.

A cellular automaton can easily be simulated by a 2-tape Turing machine.

(2.4.6) **Remark** It is interesting to note that in some sense, it is easier to define universal cellular automata than universal Turing machines. A 2-dimensional cellular automaton with state set S is characterized by a transition function $T : S^5 \to S$, showing the next state of the cell given the present states of the cell itself and its four neighbors (north, south, east, west). A universal 2-dimensional cellular automaton U has the following properties. For every other cellular automaton T we can subdivide the plane of U into an array of rectangular blocks B_{ij} whose size depends on T. With appropriate starting conditions, for some constant k, if we run U for k steps then each block B_{ij} performs a step of the simulation of one cell c_{ij} of T.

To construct a universal cellular automaton we construct first a logic circuit computing the transition function T. Then, we find a fixed cellular automaton U that can simulate the computation of an arbitrary logic circuit. Let us define cell states in which the cell behaves either as a horizontal wire, or a vertical wire, or a wire intersection or a turning wire (four different corners are possible), or a NOR gate, or a delay element (to compensate for different wire lengths). For an arbitrary logic circuit C, it is easy now to lay out a structure from such cells in the plane simulating it on the machine U. To simulate T, we place a copy of the structure simulating the circuit computing the transition function T into each block. The blocks are connected, of course, by cells representing wires. $\diamondsuit$

(2.4.7) **Example** A popular cellular automaton in two dimensions is John Conway's Game of Life. In this, each cell has two states, 0 and 1 and looks at itself and its 8 nearest neighbors. The new state is 1 if and only if either the old state is 1 and the state of 2 or 3 neighbors is 1, or the old state is 0 and the state of 3 neighbors is 1.

Using the above ideas and a lot of special properties of the Game of Life, it is possible to prove that the Game of Life is also a universal cellular automaton. $\diamondsuit$

There are many other interesting simple universal cellular automata.

2.28 **Exercise**

(a) Simulate a Turing machine by a one-dimensional cellular automaton.

*(b) Simulate a Turing machine by a one-dimensional cellular automaton in such a way that the state of the finite control in step t of the Turing machine can be determined by looking at cell 0 in step t of the cellular automaton.

$\diamondsuit$

2.5 Church Thesis, and Polynomial Church Thesis

We have had enough examples of imaginable computers to convince ourselves that all functions computable in some intuitive sense are computable on the Turing machine. This is CHURCH'S Thesis, whose main consequence (if we accept it) is that one can simply speak of computable functions without referring to the machine on which they are computable. Church stated his thesis around 1931. Later, it became apparent that not only can all imaginable machine models simulate each other but the "reasonable" ones can simulate each other in *polynomial time*. This is the Polynomial Church's Thesis (so called, probably, by LEVIN); here is a more detailed explanation of its meaning. We say that machine B simulates machine A in polynomial time if there is a constant k such that for any integer $t > 1$ and any input, t steps of the computation of machine A will be simulated by $< t^k$ steps of machine B. By "reasonable", we mean the following two requirements:

— Not unnaturally restricted in its operation.

— Not too far from physical realizability.

The first requirement excludes models like a machine with two counters; though some such machines may be able to compute all computable functions, but often only very slowly. Such machines are of undeniable theoretical interest. Indeed, when we want to reduce an undecidability result concerning computers to an undecidability result concerning e.g. sets of integer equations, it is to our advantage to use as simple a model of a computer as possible. But when our concern is the complexity of computations, such models have not much interest, and can be excluded.

All machine models we considered so far are rather reasonable, and all simulations considered so far were done in polynomial time.

A machine that does not satisfy the second requirement is a cellular automaton where the cells are arranged on an infinite binary tree. Such a machine could mobilize, in just n steps, the computing power of 2^n processors to the solution of some problems. But for large n, so many processors would simply not fit into our physical space.

The main consequence of the Polynomial Church's Thesis is that one can simply speak of functions computable in polynomial time (these are sometimes called "feasible") without referring to the machine on which they are computable, as long as the machine is "reasonable".

2.6 A realistic finite computer

Clocked circuits contain all essential elements needed to describe the work of today's computers symbolically.

In the fip-flop example, we have seen how to build memory elements. Combining the memory elements with a selection mechanism using a decoder, we can build a **random-access memory**, storing e.g. 2^{16} (64K) words with 16 bits each.

We can consider as the program in the broader sense, everything found in the memory at the beginning of the computation. The result of the computation can now be represented as $f_p(x)$, where p is the program and x is the input. Different choices of p define different functions f_p. Of course, a finite machine will be able to compute only a finite number of functions, and even these only on inputs x of a limited size.

There is a format for programs and programmable machines that has proved useful. There are some *operations*, *i.e.* small functions, used especially often. (E.g. multiplication, comparison, memory fetch and store, etc.) We build small circuits for all of these operations and arrange the circuits in an *operation table*, accessed by a decoder, just as a random- access memory. Most operations used in actual computers can be easily simulated using just 3-4 basic ones.

An **instruction** consists of the **address** of an operation in the operation table, and up to three **parameters**: arguments of the operation. A **program** is a series of instructions. It is stored in the random access memory.

A special circuit, the **instruction-interpreter**, stores a number called the **instruction pointer** that contains the address of the instruction currently executed. The interpreter goes through the following cycle:

1. It fetches the current instruction using the instruction-pointer.

2. It directs control to the circuit of the operation whose address is the first word of the instruction.

3. Upon return of the control, it adds 1 to the value of the instruction pointer.

Each operation returns control to the instruction-interpreter, but before that, it may change the value of the instruction-pointer ("goto instructions").

Just as in the RAM, it is enough to use two basic kinds of operation: **arithmetical assignment** and **flow control** (the latter involves a change in the instruction pointer). We can assign some special locations for input, for output, and for the instruction pointer. If the memory location involved in the assignment is input or output then the assignment involves an input- or output operation.

3 Algorithmic decidability

Until the 30-ies of our century, it was the—mostly, not precisely spelled out—consensus among mathematicians that every mathematical question that we can formulate precisely, can also be decided. This statement has two interpretations. We can talk about one yes-no question, and then the decision means that it can be proven or disproven from the axioms of set theory (or some other theory). Gödel published his famous result in 1931 according to which this is not true, moreover, no matter how we would extend the axiom system of set theory (subject to some reasonable restrictions, e.g. that no contradiction should be derivable and that it should be possible to decide about a statement whether it is an axiom), still there would remain unsolvable problems.

The second form of the question of undecidability is when we are concerned with a family of problems and are looking for an algorithm that decides each of them. Church in 1936 formulated a family of problems for which he could prove that it is not decidable by any algorithm. For this statement to make sense the mathematical notion of algorithm had to be created. Church used the combinatorial tools of logic for this. Of course, it seems plausible that somebody could extend the arsenal of algorithms with new tools, making it applicable to the decision of new problems. But Church also formulated the so-called *Church thesis* according to which every "calculation" can be formalized in the system he gave. We will see that Church's and Gödel's undecidability results are very closely related.

The same year, Turing created the notion of a Turing machine; We call something algorithmically computable if it can be computed on some Turing machine. We have seen in the previous section that this definition would not change if we started from the Random Access Machine instead of the Turing machine. It turned out that Church's original model and many other models of computation proposed are equivalent in this sense to the Turing machine. Nobody found a machine model (at least a deterministic one, not using any randomness) that could solve more computational problems. All this supports Church's thesis.

3.1 Recursive and recursively enumerable languages

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

We call a function $f : \Sigma_0^* \to \Sigma_0^*$ *recursive* or *computable* if there exists a Turing machine that for any input $x \in \Sigma_0^*$ will stop after finite time with $f(x)$ written on its first tape. The notions of recursive, as well as that of "recursively enumerable" and "partial recursive" defined below can be easily extended, in a unique way, to functions and sets over some countable sets different from Σ_0^*, like the set of natural numbers, the set N^* of finite strings of natural numbers, etc. The extension goes with the help of some standard coding of e.g. the set of natural numbers by elements of Σ_0^*. Therefore even though we will define these notions only over Σ_0^* we will refer to it as defined over many other domains.

(3.1.1) **Remark** We have seen in the previous section that we can assume without loss of power that the Turing machine has only one tape. $\Diamond$

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. 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.

(3.1.2) **Remark** It is obvious that there is a continuum (*i.e.* , uncountably many) of languages but only countably may 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 $\mathcal{L}$ occur not necessarily in increasing order or without repetitions.

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

(3.1.3) **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 Σ_0^*, for i steps. If the simulated T stops then T_0 ouputs x. Since every word of Σ_0^* will occur for infinitely many values of i the range of T_0 will be $\mathcal{L}$. $\blacksquare$

The technique used in this proof, that of simulating infinitely many computations by a single one, is sometimes called "dovetailing".

Now we study the relation of recursive and recursively enumerable languages.

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

Proof 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. $\blacksquare$

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

(3.1.5) **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. ∎

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

3.2 **Exercise**

(a) Prove that a language is recursively enumerable if and only if it can be enumerated without repetitions 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.

♢

Now, we will show that there are languages that are recursively enumerable but not recursive.

Let T be a Turing machine with k tapes. 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.

(3.1.6) **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 3.1.4. We prove the second statement, for simplicity, for $k = 1$.

Let us assume indirectly, 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 *diagonal* technique originating from set theory (where it was used by Cantor to show that the set of all functions of natural numbers is not countable). The technique forms the basis of many proofs in logic, set-theory and complexity theory. We will see some more of these in what follows.

There is a number of variants of the previous result, asserting the undecidability of similar problems. Instead of saying that language $\mathcal{L}$ is not recursive, we will say more graphically that the property defining $\mathcal{L}$ is undecidable.

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. When we speak about the halting problem in general, it is understood that a pair (T, x) is given where T is a Turing machine (given by its transition table) and x is an input.

(3.1.7) **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 2.1.2 (with $k = 2$), with the difference that at 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 about the latter, it is undecidable whether it halts for a given 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. It proceeds 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 only shows that if P_2 is decidable then P_1 is also. But since we know that P_1 is undecidable we learn that P_2 is also undecidable. The reduction of a problem to some seemingly unrelated problem is, of course, often very tricky.

It is worth mentioning a few consequences of the above theorem. Let us call a **description** of a Turing machine the listing of the sets Σ, Γ (where, as until now, the elements of Γ are coded by words over the set Σ_0) and the table of the functions α, β, γ.

(3.1.8) **Corollary** *It is algorithmically undecidable whether a Turing machine (given by its description) halts on 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 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. ∎

(3.1.9) **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 doing 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 Σ_0^* has not, or vice versa. Even a "more negative" results is true than this. We call a property of a language **trivial** if either all languages have it or none.

(3.1.10) **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 table) 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. ∎

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

3.3 **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. ♢

3.4 **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. ♢

3.5 **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. ♢

3.6 **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. ♢

3.7 **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? ♢

3.2 Other undecidable problems

The first algorithmically undecidable problems that we formulated (e.g. the halting problem) seem a little artificial and the proof uses the fact that we want to decide something about Turing machines, with the help of Turing machines. The problems of logic that turned out to be algorithmically undecidable (see below) may also seem simply too ambitious. We might think that mathematical problems occurring in "real life" are decidable. This, is, however, not so! A number of well-known problems of mathematics turned out to be undecidable; many of these have no logical character at all.

First we mention a problem of "geometrical character". 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].

(3.2.1) **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 3.1.5, 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:

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

Taken together with Theorem 3.2.1, we see that $\mathcal{L}_{\mathrm{TLNG}}$ can not even be recursively enumerable.

In the proof of Theorem 3.2.2, the following lemma will play important role.

(3.2.3) **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 3×3 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. ∎

3.8 **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. $\diamondsuit$

3.9 **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. $\diamondsuit$

Proof of Theorem 3.2.2 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 1×1, 2×2, 3×3, 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 3.2.3, for a large enough k already the square $k \times k$ is not tileable either, and therefore the Turing machine stops after

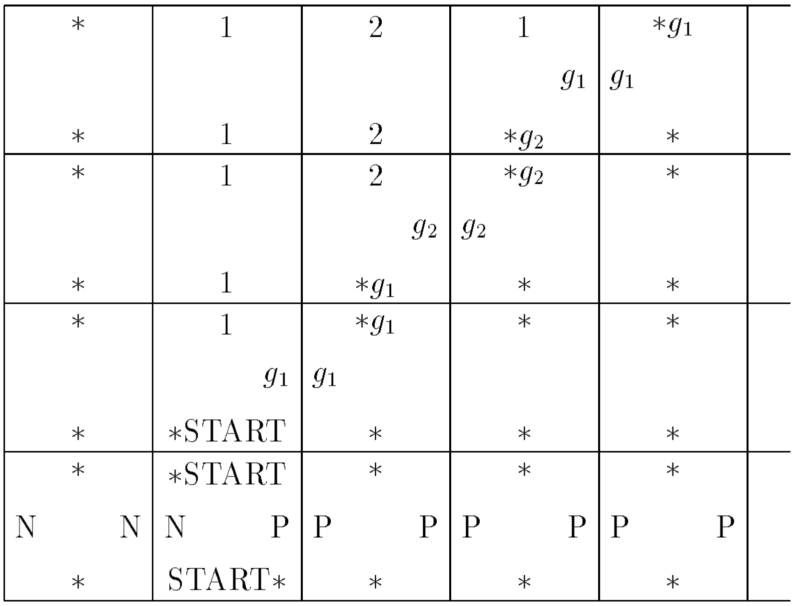


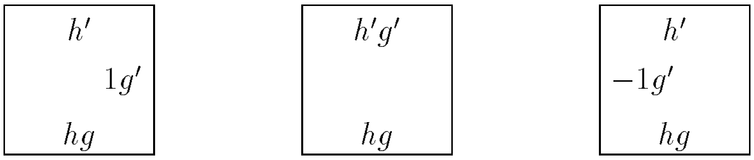
Figure 3.1: Tiling resulting from a particular computation

finitely many steps. Thus, according to Lemma 3.1.3, the language $\mathcal{L}_{\mathrm{NTLNG}}$ is recursively enumerable. ∎

Proof of Theorem 3.2.1 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 3.1.8, so it is also undecidable whether the constructed kit can tile the plane.

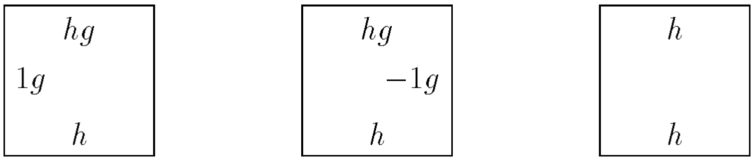
In defining the kit, we will write symbols, rather than numbers on the sides of the tiles; these are easily changed to numbers. For simplicity, assume $k = 1$. It is also convenient to assume (and achievable by a trivial modification of T) that the machine T is in the starting state only before the first 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 3.1): 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 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 3.1 shows the construction for the simple Turing machine



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

Figure 3.2: Tiles for currently scanned cell



a) b) c)

Figure 3.3: Tiles for previously scanned or unscanned cell

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 3.2. 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 3.3a-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 3.3c. Finally, the squares of the bottom line are shown in Figure 3.4. 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 3.2–3.4 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 3.4. 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 3.4 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 3.4 occurs. We must show that it can occur indeed only if the head moves onto the corresponging tape square. The symbols ± 1 in front of g exclude all other possibilities.

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

(3.2.4) **Remark** The tiling problem is undecidable even if we do not distinguish an initial domino. But the proof of this is much harder. ♢

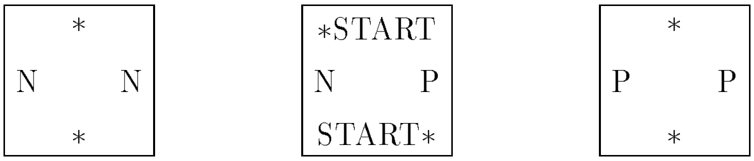


Figure 3.4: Tiles for the bottom line

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

3.11 **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. $\diamondsuit$

*(3.12) **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 3.11. Use the kit assigned to it by the proof of Theorem 3.2.1. 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 Σ_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'.] $\diamondsuit$

3.13 **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. $\diamondsuit$

3.14 **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. $\diamondsuit$

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:

(3.2.5) **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. $\diamondsuit$

(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.

(3.2.6) **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 β. Is β 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:

(3.2.7) **Contractability of polyhedrons** Can a given polyhedron be contracted into 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 $0 \le t \le 1$. 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.

3.3 Computability in logic

3.3.1 Godel's incompleteness theorem

Mathematicians have long held the conviction that a mathematical proof, when written out in all detail, can be checked unambiguously. ARISTOTLE made an attempt to formalize the rules of deduction but the correct formalism was found only by FREGE AND RUSSELL at the end of the 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 φ, an other sentence ψ called its **negation**.

(3.3.1) **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**.

(3.3.2) **Example** Here is a simple theory T_1 based on the language L_1 of the above Example 3.3.1. 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 3.3.2 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.

(3.3.3) **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 φ_n expressing the statement "$n \in L$"; moreover, this statement is a theorem in $\mathcal{T}$ if and only if it is true.

It is reasonable to expect that a theory of natural numbers with a goal of completeness be minimally adequate, *i.e.* that it should provide proofs for at least those facts that are verifiable anyway directly by computation, as "$n \in L$" indeed is. (In the next 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:

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

Proof If the theory were complete then, according to Theorem 3.3.3 it would give a procedure to decide all sentences of the form $n \in L$, which is impossible. ∎

(3.3.5) **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. $\diamondsuit$

(3.3.6) **Remark** Historically, Gödel's theorems preceded the notion of computability by 3-4 years. $\diamondsuit$

3.3.2 First-order logic

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

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

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

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

— Logical connectives: $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \ldots$.

— Quantifiers: $\forall, \exists$.

— Punctuation: (,).

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

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

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

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

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

There are natural ways to give *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.

(3.3.7) **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 3.3.7, 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.

(3.3.8) **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 3.3.3 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 $F_1, \dots, F_n$ 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 $F(X_1, \dots, X_n)$ be a Boolean formula of the variables $X_1, \dots, X_n$, with the property that it gives the value 1 for all possible substitutions of 0 or 1 into $X_1, \dots, X_n$. Let $\varphi_1, \dots, \varphi_n$ be arbitrary formulas. Formulas of the kind $F(\varphi_1, \dots, \varphi_n)$ are called **tautologies**.

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

Tautologies: All tautologies are axioms.

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

$$(t_1 = u_1 \wedge \dots \wedge t_n = u_n) \quad \Rightarrow \quad f(t_1, \dots, t_n) = f(u_1, \dots, u_n),$$
$$(t_1 = u_1 \wedge \dots \wedge t_n = u_n) \quad \Rightarrow \quad (P(t_1, \dots, t_n) \Leftrightarrow P(u_1, \dots, u_n))$$

are axioms.

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

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

The system has two rules:

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

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

(3.3.9) **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.

(3.3.10) **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).

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

(3.3.11) **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.

(3.3.12) **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 $\mathcal{P}$. ∎

3.15 **Exercise** Our proof of Gödel's theorem does not seem to give a specific sentence φ_T undecidable for a given minimally adequate theory T. Show that such a sentence can be constructed, if the language L used in the definition of "minimally adequate" is obtained by any standard coding from the nonrecursive r.e. set of Theorem 3.1.6. $\diamondsuit$

4 Storage and time

The algorithmic solvability of some problems can be very far from their *practical* solvability. We will see that there are algorithmically solvable problems that cannot be solved, for an input of size n, in fewer than 2^{2^n} steps. 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 **storage**.

Let us fix some finite alphabet Σ inluding the symbol $*$ with $\Sigma_0 = \Sigma \setminus \{*\}$. 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 over Σ_0 written over the input tape.

The **time demand** of a Turing machine T is a function $time_T(n)$ that is the maximum of the number of steps taken by T over all possible inputs of length n. We assume $time_T(n) \geq n$ (the machine must read the input; this is not necessary so but we exclude only trivial cases with this assumption). The function $space_T(n)$ is defined as the maximal number, over all inputs of length n, of all cells on all tapes into which the machine writes. (This way, the cells occupied by the input are not included into the space requirement.) Obviously, $space_T(n) \geq 1$.

Turing machine T is called **polynomial** if there is a polynomial $f(n)$ such that $time_T(n) = O(f(n))$, *i.e.* , there is a constant c such that the time demand of T is $O(n^c)$. We can define similarly the exponential algorithms (for which the time demand is $O(2^{n^c})$ for some $c > 0$), the algorithms (Turing machines) with polynomial space demand, etc.

We say that language $\mathcal{L} \in \Sigma^*$ has **time complexity** at most $f(n)$, if it can be decided by a Turing machine of time demand at most $f(n)$. We denote by 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 the **space complexity** of a language, the language classes DSPACE$(f(n))$ and the language class PSPACE.

The time-and space demand defined with the help of Turing machine is suitable for theoretical study; in practice, using the RAM is more convenient (approximates reality better). It follows, however, from Theorems 2.2.1 and 2.2.4 that from the point of view of the most important polynomial classes (polynomial, exponential time and space, etc.) it does not matter, which one is used in the definition.

(4.0.1) **Remark** When we find out that 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 DTIME$(f(n))$, 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. ♢

Church Thesis, and Polynomial Church Thesis We have had enough examples of imaginable computers to convince ourselves that all functions computable in some intuitive sense are computable on the Turing machine. This is CHURCH'S Thesis, whose main consequence (if we accept it) is that one can simply speak of computable functions without referring to the machine on which they are computable. Church stated his thesis around 1931. Later, it became apparent that not only can all imaginable machine models simulate each other but the "reasonable" ones can simulate each other in *polynomial time*. This is the Polynomial Church's Thesis (so called, probably, by LEVIN); here is a more detailed explanation of its meaning. We say that machine B simulates machine A in polynomial time if there is a constant k such that for any integer $t > 1$ and any input, t steps of the computation of machine A will be simulated by $< t^k$ steps of machine B. By "reasonable", we mean the following two requirements:

— Not unnaturally restricted in its operation.

— Not too far from physical realizability.

The first requirement excludes models like a machine with two counters; though some such machines may be able to compute all computable functions, but often only very slowly. Such machines are of undeniable theoretical interest. Indeed, when we want to reduce an undecidability result concerning computers to an undecidability result concerning e.g. sets of integer equations, it is to our advantage to use as simple a model of a computer as possible. But when our concern is the complexity of computations, such models have not much interest, and can be excluded.

All machine models we considered so far are rather reasonable, and all simulations considered so far were done in polynomial time.

A machine that does not satisfy the second requirement is a cellular automaton where the cells are arranged on an infinite binary tree. Such a machine (or other, similar machines called PRAM and considered later in these notes) could mobilize, in just n steps, the computing power of 2^n processors to the solution of some problems. But for large n, so many processors would simply not fit into our physical space.

The main consequence of the Polynomial Church's Thesis is that one can simply speak of functions computable in polynomial time (these are sometimes called "feasible") without referring to the machine on which they are computable, as long as the machine is "reasonable".

4.1 Polynomial time

Many algorithms important in practice run in polynomial time (in short, are polynomial). Polynomial algorithms are often very interesting mathematically since they require deeper insights into the problems and stronger tools. In this book, it is not our goal to overview these: we will treat only a few simple but important algorithms, partly to illustrate the notion, partly to introduce some important basic ideas. It is notable that according to

Theorems 2.2.1 and 2.2.4, the notion of a polynomial algorithm does not change if we use the RAM instead of Turing machines.

4.1.1 Combinatorial algorithms

The most important algorithms of graph theory are polynomial.

— To decide whether a given graph is *connected*.

— To find the *shortest path* between two points in a graph;

— To find a *maximum flow* between two points in a graph whose edges have capacities: e.g., Dinic-Karzanov's and Edmonds-Karp's algorithms.

— To find a maximum matching in a bipartite graph: e.g. the so-called Hungarian method.

— To find a maximum matching in an arbitrary graph: the Edmonds algorithm.

Some of these algorithms are rather difficult and belong to other subjects: graph theory, operations research. This course will not deal with them in detail.

4.1.2 Arithmetical algorithms

The basic arithmetical operations are polynomial: addition, subtraction, multiplication and division with remainder of integers. (Remember that the length of an integer n as input is the number of its binary digits, *i.e.* , $\log_2 n + O(1)$). We learn polynomial algorithms for all these operations in elementary school (linear algorithms in case of addition and subtraction, quadratic algorithms in case of multiplication and division). We also count the comparison of two numbers by size as a trivial but basic arithmetical operation.

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 with 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.

We must mention the **Euclidean algorithm**, computing the greatest common divisor of two numbers, among the fundamental arithmetical algorithms with polynomial time.

Euclidean algorithm Given two natural numbers, a and b. Let us pick one that is not bigger than the other, let this be, e.g., a. If $a = 0$ then the greatest common divisor of a and b is $\gcd(a, b) = b$. If $a \neq 0$ 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.

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 how to translate such an informal program into a formal one, and a recursive program into a machine-language program (most compilers can do that).

(4.1.1) **Lemma** *The Euclidean algorithm takes polynomial time. More precisely, it consists of $O(\log a + \log b)$ arithmetical operations carried out on natural numbers not larger than a, b.*

Proof Since $0 \le r < b$, the Euclidean algorithm will terminate sooner or later. Let us see that it terminates in polynomial time. Notice for this that $b > a + r > 2r$ 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 can be obviously carried out in polynomial time. ∎

It is interesting to note that 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 $p'a + q'b$.

4.1 **Exercise** The Fibonacci numbers are defined by the following recursion: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ for $k > 1$. Let $1 \le a \le b$ 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})? ◇

(4.1.2) **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 $0 < a \le b$ then let $b := b - a$. Though 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 remainder classes modulo m, in polynomial time. (This is different from division with remainder!) More generally, in the ring of remainder classes modulo m, we can divide by a number relatively prime to m, in polynomial time. Let $0 \le a, b < m$ with $\gcd(m, b) = 1$. Carrying out the "division" $a : b$ means that we are looking for an integer x with $0 \le x < m$ such that

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

Applying the Euclidean algorithm to compute the greatest common divisor of the numbers b, m, we obtain integers p and q such that $bp + mq = 1$. Thus, $bp \equiv 1 \pmod{m}$, *i.e.*

$b(ap) \equiv a \pmod{m}$. Thus, 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 $x_1, \dots, x_k$ with respect to the moduli $m_1, \dots, m_k$ which are all relatively prime with respect to each other. The Chinese Remainder Theorem says that these remainders uniquely determine the remainder of x modulo the product $m_1 \cdots m_k$. But how can we compute this?

It is enough to deal with the case $k = 2$ since for general k, the algorithm follows from this by mathematical induction. There are integers q_1 and q_2 such that $x = x_1 + q_1 m_1$ and $x = x_2 + q_2 m_2$. We are looking for such integers. Thus, $x_2 - x_1 = q_1 m_1 - q_2 m_2$. This equation does not determine uniquely, of course, the numbers q_1 and q_2 but this is not important. It is sufficient to find, using the Euclidean algorithm, numbers q_1' and q_2' with the property

$$x_2 - x_1 = q_1' m_1 - q_2' m_2.$$

Indeed, let $x' = x_1 + q_1' m_1 = x_2 + q_2' m_2$. Then $x' \equiv x \pmod{m_1}$ and $x' \equiv x \pmod{m_2}$ and therefore $x' \equiv x \pmod{m_1 m_2}$.

It is also worth-while to consider the operation of exponentiation. Since even to write up 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, it 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 remainder class modulo m, hence it can be represented with $\log m + O(1)$ symbols. We will show that it can be not only represented polynomially but also computed in polynomial time.

(4.1.3) **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 up b in the binary representation:

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

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 remainder classes a^{2^t} for $0 \le t \le \log b$ are easily obtained by repeated squaring, and then we multiply the k needed numbers among them. Of course, we carry out all operations modulo m, *i.e.* after every multiplication, we also perform a division with remainder by m. ∎

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

4.1.3 Algorithms of linear algebra

The basic operations of linear algebra are polynomial: addition and scalar product of vectors, multiplication and inversion of matrices, the computation of determinants. However, these facts are not 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.

At this point, if you don't remember the definition and basic facts about determinants then it is necessary to refresh them from a textbook of linear algebra. You need both the the definition as the sum of $n!$ products, and the fact that it can be computed using an algorithm transforming it into the determinant of a triangle matrix.

Let us understand, first of all, that the polynomial computation of $\det(A)$ is not inherently impossible, *i.e.* the result can be written up with polynomially many digits. Let $K = \max|a_{ij}|$, then to write up A we need obviously at least $n^2 + \log K$ bits. On the other hand, the definition of determinants gives

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

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

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

bits. Thus, $\det(A)$ can be written up with polynomially many bits. 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 with polynomially many digits.

4.2 **Exercise** Show that if A is a square matrix consisting of integers then to write up $\det(A)$ 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).] $\Diamond$

The usual procedure to compute the determinant is the so-called 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 but in the triangular matrix, the computation of the determinant is more convenient: we must only multiply the diagonal elements to obtain it. (It is also possible to obtain the inverse matrix from this form; we will not deal with this 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 positions hold a nonzero element. Pick a nonzero element from the last $n - t$ columns (if there is no such element we stop). We call this element the *pivot element* of this stage. Let us 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 $(t+1, t+2), \ldots, (t+1, n)$. It is known that the subtractions do not change value of the determinant and the rearrangement (involving as many exchanges of rows as of columns) also does not change the determinant.

Since one iteration of the Gaussian elimination uses $O(n^2)$ arithmetical operations and n iterations must be performed this means $O(n^3)$ arithmetical 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 case of the rational field. We assumed that the elements of the original matrix are integers; but during the running 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).

But 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 this but then we have to simplify each matrix element after each iteration, for which we would have to perform the Euclidean algorithm. Though this can be performed in polynomial time but it is a lot of extra work, desirable to avoid. (Of course, we also have to show that in the simplified form, the occurring numerators and denominators have only polynomially many digits.)

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 be very large (have a nonpolynomial number of digits)!

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

(4.1.5) **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 $d_1,\dots,d_k,a_{ij}^{(k)}$. Thus

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

Similarly,

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

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

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

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

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

and hence

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

This formula can be considered a recursion for computing the numbers $d_{ij}^{(k)}$. Since the left-hand side is 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.

(4.1.6) **Remark** It is worth mentioning two further possibilities for the remedy of the problem of the fractions occurring in Gaussian elimination. We can approximate the number by binary "decimals" of limited accuracy (as it seems natural from the point of view of computer implementation), allowing, say, p binary digits after the binary "decimal point". Then the result is only an approximation, but since the determinant is an integer, it would be enough to compute it with an error smaller than $1/2$. Using the methods of numerical analysis, it can be found out how large must p be chosen to make the error in the end result smaller than $1/2$. It turns out that a polynomial number of digits is 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'n have to use fractions. Since we know that $|\det(A)| < n!K^n$ it would be enough to choose for m a prime number greater than $n!K^n$. This is, however, not easy (see the section on randomized algorithms), hence we can 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 number of all digits 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, and 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 the sieve method and still keep the algorithm polynomial. 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.)

The modular method is successfully applicable in a number of other cases. We can consider it as a coding 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 coding, the arithmetic operations can be performed very simply, and even in parallel for the different primes. Comparison by magnitude is, however, awkward. $\diamondsuit$

(4.1.7) **Remark** The distinguished role of polynomial algorithms is underscored by the fact that some natural syntactic conditions can be imposed on an algorithm that are equivalent to requiring it to run in polynomial time. In the programming language Pascal, e.g., those programs are polynomial that don't have any statements with "**goto**", "**repeat**" or "**while**" and the upper bound of the "**for**" instructions is the size of the input (the converse is also true: all polynomial algorithms can be programmed this way). $\diamondsuit$

4.2 Other typical complexity classes

4.2.1 Linear time

Many basic arithmetical algorithms (the addition and comparison of two numbers) have linear time.

Linear-time algorithms are important mainly where relatively simple tasks must be performed on inputs of large size. A number of data processing algorithms have linear time. An important linear-time graph algorithm is depth-first search. With its help, several non-trivial graph-theoretical problems (e.g. drawing the graph in a plane) are solvable in linear time.

An algorithm is said to have **quasi-linear time** if its time requirement is $O(n(\log n)^c)$ here c is a constant. The most important problem solvable in quasi-linear time is sorting for which several $O(n \log n)$ algorithms are known. Important quasi-linear algorithms can be found in the area of image processing (the convex hull of an n-element plane point set can be found, e.g., also in $O(n \log n)$ steps.)

4.2.2 Exponential time

Looking over "all cases" often leads to exponential-time algorithms (*i.e.* , algorithms whose time falls between 2^{n^a} and 2^{n^b} where $a, b > 0$ are constants). If, e.g., we want to determine whether the vertices of a graph G are colorable with 3 colors (in such a way that the colors of neighboring nodes are different) then the trivial algorithm is to overview all possible colorings. This means the overview of 3^n cases where n is the number of points in the graph; one case needs time $O(n^2)$ in itself. (For this problem, unfortunately, a better—not exponential-time—algorithm is not known and, in some sense, cannot even be expected, as the section on NP-complete problems shows.)

A typical example of exponential-time algorithm is when, in a two-person board game, we determine the optimal step by surveying all possible continuations. We assume that every given situation of a game can be described by a word x of some finite alphabet Σ (typically, telling the position of the pieces on the board, the name of the player whose turn it is and possibly some more information, e.g. in the case of chess whether the king has moved already, etc.). An initial configuration is distinguished. We assume that the two players take turns and have a way to tell, for two given configurations, whether it is legal to move from one into the other, by an algorithm taking, say, polynomial time[1]. We will assume that for each game, if a configuration follows legally another one then they have the same length n. If there is no legal move in a configuration then it is a terminal configuration and a certain

[1]It would be sufficient to assume polynomial storage but it is a rather boring game in which to decide whether a move is legal takes more than polynomial time

algorithm decides in polynomial time, who won. If the game went on for more than $|\Sigma|^n$ steps then some configuration must have been repeated—in this case, we call it a draw.

A nonterminal configuration is **winning** for the player whose turn it is if there is a strategy that, starting from this configuration, leads to the victory of this player whatever the other player does.

Let us give three different exponential algorithms to decide this game. The first algorithm surveys all games; the second one is a recursive description of the first one; the third algorithm catalogues all winning and losing configurations.

1. Assume that we want to decide about position x_0 whether it is a winning or a losing one (from the point of view of the player whose turn it is). A sequence $x_0, x_1, \ldots, x_k$ of positions is a **subgame** if a legal step leads from each position x_i into x_{i+1}. In a given moment, the machine analyzes all possible continuations of some subgame. It will always hold that among all continuations of $x_0, x_1, \ldots, x_i$ $(0 \le i \le k)$ the ones smaller than x_{i+1} (with respect to the lexicographical ordering of the words of length n) are "bad steps", i.e. the player whose turn it is after $x_0, x_1, \ldots, x_i$ loses if it moves there (or the step is illegal).

The algorithm surveys all words of length n in lexicographical order whether they are legal continuations of x_k. If it finds one this is x_{k+1} and we go on to examine the one longer subgame obtained thus. If it does not find such then it marks certain positions of the subgame under consideration "winning" (for the player on turn) according to the following: it decides who wins in x_k. If the winner is the player whose turn it is then x_k is a winning position; if it is the other player then x_{k-1} is a winning position. Let i be the smallest index for which we already know that it is a winning position. If $i = 0$ then we know that the starting player wins. If $i > 1$ then it was a bad step to move here from x_{i-1}. The algorithm checks therefore whether it is possible to step from x_{i-1} legally into a position lexicographically greater than x_i. If yes then let y be the first such; the algorithm continues with checking the subgame $x_0, x_1, \ldots, x_{i-1}, y$. If no position lexicographically greater than x_i is a legal step from x_{i-1} then every step from x_{i-1} is "bad". Thus, if $i = 1$ then the starter loses in the beginning position, and we are done. If $i \ge 2$ then we can mark x_{i-2} as a winning configuration.

2. We define a recursive algorithm $W(x, t)$ that, given a configuration x and a step number t decides whether it is a winning, losing or draw configuration at time t for the player whose turn it is. By definition, $W(x, t)$ is draw if $t \ge |\Sigma|^n$. The algorithm enumerates, in a lexicographic order, all configurations y and checks if there is a legal move from x to y. If the answer is no then x is a terminal configuration and we apply the algorithm to decide who won. If there are legal moves into some configurations y then $W(y, t+1)$ is called recursively for each of them. If for some y, the answer is that it is losing then x is winning. If all legal y's are winning then x is losing. Otherwise, we have a draw.

3. Let us show an algorithm solving the same problem that catalogues all configurations into winning and losing ones. Let $W_i = W_i(x)$ be the set of configurations of length $n = |x|$ from which the player has a winning strategy in i or fewer moves. Then $W_0(x)$ is the set of terminal configurations in which the player whose move it is won. Here is a exponential-time algorithm to compute W_{i+1} from W_i. Let U_i be the set of configurations from which every

legal move of the player (if any) leads to W_i. Then W_{i+1} is the set of configurations that either belong to W_i or from which there is a move to U_i. Sooner or later, the sets W_i stop growing since they all contain strings of length n. When $W_{i+1} = W_i$ then $W = W_i$.

This algorithm shows that if there is a winning strategy from a configuration of length n then there is a strategy leading to victory in $< |\Sigma|^n$ moves even if we do not limit the length of the game.

4.2.3 Polynomial space

Obviously, all polynomial-time algorithms require polynomial space but polynomial space is significantly more general. The storage requirement of the trivial graph coloring algorithm treated in 4.2.2 is polynomial (moreover, 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.

The most typical example for a polynomial-storage algorithm is finding the optimal step in a game by searching through all possible continuations, where we assume now that the game is always finished in n^c steps.

The first exponential game-evaluation algorithm given above takes automatically only polynomial space if the length of the game is limited by a polynomial.

The second algorithm is a recursive one; when implementing the above described recursive game-evaluation method on a Turing machine, before descending to the next recursive call of W, an "activation record" of the current recursive call must be stored on tape. This record must store all information necessary for the continuation of W. It is easy to see that only the following pieces of information are needed: the depth of the recursion (the number of current step in the game), the argument x, the currently investigated next configuration y and three bits saying whether among the y's checked so far, any was winning, losing or draw. The maximum depth of recursion is only as large as the maximum length of the game, showing that the storage used is only $O(n)$ times this much. It follows that if a game is limited to a number of steps polynomial in the board size then it can be evaluated in polynomial space.

Polynomial storage (but exponential time) is required by the "trivial enumeration" in the case of most combinatorial enumeration problems. For example, if we want to determine, for a given graph G, the number of its colorings with three color then we can survey all colorings and whenever a coloring is legal, we add a 1 to the counter.

4.3 Linear space

From the point of view of storage requirement, this class is as basic as polynomial time. Those graph algorithms belong here that can be described by the changes of some labels assigned to points and edges: e.g. connectivity, the Hungarian method, the search for a shortest path, or optimal flow. Such are also the majority of bounded-precision numerical algorithms.

For an example, we describe **breadth-first search** on a Turing machine. The input of this algorithm is an (undirected) graph G and a vertex $v \in V(G)$. Its output is a spanning tree F of G such that for every point x, the path in F connecting x with v is the shortest one among all paths in G. We assume that the graph G is given by the lists of neighbors for every vertex. During the algorithm, every node can get a label. Originally, only v is labelled. While working, we keep all the labelled points on a tape called F, writing after each one, in parentheses, also the point from which we arrived at it. Some of the labelled points will have the property of having been "searched". We keep the names of the labelled but not searched points on a separate tape, called Queue. In one iteration, the machine reads the name of the first point from the queue (let this be u) and then it searches among its neighbors for one without a label. If one is found then its name will be written at the end of both the queue tape and the F tape, adding on the latter one, in parentheses, the name "u". If none is found then u will be erased from the beginning of the queue tape. The algorithm stops if the queue tape is empty. In this case, the pairs occurring on the F tape give the edges of the sought-after tree F.

The storage requirement of this algorithm is obviously only $O(n)$ numbers with $O(\log n)$ digits, and this much storage is needed already for writing up the names of the points.

4.4 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 $time_T(n) \le f(n)$ holds then there is also a Turing machine recognizing L for which this inequality holds for all n. For small n's, namely, 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 input cannot be "saved").

(4.4.1) **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 $time_S(n) \le c \cdot time_T(n) + n$.*

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. Besides these, let us also take $2p - 1$ "starting" tapes and $2p - 1$ 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 $t-p+1, \dots, t+p-1$ and the work-tape cells with indices $s-p+1, \dots, s+p-1$. 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 (0 or 1) and stops then let S do this, too. Thus, we constructed machine S that (apart from the initial copying) makes only a pth of the number of steps of T and decides obviously the same language. ∎

*(4.3) **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 $\mathit{time}_S(n) \le c \cdot \mathit{time}_T(n) + n$ (here, we allow the extension of the alphabet; see [4]). ♢

4.4 **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 $\mathcal{L} \in \mathrm{DTIME}(f(n))$ for a language then there is a constant k (depending on the language) that $\mathcal{L} \in \mathrm{DSPACE}(k \cdot f(n))$. (If extending the alphabet is allowed and $f(n) > n$ then $\mathrm{DSPACE}(k \cdot f(n)) = \mathrm{DSPACE}(f(n))$ and thus it follows that $\mathrm{DTIME}(f(n)) \subset \mathrm{DSPACE}(f(n))$.) On the other hand, the time demand is not greater than an exponential function of the space demand (since exactly the same memory configuration, taking into account also the positions of the heads and the state of the control unit, cannot occur more than once without getting into a cycle). Computing more precisely, the number of different memory configurations is at most $c \cdot f(n)^k m^{f(n)}$ 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 $\mathcal{L} \in \mathrm{DSPACE}(f(n))$ then $\mathcal{L} \in \mathrm{DTIME}((m+1)^{f(n)})$.

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

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

Proof The proof is similar to the proof of the fact that the halting problem is undecidable. We can assume $f(n) > n$. Let T be the 2-tape universal Turing machine constructed in the proof of Theorem 2.1.1, and let $\mathcal{L}$ consist of all words x for which it is true that having x as input on both of its tape, T halts in at most $f(|x|)^4$ steps. $\mathcal{L}$ is obviously recursive.

Let us now assume that $\mathcal{L} \in \mathrm{DTIME}(f(n))$. Then there is a Turing machine (with some $k > 0$ tapes) deciding $\mathcal{L}$ in time $f(n)$. From this, by Theorem 2.1.2, 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, according to Exercise 2.6, it halts in these cases within $|p|f(|x|)^3$ steps.

There are now 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 $\mathcal{L} \notin \mathrm{DTIME}(f(n))$. ∎

There is also a different way to look at the above result and related ones. For some fixed universal two-tape Turing machine U and an arbitrary function $t(n) > 0$, the **t-bounded halting problem** asks, for n and all inputs p, x of maximum length n, whether the above machine U halts in $t(n)$ steps. (Similar questions can be asked about storage.) This problem seems decidable in $t(n)$ steps, though this is true only with some qualification: for this, the function $t(n)$ must itself be computable in $t(n)$ steps (see the definition of "fully time-constructible" below). We can also expect a result similar to the undecidability of the halting problem, saying that the t-bounded halting problem cannot be decided in time "much less" than $t(n)$. How much less is "much less" here depends on some results on the complexity of simulation between Turing machines.

We call a function $f : \mathbf{Z}_+ \to \mathbf{Z}_+$ **fully time-constructible** if there is a multitape Turing machine that for each input of length n, uses exactly $f(n)$ time steps. The meaning of this strange definition is that with fully time-constructable functions, it is easy to bound the running time of Turing machines: If there is a Turing machine making exactly $f(n)$ steps on each input of length n then we can build this into any other Turing machine as a clock: their tapes, except the 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 $1 \dots 1$ of length n and we want as output a sequence $1 \dots 1$ of length $f(n)$. The results would not be changed, however, if n and $f(n)$ were represented e.g. in binary notation.) Now the following lemma is easy to prove:

(4.4.3) **Lemma**

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

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

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

This lemma allows us to use, in most cases, fully time-constructible and well-computable functions interchangeably. Following the custom, we will use the former. Further refinement of Theorem 4.4.2 (using Exercise 2.9) justifies the following:

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

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

4.5 **Exercise** Using Exercise 2.9, 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 $t'(n)$. $\diamondsuit$

4.6 **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.

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

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

Thus, there is a recursive function f with

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

moreover, there is even one with

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

Proof Let us fix a 2-tape universal Turing machine. Denote $\tau(x, y)$ the time needed for T 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 $|x|, |y| \le n$ then either $\tau(x, y) \le h(n)$ or $\tau(x, y) \ge (\varphi(h(n)))^3$.

If the function

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

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 $(x, y) \in (\Sigma_0^*)^2$, $|x|, |y| \le n$ 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 $t < s \le \varphi(t)^3$ 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) ($|y| \le n$) 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 problem, there is often no "best" algorithm; moreover, the following surprising theorem is true.

(4.4.6) **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 $g(time_S(n)) < time_T(n)$.*

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

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

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

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

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

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

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

This last step itself is already too much if $|y_i| \le k$; therefore we make a list of all pairs (x_i, y_i) with $|y_i| \le k$ (this is a finite list), and put this into the control unit. This begins therefore by checking whether the given word x is in this list as the first element of a pair, and if it is, it accepts x (beyond the reading of x, this is only one step!). Suppose that x_i is not in the list. Then y_i, if it exists, is longer than k. We can try all inputs (x, y) with $k < |y| \le |x|$ for "fastness" and this needs only $(m^2 n + 1)h(n-k-1)$ (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 Σ_0^* writing 1 or 0 on its first tape according to whether x is in the language $\mathcal{L}$) then y cannot be equal to any of the selected words y_i. This follows by the usual "diagonal" reasoning: if $y_i = y$ then let us see whether x_i is in the language $\mathcal{L}$. If yes then T_0 must give result "1" for the pair (x_i, y_i) (since $y = y_i$ decides $\mathcal{L}$). But then according to the definition of $\mathcal{L}$, we did not put x_i into it. Conversely, if $x_i \notin \mathcal{L}$ then it was left out since T_0 answers "1" on input (x_i, y_i); but then $x_i \in \mathcal{L}$ since the program $y = y_i$ decides $\mathcal{L}$. We get a contradiction in both cases.

Third, we convince ourselves that if program y decides $\mathcal{L}$ on the machine T_0 then (x, y) can be "fast" only for finitely many words x. Let namely (x, y) be "fast", where $x = x_i$. Since y was available at the selection of y_i (it was not selected earlier) therefore we would have had to choose some y_i for this i and the actually selected y_i could not be longer than y. Thus, if x differs from all words x_j with $|y_j| \le |y|$ 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 $time_{T_1}(n) \le (time_T(n))^2$. 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) $\tau(x, y) \le (time_T(|x|))^3$ for all sufficiently long words x. According to what was proved above, however, we have $\tau(x, y) \ge h(|x| - |y|)$ for all but finitely many x, and thus $time_T(n) \ge (h(n - |y|))^{1/3}$.

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

$$time_T(n) \ge (h(n - |y|))^{1/3} \ge g(h(n - |y| - 1)) \ge g(time_S(n)).$$

∎

The most important conclusion to be drawn from the speed-up theorem is that 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).

Everybody who is trying to solve an algorithmic problem efficiently is in the business of proving upper bounds. Giving lower bounds is the most characteristic (and hard) task of complexity theory but this task is often unseparable from questions of upper bounds.

4.5 EXPTIME-complete and PSPACE-complete games

The following theorem shows that the exponential-time halting problem can be reduced to the solution of a certain game. It is convenient to use cellular automata instead of Turing machines here. Let C be a one-dimensional cellular automaton with some set Γ of states one of which is called the **blank** state while another one is called the **halting** state. If a cell reaches the halting state it stays there. A finite **configuration** of C is a finite string in Γ^*, assuming blanks in the rest of the cells. A **halting configuration** is one in which one of the cells has the halting state.

Games were defined in 4.2.2.

(4.5.1) **Theorem** *There is a game G and a polynomial-time function $f(X)$ with the property that for all n, for each initial configuration X of length n the automaton C reaches a halting configuration in 2^n steps if and only if $f(X)$ is a winning configuration for G.*

Proof Let H be the halting state. Let $C(a, b, c)$ be the transition function of the cellular automaton C. Let $X = X_1 \cdots X_n$ be the initial configuration and let $y[t, i]$ be the state of cell i at time t under the computation starting from X. One of the players will be called the Prover. She wants to prove that $y[2^n, i] = H$ for some i. The other player is called the Verifier. Each move of Prover offers new bits of evidence and each move of Verifier pries into the evidence one step further. Formally, each board configuration of the game is a 7-tuple of the form

$$(X \mid t, i \mid a_{-1}, a_0, a_1 \mid b).$$

The board configuration reflects the claim $y[t, i] = b$ and $y[t-1, i+\varepsilon] = a_\varepsilon$ for $\varepsilon = -1, 0, 1$. The special symbol '?' is also permitted in place of a_{-1}, a_0, a_1 and b. Whenever none of them is '?' these states must satisfy the relation

$$b = C(a_{-1}, a_0, a_1).$$

Also, if $t = 0$ then we must have

$$a_{-1} = X_{i-1},\ a_0 = X_i,\ a_1 = X_{i+1}.$$

The starting configuration is $(X \mid 2^n, ? \mid ?, ?, ? \mid H)$. Whenever it is Prover's turn her task is to fill in all the question signs. Verifier then makes a new question by choosing ε in $\{-1, 0, 1\}$ and preparing the new configuration

$$(x \mid t-1, i+\varepsilon \mid ?, ?, ? \mid a_\varepsilon).$$

Thus, the new question relates to one of the ancestors of the space-time point (t, i). If e.g. $\varepsilon = 1$ then Verifier's next question asks for $y[t-2, i-2]$, $y[t-2, i-1]$, and $y[t-2, i]$. If Prover can always answer then she wins when $t = 1$.

We assert that

1. Prover has a winning strategy if the computation of C halts within 2^n steps.

2. Verifier has a winning strategy otherwise.

The proof of the first assertion is simple since Prover's strategy is to simply fill in the corresponding values of $y[t, i]$. To prove the second claim let us note that since C does not halt within 2^n steps Prover must lie when she fills in the question signs in the first step. Moreover, one of the three claims about the cell states at time $2^n - 1$ must be false. Verifier will follow up the false claim. In this way, Prover will be forced to make at least one false claim in each step. In the step with $t = 1$, this would lead to a false claim about some X_i, which would lead to an illegal board configuration—so, Prover loses. ∎

A similar statement holds for polynomial space.

(4.5.2) **Theorem** *Let T be a Turing machine using polynomial storage. There is a game G with a polynomial number of steps and a polynomial-time function $f(X)$ with the property that for all n, for each initial configuration X of length n the machine T halts if and only if $f(X)$ is a winning configuration for G.*

Proof Assume that for some $c > 0$, on inputs of length n, the machine T uses no more than $N = n^c$ cells of storage. Without loss of generality, let us agree that before halting, the machine T erases everything from the tape (just to make the halting configuration unique). We know that then, for some constant s, if T halts at all it halts in s^N steps. The players of the game will again be called Prover and Verifier. Each configuration of the game is a tuple

$$(x \mid t_1, y_1 \mid t_2, y_2)$$

with $t_1 \leq t_2$ where the entry y_1 can be the symbol '?'. Configurations x, y, z corresponds to configurations of the machine T. A game configuration represents Prover's claim that in a computation of T starting from x, at time t_1 we arrive at y_1 and at time t_2 we arrive at y_2. Let H be the halting configuration. Then the game starts with the position $(x \mid s^N, H \mid s^N, H)$, and it is Verifier's turn. If it is Prover's turn he has to fill in the question sign. If it is Verifier's turn and the configuration is $(x \mid t_1, y_1 \mid t_2, y_2)$ and both t_1 and $t_2 - t_1$ are at most 1 then she checks whether Prover's claim is correct. If yes Prover wins, else Verifier wins. Otherwise, Verifier decides whether she wants to follow up the first half of Prover's claim or the second half. If she decides for the first half the new configuration is

$$(x \mid \lfloor t_1/2 \rfloor, ? \mid t_1, y_1).$$

If she decides for the second half the new configuration is

$$(y_1 \mid \lfloor (t_1 + t_2)/2 \rfloor k, ? \mid t_2, y_2).$$

The rest of the proof is similar to the proof of the previous theorem. ∎

4.7 **Exercise** Construct a PSPACE-complete game in which both players can change only a single symbol on the board, and to determine what change is permissible one only has to look at the symbol and its two neighbors. ♢

4.6 Storage 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 proven 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 4.4.2.

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).

An interesting remark can be made on the relation between storage and time complexities of *parallel computations*: we return to this in the section on parallel computations.

5 Non-deterministic algorithms

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

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

These ideas, called **non-determinism** will be treated in several sections below.

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 $h_1, \dots, h_k$ 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 $h'_1, \dots, h'_k$ and moves $j_1, \dots, j_k \in \{-1, 0, 1\}$. More exactly, a non-deterministic Turing machine is an ordered 4-tuple $T = (k, \Sigma, \Gamma, \Phi)$ where $k \geq 1$ is a natural number, Σ and Γ are finite sets, $* \in \Sigma$, $\mathrm{START}, \mathrm{STOP} \in \Gamma$ (so far, everything is as with a deterministic Turing machine) and

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

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 $h_1, \dots, h_k \in \Sigma$ then for the new state g', the newly written symbols $h'_1, \dots, h'_k$ and the steps $\varepsilon_1, \dots, \varepsilon_k \in \{-1, 0, 1\}$ we have

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

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.

(5.1.1) **Remark**

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

2. The non-deterministic Turing machines do not serve for the modeling of any real computing device; we will see that these machines are tools for the *formulation* of certain problems rather than for their *solution*.

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

$\diamondsuit$

(5.1.2) **Theorem** *The languages recognizable by non-deterministic Turing machines are just the recursively enumerable languages.*

Proof Assume first that language $\mathcal{L}$ is recursively enumerable. Then, according to Lemma 3.1.3, 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 The complexity of non-deterministic algorithms

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

Let f and g be two functions that are well-computable in the sense of the definition in 4.4, 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.)

(5.2.1) **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* $\mathrm{NTIME}(g(n + 1 + f(n)))$.

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 5.1.2, 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 $O(f(|x|)) + g(|x| + 1 + f(|x|)) = O(g(|x| + 1 + f(x)))$. ∎

(5.2.2) **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.*

(5.2.3) **Remark** We mention it without proof, moreover, without exact formulation, that these properties are also equivalent to the following: one can give a definition of $\mathcal{L}$ in set theory with which, 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 5.2.2 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 $|y| \le f(n)$ 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

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

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

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

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

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

(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 $g \in \Gamma$, $-f(n) \le p \le f(n)$ 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 Σ.

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

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

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

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

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

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.

(5.2.6) **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:

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

5.1 **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. $\diamondsuit$

5.2 **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 $\mathrm{NP} \cap \mathrm{co{-}NP}$. $\diamondsuit$

5.3 **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.] $\diamondsuit$

5.3 Examples of languages in NP

In what follows, by a graph we understand a so-called **simple graph**: an undirected graph without multiple edges or loop edges, with n vertices. Such a graph can be uniquely described by the part of its adjacency matrix above the main diagonal which, written continuously, forms a word in $\{0,1\}^*$. 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 if the graph would be described in one of the other usual ways, e.g. by giving a list of neighbors for each point then this 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.

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

(5.3.2) Graph non-connectivity Witness: a proper subset of the set of points, which is not connected by any edge to the rest of the points. $\diamondsuit$

Let $p_1, p_2, \ldots, p_n$ be different points in the plane, and let l_i denote the line segment connecting p_i and p_{i+1}. The union of these line segments is a **simple polygon** if for all i, the line segment l_i intersects the union of line segments $l_1, l_2, \ldots, l_{i-1}$ only in the point p_i.

(5.3.3) Graph planarity A graph is **planar** if a picture of it can be drawn in the plane, where edges are represented by nonintersecting polygons. $\diamondsuit$

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 vertex are integers whose number of digits 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 vertices. We must be careful, however, since the coordinates of the vertices used in the drawing may have too many digits, violating the requirement on the length of the witness. (It can be proven 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 vertex are integers whose number of digits 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 vertices 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). 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:

(5.3.4) **Theorem** *If a connected planar graph has n points and m edges then it has $n+m-2$ countries.*

The sufficiency of this condition requires somewhat harder tools from topology; we will not go into these details. (Giving a set of edge sequences amounts to defining a two-dimensional surface with the graph drawn onto it. The theorem of topology mentioned says that if a graph on that surface satisfies Euler's formula then the surface is topologically equivalent (homeomorphic) to the plane.)

(5.3.5) NON-PLANARITY $\diamondsuit$

Let us review the following facts.

1. Let K_5 be the graph obtained by connecting five points in every possible way. This graph is also called a "complete pentagon". Let K_3^3 be the 6-point 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 **topological version** 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 each of its topological versions 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:

(5.3.6) **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 witness for this.

A **matching** is a set of edges that have no common nodes. A **complete matching** is a matching that covers all nodes.

(5.3.7) EXISTENCE OF COMPLETE MATCHING A witness is the complete matching itself. $\diamondsuit$

(5.3.8) NON-EXISTENCE OF A COMPLETE MATCHING $\diamondsuit$

Witnesses for the non-existence in case of bipartite graphs are based on a fundamental theorem. Let G be a bipartite graph G consisting of an "upper" and a "lower" set of points. If it has a complete matching then it has the same number of "upper" and "lower" points, and for any k, if we delete k upper points then this makes at most k lower points isolated. The following theorem says that these two conditions are also sufficient.

(5.3.9) **Frobenius's Theorem** *A bipartite graph G has a complete matching if and only if it has the same number of "upper" and "lower" points, and for any k, if we delete k upper points then this makes at most k lower points isolated.*

Hence, if in some bipartite graph there is no matching then for this, witness is the set of upper points violating the conditions of the theorem.

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

(5.3.10) **Tutte's Theorem** *In a graph, there is a complete matching if and only if for any k, if we delete any k nodes, there remain at most k connected components with odd size.*

In this way, if there is no complete matching in the graph then witness is a set of nodes whose deletion creates too many odd components.

The techniques solving the complete matching problem help also in solving the following problem, both in the bipartite and in the general case:

(5.3.11) General matching problem Given a graph G and a natural number k, does there exist a k-edge matching in G? $\diamondsuit$

(5.3.12) Existence of a Hamiltonian circuit
A Hamiltonian circuit of a graph is a circuit going through each node exactly once. It itself is the witness. $\diamondsuit$

(5.3.13) Colorability with three colors 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. If a graph can be colored with three colors the coloring itself is the witness. $\diamondsuit$

All properties listed above, up to (and including) the non-existence of complete matching, have also polynomial complexity. In the case of connectivity, this is easy to check (breadth-first or depth-first search). The first polynomial planarity-checking algorithm was given by Hopcroft and Tarjan. For the complete matching problem, in case of bipartite graph, we can use the "Hungarian method" (formulated by Kuhn, on the basis of works by König and Egerváry). For matchings in general graphs, Edmonds's algorithm is applicable. For the Hamiltonian circuit problem and the three-colorability problem, no polynomial algorithm is known (we return to this later).

In arithmetic and algebra, also many problems belong to the class NP. Every natural number can be considered a word in $\{0,1\}^*$ (representing the number in binary). In this sense, the following properties are in NP:

(5.3.14) Compositeness of an integer Witness of compositeness: a proper divisor. $\diamondsuit$

(5.3.15) Primality $\diamondsuit$

It is significantly more difficult to find witnesses for primality. We will describe a nondeterministic procedure $F(n)$ for this. We use the following fundamental theorem of number theory:

(5.3.16) **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.)

Using this theorem, we would like 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 digits than k, the number of digits of n. Using Lemma 4.1.3, we can also check the condition

$$a^{n-1} \equiv 1 \pmod n \tag{5.3.17}$$

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

$$a^m \not\equiv 1 \pmod n \qquad (1 \leq m < n-1). \tag{5.3.18}$$

We can do this for each specific m just as with (5.3.17), but (apparently) we must do this $n-2$ times, i.e. exponentially many times in terms of k. We use, however, the number-theoretical fact that if (5.3.17) holds then the smallest $m = m_0$ violating (5.3.18) (if there is any) is a divisor of $n-1$. It is also easy to see that then (5.3.18) 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 it 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 (5.3.18) 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. For this, we can call the nondeterministic procedure $F(p_i)$ 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 digits. Then, according to the above recursion,

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

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

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

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

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

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

It is not enough to decide about a number n whether it is a prime but if it is not a prime then we would 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 yes-no problem but it is not difficult to reformulate into such a problem:

(5.3.19) EXISTENCE OF A BOUNDED DIVISOR Given two natural numbers n and k; does n have a proper divisor not greater than k? The witness is the divisor. $\diamondsuit$

The complementary language is also in NP:

(5.3.20) 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 witness for this is the prime decomposition of n, together with a witness of the primality of every prime factor. $\diamondsuit$

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.

(5.3.21) REDUCIBILITY OF A POLYNOMIAL OVER THE RATIONAL FIELD
Witness: a proper divisor. $\diamondsuit$

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 up 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.

Systems of linear inequalities A system $Ax \le b$ 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:

(5.3.22) EXISTENCE OF SOLUTION $\diamondsuit$

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 digits. These facts follow from the elements of the theory of linear programming.

(5.3.23) NONEXISTENCE OF SOLUTION $\diamondsuit$

Witnesses for the non-existence of solution can be found using the following fundamental theorem known from linear programming:

(5.3.24) **Farkas's lemma** *The system $Ax \leq b$ of inequalities is unsolvable if and only if the following system of inequalities is solvable: $y^T A = 0$, $y^T b = -1$, $y \geq 0$.*

In words, this lemma says that a system of linear inequalities is unsolvable if and only if a contradiction can be obtained by a linear combination of the inequalities with nonnegative coefficients. Using this, a solution of the other system of inequalities given in the lemma (the nonnegative coefficients) is a witness of the nonexistence of a solution.

Let us now consider the existence of integer solution. The solution itself is a witness but we need some reasoning again to limit the size of witnesses, which is more complicated here (this is a result of VOTYAKOV and FRUMKIN).

It is interesting 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, moreover, such an algorithm cannot be expected (see later in these notes).

Reviewing the above list of examples, the following statements 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 not trivial; moreover, in various branches of mathematics, often the most fundamental theorems assert this for certain languages.

— It is often the case that if some property (language) turns out to be in NP $\cap$ co$-$NP then sooner or later it also turns out to be in P. This happened, for example, with the existence of complete 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 circuit, graph coloring, integer solution of a system of linear inequalities). We cannot prove that these are not in P (we don't know whether P = NP holds); but still, one can prove a certain 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 complete matching in a certain graph then we can search for complete matching in polynomial time in the following way: we delete edges from the graph as long as a complete matching still remains in it. When we get stuck, the remaining graph must be a complete matching. Using similar simple tricks, the search problem corresponding to the existence of Hamiltonian circuits, 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, together with a binary search, it will provide a polynomial algorithm to solve the associated optimization problem.

There are, of course, interesting 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-vertex complete graph with two colors, some color contains a copy of G. 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-vertex complete graph, and the number of these is $2^{N(N-1)/2}$.

5.4 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:

(5.4.1) **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 $\{0, 1\}$.

(5.4.2) **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.

(5.4.3) **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 5.4.3) we will prove the NP-completeness of many other problems.

A Boolean polynomial is called **satisfiable** if the Boolean function defined by it is not identically 0.

(5.4.4) 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. $\diamondsuit$

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

5.5 **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.) $\diamondsuit$

5.6 **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.] $\diamondsuit$

We can consider each conjunctive normal form also 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.

(5.4.5) **Cook's Theorem** *The language* SAT *is* NP*-complete.*

(The theorem was also independently discovered by Levin.)

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:

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

If a legal computation of the machine T is given then let us assign to these variables the following values: $x[n,g]$ 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:

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

and it should not be in two states:

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

We can require, similarly, that the head should be only in one position in each step and there should be one and only one symbol in each tape cell. We write that initially, the machine is in state START and at the end of the computation, in state STOP, and the head starts from cell 0:

$$x[0, \mathrm{START}] = 1, \qquad x[N, \mathrm{STOP}] = 1, \qquad y[0,0] = 1;$$

and, similarly, that the tape contains initially the input $h_1 \cdots h_n$ and finally the symbol 1 on cell 0:

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

We must further express the computation rules of the machine, *i.e.* , that for all $g, g' \in \Gamma$, $h, h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$ and $-N \le p \le N$ 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 convenient for the following to prove the NP-completeness of a special case 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.

(5.4.6) **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

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

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 requirement

$$y_i.$$

Each of these requirements involves only three variables and is therefore expressible as a 3-normal forms. The conjunction of all these is satisfiable if and only if B is satisfiable. ■

The question occurs naturally 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.)

(5.4.7) **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 point x_i^ε to point x_j^δ if $x_i^{1-\varepsilon} \vee x_j^\delta$ is an elementary disjunction in B. (This disjunction is equivalent to $x_i^\varepsilon \Rightarrow x_j^\delta$.) Let us notice that then in this graph, there is also an edge from $x_j^{1-\delta}$ to $x_i^{1-\varepsilon}$. In this directed graph, let us consider the **strongly connected components**; these are the classes of points obtained when we group two points in one class whenever there is a directed path between them.

(5.4.8) **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 5.4.8 Let us note first that if an assignment of values satisfies formula B and x_i^ε is "true" in this assignment then every x_j^δ is "true" to which an edge leads from x_i^ε: otherwise, the elementary disjunction $x_i^{1-\varepsilon} \vee x_j^\delta$ would not be satisfied. It follows from this that the points 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 point 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. ■

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

Proof Let B be a Boolean formula of the variables $x_1, \ldots, x_n$. For each variable x_j, replace the i-th occurrence of x_j in B, with new variable y_j^i: let the new formula be B'. For each j, assuming there are m occurrences of x_j in B, form the conjunction

$$C_j = (y_j^1 \Rightarrow y_j^2) \wedge (y_j^2 \Rightarrow y_j^3) \wedge \cdots \wedge (y_j^m \Rightarrow y_j^1).$$

(Of course, $y_j^1 \Rightarrow y_j^2 = \neg y_j^1 \vee y_j^2$.) 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. ■

5.7 **Exercise** Define the language 3-SAT-3 and show that it is NP-complete. ♢

5.5 Further NP-complete problems

In what follows, we will show the NP-completeness of various important languages. The majority of these are not of logical character but describe "everyday" combinatorial, algebraic, etc. problems. When we show about a problem that it is NP-complete then it follows that it can only be in P if P = NP. Though this equality is not refuted the hypothesis is rather generally accepted that it does not hold. 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:

(5.5.1) HITTING 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? ♢

(5.5.2) **Theorem** *The hitting 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 variable symbols and negated variable symbols occurring in it; let us further take the sets $\{x_i, \overline{x}_i\}$. The elements of this set system can be hit with at most n points if and only if the normal form is satisfiable. ■

The hitting problem remains NP-complete even if we impose various restrictions on the set system. It can be seen from the above construction that the hitting set problem is complete even for a system of sets with at most three elements. (We will see a little later that it is also achievable that only two-element sets (*i.e.* , the edges of a graph) occur.) If we reduce the language SAT first to the language SAT-3 according to Theorem 5.4.9 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. In a little more complex way, we could also reduce the problem to the hitting of a set system in which each element is in at most 3 sets. We cannot go further than this: if each element is in at most 2 sets then the set hitting problem is solvable in polynomial time (see Exercise 5.13).

It is easy to see that the following problem is equivalent to the hitting problem (only the roles of "elements" and "subsets" must be interchanged):

(5.5.3) 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 is NP-complete already even when each of the given subsets has at most 4 elements. It can also be proved that this problem is NP-complete when each subset has at most 3 elements. On the other hand, when each subset has only 2 elements, the problem becomes polynomially solvable, as the following exercise shows:

5.8 **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:

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

(5.5.5) Partition problem: Given a system $\{A_1, \ldots, A_m\}$ of subsets of a finite set S. Can a subsystem $\{A_{i_1}, \ldots, A_{i_k}\}$ be selected that gives a partition of the underlying set? $\diamondsuit$

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

Proof The problem we will reduce to the k-partition problem is the problem of covering with sets having at most 4 elements. Given is therefore a system of sets having at most 4 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 $2^4 = 16$-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 problem of complete matching and can therefore be solved in polynomial time. But it can be shown that for 3-element sets, the partition problem is already NP-complete.

Now we treat a fundamental graph-theoretic problem, the coloring problem. The problem of coloring in two colors is solvable in polynomial time.

5.9 **Exercise** Prove that the problem of coloring a graph in two colors is solvable in polynomial time. $\diamondsuit$

On the other hand:

(5.5.7) **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 points of the graph G, we first take the literals, and we connect each variable with its negation. We take two more points, 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 plays a key role in the proof: if for some elementary disjunction $z_{i_1} \vee z_{i_2} \vee z_{i_3}$, the points z_{i_1}, z_{i_2}, z_{i_3} and v are colored with three colors then this coloring can be extended to the pentagon as a legal coloring if and only if the colors of the four points 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 point, this coloring can be legally extended to the points 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 point v is blue and the point u is yellow. Then the points 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 point. But this also means that taking the red points 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.

In the set system constructed in the proof of Theorem 5.5.2 there were sets of at most three elements, for the reason that we reduced the 3-SAT problem to the hitting problem.

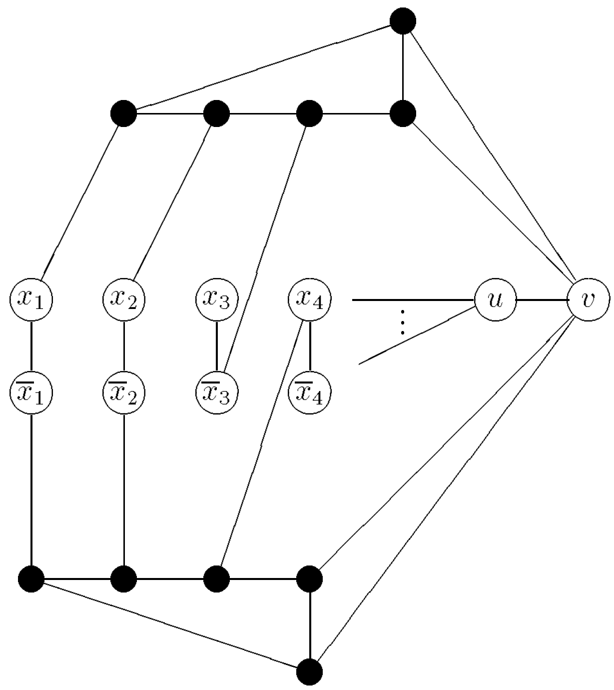


Figure 5.1: The graph whose 3-coloring is equivalent to satisfying the expression $(\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$

Since the 2-SAT problem is in P, we could expect that the hitting problem for two-element sets is in P. We note that this case is especially interesting since the issue here is the hitting of the edges of graphs. We can notice that the points outside a hitting 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 point can be added to it while preserving independence) then its complement is a hitting set for the edges. Our search for a minimum hitting 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:

(5.5.8) Independent point set problem: Given a graph G and a natural number k, are there k independent points in G? $\diamondsuit$

Unfortunately, this problem is not significantly easier than the general hitting problem:

(5.5.9) **Theorem** *The independent point 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 points 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 points of the three copies. Let H be the graph obtained, this has thus $3n$ points.

We claim that there are n independent points 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 points in G_1 corresponding to the red points, the points in G_2 corresponding to the blue points and the points in G_3 corresponding to the yellow points are independent even if taken together in H; and their number is n. The converse can be seen similarly. ∎

(5.5.10) **Remark** The independent vertex set problem (and similarly, the hitting 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 points it can be decided in polynomial time (in the given example, in time $O(n^{137})$) whether it has k independent points. The situation is different with colorability, where already the colorability with 3 colors is NP-complete. $\diamondsuit$

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

5.11 **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. $\diamondsuit$

5.12 **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. $\diamondsuit$

5.13 **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 hitting set problem with respect to this system is reducible to the general matching problem 5.3.11. $\diamondsuit$

5.14 **Exercise** Prove that for "hypergraphs", already the problem of coloring with two colors is NP-complete: Given a system $\{A_1, \dots, A_n\}$ of subsets of a finite set. Can the points of S be colored with two colors in such a way that each A_i contains points of both colors? $\diamondsuit$

Very many other important combinatorial and graph-theoretical problems are NP-complete: the existence of a Hamiltonial circuit, coverability of the points with disjoint triangles (for "2-angles", this is the matching problem!), the existence of point-disjoint paths connecting given point pairs, etc. The book "Computers and Intractability" by Garey and Johnson (Freeman, 1979) lists NP-complete problems by the hundreds.

A number of NP-complete problems is known also outside combinatorics. The most important one among these is the following.

(5.5.11) Diophantine inequality system: Given a system $Ax \le b$ of linear inequalities with integer coefficients, we want to decide whether it has a solution in integers. $\diamondsuit$

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

(5.5.12) **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 $x_1, \dots, x_n$. Let us write up the following inequalities:

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

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:

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

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

Proof We reduce the set-partition problem to the subset sum problem. Let $\{A_1, \dots, A_m\}$ be a family of subsets of the set $S = \{0, \dots, n-1\}$, we want to decide whether it has a subfamily giving a partition of S. Let $q = m+1$ and let us assign a number $a_i = \sum_{j \in A_i} q^j$ to each set A_i. Further, let $b = 1 + q + \cdots + q^{n-1}$. We claim that $A_{i_1} \cup \cdots \cup A_{i_k}$ is a partition of the set S if and only if

$$a_{i_1} + \cdots + a_{i_k} = b.$$

The "only if" is trivial. Conversely, assume $a_{i_1} + \cdots + a_{i_k} = b$. Let d_j be the number of those sets A_{i_r} that contain the element j $(0 \le j \le n-1)$. Then

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

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 is a good example to show that the coding of numbers can significantly influence the results. Let us assume namely that each number a_i is given 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 with respect to the length of the input.

(5.5.15) **Theorem** *In case of 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 5.5.12 which used only coefficients with absolute value at most 2.)

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 $T_p \subset \{0, \dots, \sum_i a_i\}$ 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 idea of this proof, that of keeping the results of recursive calls to avoid recomputation later, is called **dynamic programming**.

5.15 **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 $i = 1, \dots, m$, $j = 1, \dots, n$. The task is to see if the set of equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad (i = 1, \dots, m)$$

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. ♢

5.16 **Exercise** The SUM PARTITION PROBLEM is the following. Given a set $A = \{a_1, \dots, a_n\}$ 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.] ♢

5.17 **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. ♢

(5.5.16) **Remark** A function f is called NP-**hard** if it is not necessarily in NP but 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. Every NP-complete problem has an NP-hard characteristic function but there are languages with NP-hard characteristic functions that are definitely harder than any NP problem (e.g., to decide from a position of the GO game on an $n \times n$ board, who is the winner).

There are many 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 traveling salesman problem (a "cost" is assigned to each edge of a graph and search for the minimum cost Hamilton circuit);

— the Steiner problem (under the previous conditions, search for a minimum-cost connected graph containing given vertices);

— the knapsack problem (the optimization problem associated with a more general version of the subset sum problem);

— a large part of the scheduling problems.

Many enumeration problems are also NP-hard (e.g., to determine the number of all complete matchings, Hamilton circuits or legal colorings). $\diamondsuit$

(5.5.17) **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:

— Is a given natural number a prime?

— Does a given natural number n have a proper divisor not greater than k?

— Are two given graph isomorphic?

The prime number problem is probably in P (this is proved using an old number-theoretical conjecture, the so-called Generalized Riemann Conjecture; see the section on "Randomized algorithms"). For the problems of bounded divisor and isomorphism, it is rather expected that they are not in P but are also not NP-complete. $\diamondsuit$

(5.5.18) **Remark** When a problem turns out to be NP-complete we cannot hope to find for it such an efficient, polynomial algorithm 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; exponential algorithms that are fairly well usable for not too large inputs; heuristics 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. $\diamondsuit$

5.18 **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. $\diamondsuit$

5.19 **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. $\diamondsuit$

6 Randomized algorithms

We cited Church's Thesis in Section 3: every "algorithm" (in the heuristic meaining of the word) is realizable on, e.g., a Turing machine. If in an algorithm, we permit "coin tossing" i.e., the generation of a random number as an elementary step, then provably, we will be able to solve problems the Turing machine cannot (we will show this in a later section). In other cases, the random choice significantly accelerates the algorithms. We will see examples for this in the present section.

Since in this way, we obtain a new, stronger mathematical notion of a machine, corresponding complexity classes can also be introduced. Some of the most important of these will be treated at the end of the section.

6.1 Verifying a polynomial identity

Let $f(x_1,\dots,x_n)$ 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:

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

$$\text{Prob}\{f(\xi_1,\dots,\xi_n)=0\}\le\frac{kn}{N}.$$

Proof We prove the assertion by mathematical induction on n. The statement is true for $n = 1$ since a polynomial 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 $t \le k$. Now,

$$\begin{aligned}
&\mathrm{Prob}\{f(\xi_1,\ldots,\xi_n) = 0\} \\
&\le \ \mathrm{Prob}\{f(\xi_1,\ldots,\xi_n) = 0 \mid f_t(\xi_2,\ldots,\xi_n) = 0\}\mathrm{Prob}\{f_t(\xi_2,\ldots,\xi_n) = 0\} \\
&+ \ \mathrm{Prob}\{f(\xi_1,\ldots,\xi_n) = 0 \mid f_t(\xi_2,\ldots,\xi_n) \ne 0\}\mathrm{Prob}\{f_t(\xi_2,\ldots,\xi_n) \ne 0\} \\
&\le \ \mathrm{Prob}\{f_t(\xi_2,\ldots,\xi_n) = 0\} + \mathrm{Prob}\{f(\xi_1,\ldots,\xi_n) = 0 \mid f_t(\xi_2,\ldots,\xi_n) \ne 0\}.
\end{aligned}$$

Here, we can estimate the first term by the inductive assumption, and the second term is at most k/N (since ξ_1 is independent of the variables $\xi_2, \ldots, \xi_n$, therefore if the latter are fixed in such a way that $f_t \ne 0$ and therefore f as a polynomial of x_1 is not identically 0 then the probability that ξ_1 is its root is at most k/N). Hence

$$\mathrm{Prob}\{f(\xi_1,\ldots,\xi_n) = 0\} \le \frac{k(n-1)}{N} + \frac{k}{N} \le \frac{kn}{N}.$$

∎

This offers the following a **randomized** algorithm (i.e., one that uses randomness) to decide whether a polynomial f is identically 0:

(6.1.2) **Algorithm** *We compute $f(\xi_1, \ldots, \xi_n)$ with integer values ξ_i chosen randomly and independently of each other according to the uniform distribution in the interval $[0, 2kn]$. 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 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 Subsection 4.1). 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 (compare with Theorem 5.3.9):

(6.1.3) **Theorem** *There is a complete matching in the bipartite graph G if and only if* $\det(M)$ *is not identically 0.*

Proof Consider a term in the expansion of the determinant:

$$\pm m_{1\pi(1)} m_{2\pi(2)} \cdots m_{n\pi(n)},$$

where π is a permutation of the numbers $1, \ldots, n$. For this not to be 0, we need that a_i and $b_{\pi(i)}$ be connected for all i; in other words, that $\{a_1 b_{\pi(1)}, \ldots, a_n b_{\pi(n)}\}$ be a complete matching in G. In this way, if there is no complete matching in G then the determinant is identically 0. If there are complete 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 complete 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:

(6.1.4) **Theorem** *t:MatchingDet There is a complete 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 complete matching in G.

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.] $\diamondsuit$

6.2 **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.

$\diamondsuit$

6.2 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.

(6.2.1) **"Little" Fermat Theorem** *If m is a prime then $a^{m-1} - 1$ is divisible by m for all natural numbers $1 \le a \le m - 1$.*

If—with 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 $1 \le a \le m-1$, also characterizes the primes since if m is a composite number, and we choose for a any integer not relatively prime to m, then $a^{m-1} - 1$ is obviously not divisible by m. 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? There are composite numbers m (the so-called pseudo-primes) for which the Fermat condition is satisfied for all primitive residue classes a; for such numbers, it will be especially difficult to find an integer a violating the condition. (Such pseudo-prime is e.g. $561 = 3 \cdot 11 \cdot 17$.)

Let us recall that the set of integers giving identical remainder after division by a number m is called a **residue class** modulo m. This residue class is **primitive** if its members are relatively prime to m (this is satisfied obviously at the same time for all elements of the residue class, just as the Fermat condition). In what follows the residue classes that satisfy the Fermat condition (and are therefore necessarily relatively prime to m) will be called **Fermat-accomplices**. The residue classes violating the condition are, on the other hand, the **traitors**.

(6.2.2) **Lemma** *If m is not a pseudo-prime then at most half of the modulo m primitive residue classes is a Fermat-accomplice.*

Note that none of the non-primitive residue classes are Fermat accomplices.

Proof If we multiply all accomplices by a traitor relatively prime to m then we get all different traitors. ∎

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 though happen that m is composite but if m is not a pseudo prime then the probability to have found an integer a satisfying the condition was less than $1/2$ in every step, and the probability that this should occur 100 consecutive times is less than 2^{-100}.

Unfortunately, this method fails for pseudoprimes (it finds them prime with large probability). We will therefore modify the Fermat condition somewhat. Let us write the number $m-1$ in the form $2^k M$ where M is odd. We say that M **satisfies the Miller condition** if at least one of the numbers

$$a^M - 1, a^M + 1, a^{2M} + 1, a^{4M} + 1, \dots, a^{2^{k-1}M} + 1$$

is divisible by m. Since the product of these numbers is just $a^{m-1}-1$, every number satisfying the Miller condition also satisfies the Fermat condition (but not conversely, since m can be composite and thus can be a divisor of a product without being the divisor of any of its factors).

(6.2.3) **Lemma** *m is a prime if and only if every integer $1 \leq a \leq m-1$ satisfies the Miller condition.*

Proof

1. If m is composite then each of its proper divisors violates the Miller condition.

2. Suppose that m is prime. Then according to the Fermat condition, $a^{m-1} - 1$ is divisible by m for every integer $1 < a < m$. This number can be decomposed, however, into the product

$$a^{m-1} - 1 = (a^M - 1)(a^M + 1)(a^{2M} + 1)(a^{4M} + 1) \cdots (a^{2^{k-1}M} + 1).$$

Hence (using again the fact that m is a prime) one of these factors must also be divisible by m, i.e. a satisfies the Miller condition. ∎

We will need some fundamental remarks on divisibility in general and on pseudoprimes in particular.

(6.2.4) **Lemma** *Every pseudoprime m is*

(a) *Odd*

(b) *Squarefree (is not divisible by any square).*

Proof

(a) If $m > 2$ is even then $a = -1$ (if somebody prefers positive remainders they can take $m-1$) will be a Fermat traitor since $(-1)^{m-1} \equiv -1 \not\equiv 1 \pmod{m}$.

(b) Assume that $p^2 \mid m$; let k be the largest exponent for which $p^k \mid m$. Then $a = m/p - 1$ is a Fermat traitor 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. ∎

(6.2.5) **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 $1 \le i \le t$.*

Proof If $p_i - 1$ divides k for all i with $1 \le i \le t$ 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 spelled out in Theorem 5.3.16). According to the Chinese Remainder Theorem, there is a residue class h modulo m with $h \equiv g \pmod{p_1}$ and $h \equiv 1 \pmod{p_i}$ for all $i \ge 2$. Then $(h, m) = 1$ and $p_1 \not| h^k - 1$, so $m \not| h^k - 1$. ∎

(6.2.6) **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, $t \ge 2$ and $(p_i - 1)$ divides $(m-1)$ for all i with $1 \le i \le t$.*

(6.2.7) **Remark** This is how one can show about the above example 561 that it is a pseudoprime. ♢

The key idea of the algorithm is the result that in case of a composite number—contrary to the Fermat condition—the majority of the residue classes violates the Miller condition.

(6.2.8) **Theorem** *If m is a composite number then at least half of the primitive residue classes modulo m violate the Miller condition* (we can call these Miller traitors).

Proof Since we have already seen the truth of the lemma for non-pseudoprimes, in what follows we can assume that m is a pseudoprime. Let $p_1 \cdots p_t$ $(t \ge 2)$ be the prime decomposition of m. According to the above Corollary, we have $(p_i - 1) \mid (m-1) = 2^k M$ for all i with $1 \le i \le t$.

Let l be the largest exponent with the property that none of the numbers $p_i - 1$ divides $2^l M$. Since the numbers $p_i - 1$ are even while M is odd, such an exponent exists (e.g. 0) and clearly $0 \le l < k$. Further, by the definition of l, there is a j for which $p_j - 1$ divides $2^{l+1}M$. Therefore $p_j - 1$ divides $2^s M$ for all s with $l < s \le k$, 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 $1 \le i \le t$. Since $p_i - 1$ does not divide the exponent $2^l M$, Lemma 6.2.5 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. (This is actually a Fermat traitor belonging to the exponent $2^l M$, mod p_i—of course, not mod m!) The reasoning of Lemma 6.2.2 shows that then at most half of the mod p_i residue classes will be Fermat accomplices 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 part of the primitive residue classes is such that every p_i divides $(a^{2^l M} - 1)$. Therefore at most a 2^t-th part 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 $t \ge 2$) at most a half. ■

(6.2.9) **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 Lemma 4.1.3: the remainder of a^b modulo c is computable in polynomial time. Based on these three lemmas, the following randomized algorithm can be given for prime testing:

(6.2.10) **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 Rabin condition is satisfied 100 times consecutively then we declare that m is a prime. $\diamondsuit$

If m is a prime then the algorithm will certainly assert this. If m is composite then the number a chosen randomly violates the Rabin condition with probability 1/2. After hundred independent experiments the probability will therefore be at most 2^{-100} that the Rabin condition is not violated even once, i.e., that the algorithm asserts that m is a prime.

(6.2.11) **Remarks**

1. If m is found composite by the algorithm then—interestingly—we see this not from its finding a divisor but (essentially) 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 polynomial algorithm in the case when the Fermat condition is also violated. This problem is significantly more difficult also in practice than the decision 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 Rabin condition not by random choice but by trying out the numbers 1,2, etc. It is not known how small is the first such integer if m is composite. Using, however, a hundred year old conjecture of analytic number theory, the so-called Generalized Riemann Hypothesis, one can show that it is not greater than $\log_2 m$. 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. $(\log e)/n$. 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 $[1, 2^n]$.

6.3 **Exercise** Show that if m is a pseudoprime then the above algorithm not only discovers this with large probability but it can also be used to find a decomposition of m into two factors. $\diamondsuit$

6.3 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 $g \in \Gamma$, $h_1, \dots, h_k \in \Sigma$ on the set

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

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

(6.3.1) **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. $\diamondsuit$

6.4 **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. $\diamondsuit$

6.5 **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. $\diamondsuit$

It can be seen that every language in RP is also in NP. It is trivial that the classes BPP and ΔRP 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" gave a useful characterization of the class NP. An analogous theorem holds also for the class RP.

(6.3.2) **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 ΔRP is closer than it could be expected on the basis of the analogy to the classes NP and P:

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

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

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

(iii) *There is a randomized Turing machine with polynomial (worst-case) running time that can write, besides the symbols "0" and "1", also the words "I GIVE UP"; the answers "0" and "1" are never wrong, i.e., in case of $x \in \mathcal{L}$ the result is "1" or "I GIVE UP", and in case of $x \notin \mathcal{L}$ it is "0" or "I GIVE UP". The probability of the answer "I GIVE UP" is at most* $1/2$.

Proof It is obvious that (i) implies (ii). It can also be easily seen that (ii) implies (iii). Let us submit x to a randomized Turing machine that accepts $\mathcal{L}$ in polynomial time and also to one that accepts $\Sigma^* \setminus \mathcal{L}$ in polynomial time. If the two give opposite answers then the answer of the first machine is correct. If they give identical answers then we "give it up". In this case, one of them made an error and therefore this has a probability at most 1/2.

Finally, to see that (iii) implies (i) we just have to modify the Turing machine T_0 given in (iii) in such a way that instead of the answer "I GIVE IT UP", it should start again. If on input x, the number of steps of T_0 is τ and the probability of giving it up is p then on this same input, the expected number of steps of the modified machine is

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

∎

We have seen in the previous subsection that the "language" of composite numbers is in RP. Even more is true: lately, Adleman and Huang have shown that this language is also in ΔRP. 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 ΔRP but no polynomial algorithm is known for their solution.

(6.3.4) **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 beginnings, that we will not treat here. ♢

6.6 **Exercise** Let us call a Boolean formula with n variables *simple* 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 simple formulas. ♢

7 Information complexity (the complexity-theoretic notion of randomness)

The mathematical foundation of probability theory appears among the above-mentioned famous problems of Hilbert formulated in 1900. Von Mises made an important attempt in 1919 to define the randomness of a 0-1 sequence by requiring the frequency of 0's and 1's to be approximately the same and requiring this to be true also, e.g., for all subsequences selected by an arithmetical sequence. This approach did not prove sufficiently fruitful. Kolmogorov started in an other direction in 1931, using measure theory. His theory was very successful from the point of view of probability theory but it failed to capture some important questions. So, e.g., in the 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, though in an everyday sense, about the sequence "Head,Head,Head,...", it is "obvious" in itself that it cannot be the result of coin tossing. In the 1960's, Kolmogorov (and later, Chaitin) revived Mises's idea, using complexity-theoretical tools. The interest of their results points beyond the foundation problem of probabity theory; it contributes to the clarification of the basic notions of several field, among others, data compression, information theory, statistics (inductive inference).

7.1 Information complexity

Fix an alphabet Σ. Let $\Sigma_0 = \Sigma \setminus \{*\}$ and consider a two-tape universal Turing machine over Σ. It will be convenient to identify Σ_0 with the set $\{0, 1, \ldots, m-1\}$. Consider a 2-tape, universal Turing machine T over Σ. We say that the word (program) q over Σ **prints** word x if when we write q on the second tape of T leaving the first tape empty, the machine stops in finitely many steps having the word x on its first tape. Let us note right away that every word is printable on T. There is namely a one-tape (rather trivial) Turing machine S_x that does not do anything with the empty tape but writes the word x onto it. This Turing machine can be simulated by a program q_x that, in this way, prints x.

On the **complexity**, or, more completely, **description complexity**, or **information complexity** of a word $x \in \Sigma_0^*$ we mean the length of the shortest word (program) printing on T 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 on how much time this decoding (or coding, the finding of 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 "slides over" the intermediate letters: then every word must be defined twice as complex as when these letters would not even have to be written down.

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 will we be using for the definition of complexity. Crudely 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 before start, on the program tape of the machine, we write a word of the form xDATAy where the word x already does not contain the subword then the machine halts if and only if it would halt when started with y written on the data tape and x on the program tape, and at halting, the content of the data tape is the same.

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 this property.

(7.1.1) **Lemma** *There is a constant c_T (depending only on T) such that $\mathbf{K}_T(x) \le |x| + c_T$.*

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_0DATAx will print the word x. The constant $c_T = |p_0| + 4$ satisfies therefore the conditions. ∎

In what follows we assume, for simplicity, that $c_T \le 100$.

(7.1.2) **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.

(7.1.3) **Invariance Theorem** *Let T and S be a universal Turing machine satisfying the conditions $(a),(b)$. Then there is a constant c_{TS} such that for every word x we have $|\mathbf{K}_T(x) - \mathbf{K}_S(x)| \le c_{TS}$.*

Proof We can simulate the work of 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 tape of S_0, it also stops in finitely many steps, having x written 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.

Let now x be an arbitrary word from Σ_0^* and let q_x be a shortest program printing x on S. Consider on T the program p_{S_0}DATAq_x: this obviously prints x and has only length $|q_x| + |p_{S_0}| + 4$. The inequality in the other direction is obtained similarly. ∎

On the basis of this lemma, we will consider T fixed and do not write out the index T from now on.

7.1 **Exercise** Suppose that the universal Turing machine used in the definition of $\mathbf{K}(x)$ uses programs written in a two-letter alphabet and outputs strings in an s-letter alphabet.

(a) Prove that $\mathbf{K}(x) \le |x| \log_2 s + O(1)$.

(b) Prove that, moreover, there are a polynomial-time functions f, g mapping strings x of length n to binary strings of length $n \log_2 s + O(1)$ and vice versa with $g(f(x)) = x$.

$\diamondsuit$

7.2 **Exercise**

(a) Give an upper bound on the Kolmogorov complexity of Boolean functions of n variables.

(b) Give a lower bound one the complexity of the most complex Boolean function of n variables.

(c) Use the above result to find a number $L(n)$ such that there is a Boolean function with n variables which needs a Boolean circuit of size at least $L(n)$ to compute it.

$\diamondsuit$

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

(7.1.4) **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 now that $\mathbf{K}(x)$ is computable. Let c be a natural number to be chosen appropriately. Arrange the elements of Σ_0^* in increasing order. Let $x(k)$ denote the k-th word according to this ordering and let x_0 be the first word with $\mathbf{K}(x_0) \ge c$. 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.
```

This program obviously prints x_0. When determining its length we must take into consideration the programs for the computation of the functions $x(k)$ and $Kolm(k) = \mathbf{K}(x(k))$ (where $x(k)$ is the k-th string). Even when taken together, the number of all these symbols is, however, only $\log c+$ some constant. 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. Why is it namely not possible to compute x as follows? Let us take the words in order and try to see whether, when we write them on the program tape of T, it will halt in a finite number of steps, having written x on the data tape. Suppose that the halting problem is solvable. Then there is an algorithm that about a given program decides whether, when we write it on the program tape, T halts in a finite number of steps. It helps "filter out" the programs on which T would work forever, and we will not even try these. Among the remaining words, the length of the first one with which T prints x is $\mathbf{K}(x)$.

According to the above theorem, this "algorithm" cannot work; its only problem can be, however, that we cannot filter out the programs running for infinite time, i.e., that the halting problem is not decidable.

7.3 **Exercise** Show that we cannot compute the function $\mathbf{K}(x)$ even approximately, in the following sense: If f is a recursive function then there is no algorithm that for every word x computes a natural number $\gamma(x)$ such that for all x

$$\mathbf{K}(x) \le \gamma(x) \le f(\mathbf{K}(x)).$$

$\diamondsuit$

7.4 **Exercise** Show that there is no algorithm that for every given number n constructs a 0-1 sequence of length n with $\mathbf{K}(x) > 2\log n$. $\diamondsuit$

7.5 **Exercise** If $f \le \mathbf{K}$ for a recursive function $f : \Sigma_0^* \to \mathbf{Z}_+$ then f is bounded. $\diamondsuit$

In contrast to Theorem 7.1.4 and Exercise 7.4, we show that the complexity $\mathbf{K}(x)$ can be very well approximated *for almost all x*. For this, we must first make it precise what we understand by "almost all" x. Assume that the incoming words are supplied randomly; in other words, every word $x \in \Sigma_0^*$ has a probability $p(x)$. We know therefore

$$p(x) \ge 0, \qquad \sum_{x\in\Sigma_0^*} p(x) = 1.$$

Beyond this, we must only assume that $p(x)$ is *algorithmically computable* (each $p(x)$ is assumed to be a rational number whose numerator and denominator are computable from x). A function with such a property will be called a **computable probability distribution**. A simple example of computable probability distributions 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.

(7.1.5) **Remark** There is a more natural and more general notion of computable probability distribution than the one given here, that does not restrict probabilities to rational numbers: $\{e^{-1}, 1-e^{-1}\}$ would also be considered a computable probability distribution. Our theorems would also hold for this more general class. $\diamondsuit$

(7.1.6) **Theorem** *For every computable probability distribution there is an algorithm computing a Kolmogorov code $f(x)$ for every word x with the property that the expected value of $|f(x)| - \mathbf{K}(x)$ is finite.*

Proof Let $x_1, x_2, \ldots$ be an ordering of the words in Σ_0^* for which $p(x_1) \geq p(x_2) \geq \cdots$, and the words with equal probability are, say, in increasing order.

(7.1.7) **Lemma** *The word x_i is algorithmically computable from the index i.*

Proof Let $y_1, y_2, \ldots$ be the words arranged in increasing order. Let k be $i, i+1, \ldots$; let us compute the numbers $p(y_1), \ldots, p(x_k)$ and let t_i be the i-th largest among these. Obviously, $t_i \leq t_{i+1} \leq \cdots$ and $t_k \leq p(x_i)$. Further, if

$$p(y_1) + \cdots + p(y_k) \geq 1 - t_k \tag{7.1.8}$$

then none of the remaining words can have a larger probability than t_k hence $t_k = p(x_i)$ and x_i is the first y_j with $p(y_j) = t_k$.

Thus, taking the values $k = i, i+1, \ldots$, we can stop if the inequality 7.1.8 holds. Since the left-hand side converges to 1 while the right-hand side is monotonically non-decreasing, this will occur sooner or later. This proves the statement. ∎

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, $|f(x)| \geq \mathbf{K}(x)$. Further, the expected value of $|f(x)| - \mathbf{K}(x)$ is

$$\sum_{i=1}^{\infty} p(x_i)(|f(x_i)| - \mathbf{K}(x_i)) = \sum_{i=1}^{\infty} p(x_i)|f(x_i)| - \sum_{i=1}^{\infty} p(x_i)\mathbf{K}(x_i).$$

Let $m = |\Sigma_0|$. We assert that both sums deviate from the sum $\sum_{i=1}^{\infty} p(x_i) \log_m i$ only by a bounded amount. Take the first term:

$$\begin{aligned}\sum_{i=1}^{\infty} p(x_i)|f(x_i)| &\leq \sum_{i=1}^{\infty} p(x_i)(\log_m i + O(1)) \leq \sum_{i=1}^{\infty} p(x_i) \log_m i + O(1) \sum_{i=1}^{\infty} p(x_i) \\ &= \sum_{i=1}^{\infty} p(x_i) \log_m i + O(1).\end{aligned}$$

On the other hand, in the second term, the number of those terms with $\mathbf{K}(x_i) = k$ is at most m^k. We decrease the sum if we rearrange the numbers $\mathbf{K}(x_i)$ in increasing order (since the coefficients $p(x_i)$ are decreasing). After the rearrangement, the coefficient of $p(x_i)$ is the i-th smallest $\mathbf{K}(x_i)$, which is at most $\log_m i$. Thus, the sum is at least $\sum_i p(x_i) \log_m i$. ∎

7.2 Self-delimiting information complexity

The Kolmogorov-code, strictly taken, uses an extra symbol besides the alphabet Σ_0: it recognizes the end of the program while reading the program tape by encountering the symbol "$*$". We can modify the concept in such a way that this should not be possible: the head reading the program should not run beyond program. We will call a word **self-delimiting** if, when it is written on the program tape of our two-tape universal Turing machine, the head does not even try to read any cell beyond it. The length of the shortest self-delimiting program printing x will be denototed by $\mathbf{H}_T(x)$. This modified information complexity notion was introduced by LEVIN and CHAITIN. It is easy to see that the Invariance Theorem here also holds and therefore it is again justified to use the indexless notation $\mathbf{H}(x)$. The functions $\mathbf{K}$ and $\mathbf{H}$ do not differ too much, as it is shown by the following lemma:

(7.2.1) **Lemma**
$\mathbf{K}(x) \le \mathbf{H}(x) \le \mathbf{K}(x) + 2\log_m(\mathbf{K}(x)) + O(1)$.

Proof The first inequality is trivial. To prove the second inequality, let p be a program of length $\mathbf{K}(x)$ for printing x on some machine T. Let $n = |p|$, let $u_1 \cdots u_k$ be the form of the number n in the base m number system. Let $u = u_1 0 u_2 0 \cdots u_k 011$. Then the prefix u of the word up can be uniquely reconstructed, and from it, the length of the word can be determined without having to go beyond its end. Using this, it is easy to write a self-delimiting program of length $2k + n + O(1)$ that prints x. ∎

From the foregoing, it may seem that the functio $\mathbf{H}$ is a slight technical variant of the Kolmogorov complexity. The next lemma shows a significant difference between them.

(7.2.2) **Lemma**

(a) $\sum_x m^{-\mathbf{K}(x)} = +\infty$.

(b) $\sum_x m^{-\mathbf{H}(x)} \le 1$.

Proof The statement (a) follows immediately from Lemma 7.1.1. For the purpose of proving the statement (b), consider an optimal code $f(x)$ for each word x. Due to the self-delimiting, neither of these can be a prefix of another one; thus, (b) follows immediately from the simple but important information-theoretical lemma below. ∎

(7.2.3) **Lemma** *Let $\mathcal{L} \subset \Sigma_0^*$ be a language such that neither of its words is a prefix of another one. Let $m = |\Sigma_0|$. Then*

$$\sum_{y\in\mathcal{L}} m^{-|y|} \le 1.$$

Proof Choose letters $a_1, a_2, \ldots$ independently, with uniform distribution from the alphabet Σ_0; stop if the obtained word is in $\mathcal{L}$. The probability that we obtained a word $y \in \mathcal{L}$ is exactly $m^{-|y|}$ (since according to the assumption, we did not stop on any prefix of y). Since these events are mutually exclusive, the statement of the lemma follows. ∎

The following exercises formulate a few consequences of Lemmas 7.2.1 and 7.2.2.

7.6 **Exercise** Show that the following strengthening of Lemma 7.2.1 is not true:

$$\mathbf{H}(x) \le \mathbf{K}(x) + \log_m \mathbf{K}(x) + O(1).$$

♢

7.7 **Exercise** The function $\mathbf{H}(x)$ is not recursive. $\diamondsuit$

The next theorem shows that the function $\mathbf{H}(x)$ can be approximated well.

(7.2.4) **Coding Theorem** *Let p be a computable probability distribution on Σ_0^*. Then for every word x we have*

$$\mathbf{H}(x) \leq -\log_m p(x) + O(1).$$

Proof Let us call m**-ary rational** those rational numbers that can be written with a numerator that is a power of m. The m-ary rational numbers of the interval $[0,1)$ can be written in the form $0.a_1 \dots a_k$ where $0 \leq a_i \leq m-1$.

Subdivide the interval $[0,1)$ beginning into left-closed, right-open intervals $J(x_1), J(x_2), \dots$ of lengths $p(x_1), p(x_2), \dots$ respectively (where $x_1, x_2, \dots$ is a size-ordering of Σ_0^*). For every $x \in \Sigma_0^*$ with $p(x) > 0$, there will be an m-ary rational number $0.a_1 \dots a_k$ with $0.a_1 \dots a_k \in J(x)$ and $0.a_1 \dots a_{k-1} \in J(x)$. We will call a shortest sequence $a_1 \dots a_k$ with this property the **Shannon-Fano code** of x.

We claim that every word x can be computed easily from its Shannon-Fano code. Indeed, for the given sequence $a_1, \dots, a_k$, for values $i = 1, 2, \dots$, we check consecutively whether $0.a_1 \dots a_k$ and $0.a_1 \dots a_{k-1}$ belong to the same interval $J(x)$; if yes, we print x and stop. Notice that this program is self-delimiting: we need not know in advance how long is the code, and if $a_1 \dots a_k$ is the Shannon-Fano code of a word x then we will never read beyond the end of the sequence $a_1 \dots a_k$. Thus $\mathbf{H}(x)$ is not greater than the common length of the (constant-length) program of the above algorithm and the Shannon-Fano code of x; about this, it is easy to see that it is at most $\log_m p(x) + 1$. ∎

This theorem implies that the expected value of the difference between $\mathbf{H}(x)$ and $-\log_m p(x)$ is bounded (compare with Theorem 7.1.6).

(7.2.5) **Corollary** *With the conditions of Theorem 7.2.4*

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)| = O(1).$$

Proof

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|$$
$$= \sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_+ \sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_-.$$

Here, the first sum can be estimated, according to Theorem 7.2.4, as follows:

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_+ \leq \sum_x p(x)O(1) = O(1).$$

We estimate the second sum as follows:

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_- \leq m^{-\mathbf{H}(x) - \log_m p(x)} = \frac{1}{p(x)} m^{-\mathbf{H}(x)},$$

and hence according to Lemma 7.2.2,

$$\sum_x p(x)|\mathbf{H}(x) + \log_m p(x)|_- \le \sum_x m^{-\mathbf{H}(x)} \le 1.$$

∎

(7.2.6) **Remark** The following generalization of the coding theorem is due to Levin.

We say that $p(x)$ is a **semicomputable semimeasure** over Σ_0^* if $p(x) \ge 0$, $\sum_x p(x) \le 1$ and there is a computable function $g(x, n)$ taking rational values such that $g(x, n)$ is monotonically increasing in n and $\lim_{n\to\infty} g(x, n) = p(x)$.

Levin proved the coding theorem for the more general when $p(x)$ is a semicomputable semimeasure. Lemma 7.2.2 shows that $m^{-\mathbf{H}(x)}$ is a semicomputable semimeasure. Therefore Levin's theorem implies that $m^{-\mathbf{H}(x)}$ is maximal, to within a multiplicative constant, among all semicomputable semimeasures. This is a technically very useful characterization of $\mathbf{H}(x)$. ♢

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.

Crudely speaking, we want to consider a sequence random if there is no regularity in it. Here, we consider the kind of regularity that would enable a more economical coding of the sequence, i.e., the complexity of the sequence would be small.

(7.3.1) **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. As we will see later that this kind of regularity is compatible with 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 the "average" 0-1 sequences.

(7.3.2) **Lemma** *The number of 0-1 sequences x of length n with $\mathbf{K}(x) \le n - k$ 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. ∎

(7.3.3) **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 $|\mathbf{K}(x) - n| \le 100$ 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 7.1.4, this problem is algorithmically unsolvable. On the other hand, the above lemma shows that with large probability, a randomly chosen sequence is appropriate.

According to Theorem 7.1.4, it is algorithmically impossible to find the best code. There are, however, some easily recognizable properties telling about a word that it is codable more efficiently than its length. The next lemma shows such a property:

(7.3.4) **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 ε.

Proof x can be given as the "lexicographically t-th one among the 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 $\log_2\binom{n}{k} + 2\log_2 n + @\log_2 k$ bits. Here, the factor 2 is due to the need to separate the three pieces of information from each other; we can use the trick of the proof of Lemma 7.2.1). The program choosing the appropriate sequence needs only constantly many bits.

The estimate of the binomial coefficient is done by the method familiar from probability theory. ∎

On the basis of the above, one consider $|x| - \mathbf{K}(x)$ (or $|x|/K(x)$) a measure of the randomness of the word x. In case of infinite sequences, a more sharp difference can be made: we can define whether a given sequence is random. Several definitions are possible depending on whether we use the function $\mathbf{H}$ or $\mathbf{K}$, or whether we want to consider more or fewer sequences random. We introduce here the two (still sensible) "extremes". Let x be an infinite 0-1-sequence, and let x_n denote its starting segment formed by the first n elements. We call the sequence x **(informatically) weakly random** if $\mathbf{K}(x_n)/n \to 1$ when $n \to \infty$; we call the sequence **(informatically) strongly random** if $n - \mathbf{H}(x_n)$ is bounded from above.

Lemma 7.2.1 implies that every informatically strongly random sequence is also weakly random. It can be shown that every informatically weakly random sequence satisfies the laws of large numbers. The strongly random sequences pass also much stronger tests, e.g. various statistical tests, etc. 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:

(7.3.5) **Theorem** *If x is informatically weakly 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 are such:

(7.3.6) **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 each k, let S_k be the set of all those finite sequences y for which $\mathbf{H}(y) < |y| - k$ and let A_k denote the event that there is an n with $x_n \in S_k$. Then according to Lemma 7.2.2,

$$\mathrm{Prob}(A_k) \le \sum_{y \in S_k} 2^{-|y|} < 2^{-k} \sum_{y \in S_k} 2^{-\mathbf{H}(y)} \le 2^{-k},$$

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

7.4 Kolmogorov complexity and entropy

Let $p = (p_1, p_2, \ldots)$ be a **discrete probability distribution**, i.e., a non-negative (finite or infinite) sequence with $\sum_i p_i = 1$. Its **entropy** is the quantity

$$H(p) = \sum_i -p_i \log p_i$$

(the term $p_i \log p_i$ is considered 0 if $p_i = 0$). Notice that in this sum, all terms are nonnegative, so $H(p) \ge 0$; equality holds if and only if the value of some p_i is 1 and the value of the rest is 0. It is easy to see that for fixed m, the probability distribution with maximum entropy is $(1/m, \ldots, 1/m)$ and the entropy of this is $\log m$.

Entropy is a basic notion of information theory and we do not treat it in detail in these notes, we only point out its connection with Kolmogorov complexity. We have met with entropy for the case $m = 2$ in Theorem 7.3.4. This lemma is easy to generalize to arbitrary alphabets:

(7.4.1) **Lemma** *Let $x \in \Sigma_0^*$ with $|x| = n$ and let p_h denote the relative frequency of the letter h in the word x. Let $p = (p_h : h \in \Sigma_0)$. Then*

$$\mathbf{K}(x) \le \frac{H(p)}{\log m} n + O(m \log n).$$

We mention another interesting connection between entropy and complexity: the entropy of a computable probability distribution over all strings is close to the average complexity. This is stated by the following reformulation of Corollary 7.2.5:

(7.4.2) **Theorem** *Let p be a computable probability distribution over the set Σ_0^*. Then*

$$|H(p) - \sum_x p(x)\mathbf{H}(x)| = O(1).$$

7.5 Kolmogorov complexity and coding

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

$$\eta(n) = \max\{\, |f(x)| : x \in \mathcal{L},\ |x| \le n \,\}.$$

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,

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

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 $\log_2 t \le \log_2 |\mathcal{L}_n|$ bits, plus a constant number of additional bits (the program for taking the elements of Σ^* in lexicographic order, checking their membership in $\mathcal{L}$ and printing the t-th one).

We arrive at more interesting questions if we stipulate that the code from the word and, conversely, the word from the code should be polynomially computable. In other words: we are looking for a language $\mathcal{L}'$ and two polynomially computable functions:

$$f : \mathcal{L} \to \mathcal{L}', \qquad g : \mathcal{L}' \to \mathcal{L}$$

with $g \circ f = \mathrm{id}_{\mathcal{L}}$ for which, for every x in $\mathcal{L}$ the code $|f(x)|$ is "short" compared to $|x|$. Such a pair of functions is called a **polynomial-time code**. (Instead of the polynomial time bound we could, of course, consider other complexity restrictions.)

We present some examples when a polynomial-time code approaches the information-theoretical bound.

(7.5.1) **Example** In the proof of Lemma 7.3.4, 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 $\{n-1, n-2, \dots, 0\}$. Each such set can be written as $\{a_1, \dots, a_m\}$ with $a_1 > a_2 > \cdots > a_m$. Then the set $\{b_1, \dots, b_m\}$ precedes the set $\{a_1, \dots, a_m\}$ 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 $\{a_1, \dots, a_m\}$, be the lexicographically t-th set. Then the number of subsets $\{b_1, \dots, b_n\}$ with this property is exactly $\binom{a_i}{m-i+1}$. Summing this for all i we find that

$$t = 1 + \binom{a_1}{m} + \binom{a_2}{m-1} + \cdots + \binom{a_m}{1}. \tag{7.5.2}$$

For fixed m, this formula 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 $\binom{a_1}{m} \le t - 1$, then the greatest number a_2 with

$\binom{a_2}{m-1} \le t-1-\binom{a_1}{m}$, etc. We do this for m steps. The numbers obtained this way satisfy $a_1 > a_2 \cdots$; indeed, e.g. 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 comes out similarly that $a_m \ge 0$ and that there is no "remainder" after m steps, i.e., that 7.5.2 holds. It can therefore be found out in polynomial time which subset is lexicographically the t-th. $\diamondsuit$

(7.5.3) **Example** Consider the trees, given by their adjacency matrix (but 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 points i, j are connected in the first one they are also connected in the second one and vice versa (so, if we renumber the points 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 applies here:

(7.5.4) **Theorem** *[Cayley's Theorem] The number of n-point labeled trees is n^{n-2}.*

Consequently, according to the information-theoretical lower bound, with any encoding, an n-point tree needs a code with length at least $\lceil \log(n^{n-2}) \rceil = \lceil (n-2)\log n \rceil$. Let us investigate whether this lower bound can be achieved by a polynomial-time code.

(a) If we code the trees by their adjacency matrix this is 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 endpoints. 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 point 0, and we specify the tree by the sequence $(\alpha(1), \ldots, \alpha(n-1))$ in which $\alpha(i)$ is the first inside point on the path from node i to the root (the "father" of i). This is $(n-1)\lceil \log_2 n \rceil$ bits, which is already nearly optimal.

(d) There is, however, also a procedure, the so-called Prüfer code, that sets up a bijection between the n-point labeled trees and the sequences of length $n-2$ of the numbers $0, \ldots, n-1$. (Therewith 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-point 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 endpoint (leaf) of the tree; if $i_1, \ldots, i_k$ are already defined then let i_{k+1} be the smallest endpoint of the graph remaining after deleting the points $i_1, \ldots, i_k$. (We do not consider the root 0 an endpoint.) 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 point 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.

(7.5.5) **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 namely we know already the edges of the tree (the pairs $[i, \alpha(i)]$). The point i_1 is the smallest endpoint of the tree, for its determination it is therefore enough to figure out the endpoints from the Prüfer code. But this is obvious: the endpoints are exactly those that are not the "fathers" of other points, i.e., the ones that do not occur among the numbers $\alpha(i_1), \ldots, \alpha(i_{n-1}), 0$. The point i_1 is therefore uniquely determined.

Assume that we know already that the Prüfer code uniquely determines $i_1, \ldots, i_{k-1}$. It obtains similarly to the above reasoning 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.

(7.5.6) **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 above proof, 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 points $1, \ldots, n$. 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 with the point 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. $\diamondsuit$

(7.5.7) **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. $\diamondsuit$

(7.5.8) **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 $n^{n-2}/n! \le 2^{n-2}$. (According to a difficult result of George Pólya, the number of n-point 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. Take an n-point tree F. Walk through F by the "depth-first search" rule: Let x_0 be the point labeled 0 and define the points $x_1, x_2, \ldots$ 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 points step-for-step. In step $(i+1)$, if $\varepsilon_i = 1$ then we take a new point (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. $\diamondsuit$

(7.5.9) **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 point x_i from the point 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 the known combinatorial theorem, $\frac{1}{n}\binom{2n-2}{n-1}$. 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 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.

8.1 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

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

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.

8.1 **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?

$\diamondsuit$

8.2 **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. $\diamondsuit$

8.3 **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.

$\diamondsuit$

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.

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

(8.1.2) **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 $2pz+p+i-2^{k-1}, \ldots, 2pz+p+i-(2^k-1)$. 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 $j \le i$ and their subordinates have written 1 into each of the 2^{k-1} positions downwards starting with $2pz+p+i$ that are still $\ge 2pz+p$. If a supervisor processor or its subordinates started to the right from them and reaches a cell $\le i$ 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.

(8.1.3) **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).

8.2 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 8.1.1, 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.

(8.2.1) **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.

(8.2.2) **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.

(8.2.3) **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.

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

(8.2.4) **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 ($k \le n$) 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 4.1.3). ∎

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

(8.2.5) **Csánky's Theorem** *The determinant of an arbitrary integer matrix can be computed by an* NC *algorithm. Consequently, the invertible matrices form an* NC*-language.*

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 6.1.3 with random substitutions, we arrive at the following important application:

(8.2.6) **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).

8.5 **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. ♢

9 Decision trees

The logical framework of a lot of algorithms can be described by a tree: we start from the root and in every branching point, the result of a certain "test" determines which way we continue. E.g., most sorting algorithms make sometimes 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 necessary information about the input, i.e., when we arrive at an endpoint all that is left is to read the output from the endpoint. The complexity of the tree gives some information about the complexity of the algorithm; the depth of the tree (the number of edges in the longest path leaving the root) tells, e.g., how many tests must be performed in the worst case during the computation. We can describe, of course, every algorithm bya trivial tree of depth 1 (the test performed in the root is the computation of the end result); this algorithm scheme makes sense therefore 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 if we assume that information on the input is available only by the permissible tests (e.g. in sorting, we can only compare the given numbers with each other and cannot perform e.g. arithmetic operations on them).

9.1 Algorithms using decision trees

Consider some simple examples.

9.1.1 Finding a false coin with a one-armed scale

Given are n coins looking outwardly identical. 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 already 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 endpoints to the 1-element sets.) For every branching point 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.

9.1.2 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 branching point 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 endpoints to the one-element sets.) For each branching point 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.

9.1 **Exercise** Prove that fewer measurements do not suffice in either problem 9.1.1 or problem 9.1.2. $\diamondsuit$

9.1.3 Sorting

Given are n elements that are ordered in some way (unknown for 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 endpoint corresponds to a complete ordering.

(9.1.1) **Remark** In the above sorting algorithm, we only counted the comparisons. With a real program, 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 algorithm requiring altogether only $O(n \log n)$ steps. $\diamondsuit$

9.1.4 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 $p_{j_0}, \ldots, p_{j_a}, p_n, p_{j_b}, p_{j_m}$.

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.

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.)

9.2 **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. $\diamondsuit$

*(9.3) **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. $\diamondsuit$

To formalize the notion of a decision tree let us be given the set A of possible inputs, the set B of possible outputs and a set Φ of functions defined on A with values in $\{1, \ldots, d\}$, the **allowed test-functions**. A **decision tree** is a rooted tree whose internal points (including the root) have d children (the tree is d-regular), its endpoints are labelled with the elements of B, the other points with the functions of Φ. 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 an endpoint as follows. If we are in an internal point 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 an endpoint w; the value of $f(a)$ is the label of w.

The question is that for a given function f, what is the shallowest decision tree 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 (2-regular), the branching points are indexed with the variables, the endpoints with 0 and 1. Such is the yes-no question of whether there is an absolute winner, in a competition by elimination. Notice that the decision tree concerning sorting is not such: 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.

(9.1.2) **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 following simple decision tree. (Here, the root is on the left, the leaves on the right and the levels of the tree are indicated by indentation.)

- x_2
    - x_3
        - 1
        - x_4
            - 1
            - 0
    - x_3
        - x_1
            - 1
            - 0
        - 0

Therefore $D(f) \leq 3$. It is easy to see that $D(f) = 3$. $\diamondsuit$

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 Φ. How many questions do suffice for her to compute the answer? Yvette's strategy corresponds to a decision tree, and Xavier plays optimally if with his answers, he drives Yvette to the endpoint 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.)

9.2 Nondeterministic decision trees

The idea learned in Section 5, **nondeterminism**, helps in other complexity-theoretic investigations, too. In the decision-tree model, this 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 non-deterministic polynomial classes, P and 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 can 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.

(9.2.1) **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 if not (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 connectedness or unconnectedness 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 $n-1$-arm 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 subsection (exercise 9.8). $\diamondsuit$

(9.2.2) **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:

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

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

$$D_0(f) \ge n - \alpha$$

where α is the maximum number of independent points in the graph. It can also be proved (see Theorem 9.3.7) 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:

(9.2.3) **Theorem**
$$D(f) \le D_0(f)D_1(f).$$

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

Let (say) $f(0,\dots,0) = 0$; then $k \le D_0(f)$ 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 tree: we ask the value of the first k variables, let the obtained answers be $a_1,\dots,a_k$. Fixing these, we obtain a Boolean function

$$g(x_{k+1},\dots,x_n) = f(a_1,\dots,a_k,x_{k+1},\dots,x_n).$$

Obviously, $D_0(g) \le D_0(f)$ and $D_1(g) \le D_1(f)$. We claim that the latter inequality can be strenghened:

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

Consider an input $(a_{k+1},\dots,a_n)$ of g with $g(a_{k+1},\dots,a_n) = 1$. Together with the bits $a_1,\dots,a_k$, this gives an input of the Boolean function f for which $f(a_1,\dots,a_n) = 1$. According to the definition of the quantity $D_1(f)$, one can choose $m \le D_1(f)$ 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, $f(0, \ldots, 0, a_{k+1}, \ldots, a_n)$ would have to be 0 (due to the fixing of the first k variables) but would also have to be 1 (due to the fixing of $x_{i_1}, \ldots, x_{i_m}$), which is a contradiction. Thus, in the function g, at the position $a_{k+1}, \ldots, a_k$, only $m-1$ variables must be fixed to obtain the identically 1 function. From this, the claim follows.

From the inductive assumption,

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

and hence

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

∎

In Example 9.2.2, we could define the function by a disjunctive 2-normal form and $D_1(f) = 2$ was true. This is not an accidental coincidence:

(9.2.4) **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 $(a_1, \ldots, a_n)$ be an input for which the value of the function is 1. Then there is an elementary conjunction in the disjunctive normal form whose value is 1. If we fix the variables occurring in this conjunction then the value of the function will be 1 independently of the values of the other variables. ∎

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

(9.2.5) **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 9.2.4, it is sufficient to see that if $D_1(f) = k$ then f is expressible by a disjunctive k-normalform. Let $\{x_{i_1}, \ldots, x_{i_m}\}$ be a subset of the variables minimal with respect to containment, that can be fixed in such a way as to make the obtained function is identically 1. (Such a function is called a **mintag**.) 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 $x_{i_1}, \ldots, x_{i_m}$ and 0 to the others. According to the foregoing, the value of the function is 1. By the definition of the quantity $D_1(f)$, we can fix in this assignment k values in such a way as to make the 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 $x_{i_1}, \ldots, x_{i_m}$. But then due to the minimality of the set $\{x_{i_1}, \ldots, x_{i_m}\}$, we had to fix all of them, and hence $m \le k$.

Let us prepare for every mintag 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. ∎

9.4 **Exercise** Give an example showing that in Proposition 9.2.5, the condition of monotonicity cannot be omitted. $\diamondsuit$

9.3 Lower bounds on the depth of decision trees

We mentioned that decision trees as computation models have the merit that non-trivial lower bounds can be given for their depth. First we mention, however, a non-trivial lower bound also called **information-theoretic estimate**.

(9.3.1) **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 endpoints. Since every element of the range of f must occur as a label of an endpoint it follows that $t \geq d^h$. $\blacksquare$

For 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:

$$\varphi_{ij}(a_1, \ldots, a_n) = \begin{cases} 1 & \text{if } a_i < a_j, \text{ and} \\ 0 & \text{if } a_i > a_j. \end{cases}$$

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.

(9.3.2) **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 9.3.1 gives only $\log n$ for lower bound; but we can apply Lemma 9.3.2 as follows. Let $a = (a_1, \ldots, a_n)$ be an arbitrary permutation, and consider $k < n-1$ comparison tests. The pairs (i, j) for which a_i and a_j will be compared form a graph G over the underlying set $\{1, \ldots, n\}$. Since it has fewer than $n-1$ edges this graph falls into two disconnected parts, G_1 and G_2. Without loss of generality, let G_1 contain the maximal element and let p denote its number of vertices. Let $a' = (a'_1, \ldots a'_n)$ 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.

9.5 **Exercise** Show that to pick the middle one by magnitude among $2n + 1$ elements,

(a) at least $2n$ comparisons are needed;

*(b) $O(n)$ comparisons suffice.

$\diamondsuit$

In what follows we show estimates for the depth of some more special decision trees, applying, however, some more interesting methods. First we mention a result of Best, Schrijver and Van Emde Boas, then one of Rivest and Vuillemin which gives a lower bound of unusual character for the depth of decision trees.

(9.3.3) **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 leave of this tree. Here, $m \le d$ variables are fixed, therefore there are at least 2^{n-m} inputs leading to this leaf. All of these correspond to the same function value, therefore the number of inputs leading to this leaf and giving the function value "1" is given by 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 $k \ge n-d$. ∎

By an appropriate extension of the above proof, the following generalization of Theorem 9.3.3 can be proved.

(9.3.4) **Theorem** *For a given n-variable Boolean function f, let us 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 $(x_1,\ldots,x_n) \in \{0,1\}^n$. If f can be computed by a decision tree of depth d then $\Psi_f(t)$ is divisible by $(t+1)^{n-d}$.*

9.6 **Exercise** Prove Theorem 9.3.4. $\diamondsuit$

We call a Boolean function f of n variables **laconic** if it cannot be computed by a decision tree of length smaller than n. It follows from Theorem 9.3.3 that *if a Boolean function has an odd number of substitutions making it "1" then the function is laconic.*

We obtain another important class of laconic 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 resp. 1.

(9.3.5) **Proposition** *Every non-constant symmetric Boolean function is laconic.*

Proof Let $f : \{0,1\}^n \to \{0,1\}$ be the Boolean function in question. Since f is not constant, there is a j with $1 \le j \le n$ such that if $j-1$ variables have value 1 then the function's value is 0 but if j variables are 1 then the function's value is 1 (or conversely).

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:

(9.3.6) **Conjecture** *If a non-constant monotonic Boolean function is weakly symmetric then it is laconic.*

We show by an application of Theorem 9.3.4 that this conjecture is true in an important special case.

(9.3.7) **Theorem** *If a non-constant monotonic Boolean function is weakly symmetric and the number of its variables is a prime number then it is laconic.*

Proof It is enough to show that $\Psi_f(n-1)$ is not divisible by n. First of all, we use the group-theoretical result that (with a suitable indexing of the variables) the substitution $x_1 \to x_2 \to \cdots \to x_n \to x_1$ does not change the value of the function. It follows that in the definition of $\Psi_f(n-1)$, if in some term, not all the values $x_1, \ldots, x_n$ are identical then n 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 monotonic, it follows that $f(0,\ldots,0) = 0$ and $f(1,\ldots,1) = 1$, from which it can be seen that $\Psi_f(n-1)$ gives remainder $(-1)^n$ modulo n. ∎

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 **monotonic** 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 applied, in its original form, to graph properties:

(9.3.8) **Conjecture** *Every non-trivial monotonic graph property is laconic, 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).

9.7 **Exercise** Prove that the connectedness of a graph is a laconic property. ♢

9.8 **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 laconic.

*(c) This statement holds also for odd n.

♢

9.9 **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 laconic. ♢

Among the more complex decision trees, the **algebraic decision trees** are important. In this case, the input is n real numbers $x_1, \dots, x_n$ and every test function is described by a polynomial; in the branching points, we can go in three directions according to whether the value of the polynomial is negative, 0 or positive (sometime, we distinguish only two of these and the tree branches only in two). An example is provided for the use of such a decision tree by sorting, where the input can be considered n real numbers and the test functions are given by the polynomials $x_i - x_j$.

A less trivial example is the determination of the convex hull of n planar points. Remember that the input here is $2n$ real numbers (the coordinates of the points), and the test functions are represented either by the comparison of two coordinates or by the determination of the orientation of a triangle. The points $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) form a triangle with positive orientation if and only if

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} > 0.$$

This can be considered therefore the determination of the sign of a second-degree polynomial. The algorithm described in Subsection 9.1 gives thus an algebraic decision tree in which the test functions are given by polynomials of degree at most two and whose depth is $O(n \log n)$.

The following theorem of Ben-Or provides a general lower bound on the depth of algebraic decision trees. Before the formulation of the theorem, we introduce an elementary topological notion. Let $U \subset \mathbf{R}^n$ be a set in the n-dimensional space. Two points x_1, x_2 of the set U are called **equivalent** if there is no decomposition $U = U_1 \cup U_2$ for which $x_i \in U_i$ and the closure of U_1 is disjoint from the clusure of U_2. The equivalence classes of this equivalence relation are called the **components** of U. We call a set **connected** if it has only a single connected component.

(9.3.9) **Theorem** *Suppose that the set $U \subset \mathbf{R}^n$ has at least N connected components. Then every algebraic decision tree deciding $x \in U$ whose test functions are polynomials of degree at most d, has depth at least $\log N / \log(6d) - n$. If $d = 1$ then the depth of every such decision tree is at least $\log_3 N$.*

Proof We give the proof first for the case $d = 1$. Consider an algebraic decision tree of depth h. This has at most 3^h endpoints. Consider an endpoint reaching the conclusion $x \in U$. Let the results of the tests on the path leading here be, say,

$$f_1(x) = 0, \quad \ldots, \quad f_j(x) = 0, \quad f_{j+1}(x) > 0, \quad \ldots, \quad f_h(x) > 0.$$

Let us denote the set of solutions of this set of equations and inequalities by K. Then every input $x \in K$ leads to the same endpoint and therefore we have $K \subset U$. Since every test function f_i is linear, the set K is convex and is therefore connected. Therefore K is contained in a single connected component of the set U. It follows that the inputs belonging to different components of U lead to different endpoints of the tree. Therefore $N \le 3^h$, which proves the statement referring to the case $f = 1$.

In the general case, the proof must be modified in that K is not necessarily convex and so not necessarily connected either. Instead, we can use an important result from algebraic geometry (a theorem of Milnor and Thom) implying that the number of connected components of K is at most $(2d)^{n+h}$. From this, it follows similarly to the first part that

$$N \ge 3^h (2d)^{n+h} \ge (6d)^{n+h},$$

which implies the statement of the theorem. ∎

For an application, consider the following problem: given n real numbers $x_1, \ldots, x_n$; let us decide whether they are all different. We consider an elementary step the comparison of two given numbers, x_i and x_j. This can have three outcomes: $x_i < x_j$, $x_i = x_j$ and $x_i > x_j$. What is the decision tree with the smallest depth solving this problem?

It is very simple to give a decision tree of depth $n \log n$. Let us namely apply an arbitrary sorting algorithm to the given elements. If anytime during this, two compared elements are found to be equal then we can stop since we know the answer. If not then after $n \log n$ steps, we can order the elements completely, and thus they are all different.

Let us convince ourselves that $\Omega(n \log n)$ comparisons are indeed needed. Consider the following set:

$$U = \{ (x_1, \ldots, x_n) : x_1, \ldots, x_n \text{ are all different } \}.$$

This set has exactly $n!$ connected components (two n-tuples belong to the same component if they are ordered in the same way). So, according to Theorem 9.3.9, every algebraic decision tree deciding $x \in U$ in which the test functions are linear, has depth at least $\log_3(n!) = \Omega(n \log n)$. The theorem also shows that we cannot gain an order of magnitude with respect to this even if we permitted quadratic or other bounded-degree polynomials as test polynomials.

We have seen that the convex hull of n planar points in general position can be determined by an algebraic decision tree of depth $n \log n$ in which the test polynomials have degree at most two. Since the problem of sorting can be reduced to the problem of determining the convex hull it follows that this is essentially optimal.

9.10 **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.

$\diamondsuit$

9.11 **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. $\diamondsuit$

10 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 ε_1 to the other one; then one of them (maybe the other one, maybe the same one) sends a bit ε_2, and so on. At the end, both processors must "know" the bit to be computed.

To make it more graphic, instead of the two processors, we will speak of two **players**, Alice and Bob. Imagine that Alice 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.

10.1 Communication matrix and protocol-tree

Let Alice's possible inputs be $a_1, \ldots, a_n$ and Bob's possible inputs $b_1, \ldots, b_m$ (since the local computation is free it is indifferent for us how these are coded). Let c_{ij} be the value to be computed for inputs a_i and b_j. The matrix $C = (c_{ij})_{i=1}^{n}{}_{j=1}^{m}$ is called the **communication matrix** of the problem in question. This matrix completely describes the problem: both players know the whole matrix C. Alice knows the index i of a row of C, while Bob knows the index j of a column of C. Their task is to determine the element c_{ij}. The trivial protocol is that e.g. Alice sends Bob the number i; this means $\lceil \log n \rceil$ bits. (If $m < n$ then it is better, of course, to proceed the other way.)

Let us see first what a protocol means for this matrix. First of all, the protocol must determine who starts. Suppose that Alice sends first a bit ε_1. This bit must be determined by the index i known to Alice; in other words, the rows of C must be divided in two parts according to $\varepsilon_1 = 0$ or 1. The matrix C is thus decomposed into two submatrices, C_0 and C_1. This decomposition is determined by the protocol, therefore both players know it. Alice's message determines which one of C_0 and C_1 contains her row. From now on therefore the problem has been narrowed down to the corresponding smaller matrix.

The next message decomposes C_0 and C_1. If the sender is Bob then he divides the columns into two classes; if it is Alice then she divides the rows again. It is not important that the second message 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.

(10.1.1) **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. ∎

(10.1.2) **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.

(10.1.3) **Example** Both Alice and Bob know some 0-1 sequence of length n; they want to decide whether the two sequences are equal. $\diamondsuit$

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:

(10.1.4) **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.

(10.1.5) **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 Γ is the set of states of the machine). At the end, the Turing machine writes the answer into cell 0 and Alice will know whether the word is a palindrome. For the price of 1 bit, she can let Bob also know this.

According to Theorem 10.1.4, 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 α of length $2h$ and write in front of it a sequence β of length $n-h$ and behind it a sequence γ of length $n-h$. The sequence obtained this way is a palindrome if and only if $\beta = \gamma^{-1}$ where we denoted by γ^{-1} the inversion of γ. By Theorem 10.1.4 and the above reasoning, for every α there are at most $2^{n/2}$ strings β for which on input $\beta\alpha\beta^{-1}$, the head passes between cells $n-h$ and $n-h+1$ fewer than cn times. Since the number of α's is 2^h the assertion follows.

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

10.1 **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.

$\diamondsuit$

10.2 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.

(10.2.1) **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. $\diamondsuit$

10.2 **Exercise** Prove that in Example 10.2.1, any protocol requires at least $\log n$ bits. $\diamondsuit$

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

(10.2.2) **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 $\lceil\log n\rceil(2+\lceil\log n\rceil)/2$. $\diamondsuit$

10.3 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" $x_1,\dots,x_n\in\{0,1\}^*$ 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 $\kappa_{ND}(C)$

(10.3.1) **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.) $\diamondsuit$

(10.3.2) **Example** In Example 10.1.3, 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:

(10.3.3) **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 10.3.2, the matrix C is the $2^n\times 2^n$ unit matrix. Obviously, only the 1×1 submatrices of this are all-1, the covering of the 1's requires therefore 2^n such submatrices. Thus, the non-deterministic complexity of this problem is also n.

Let $\kappa(C)=s$. Then C can be decomposed into 2^s submatrices half of which are all-0 and half are all-1. According to Lemma 10.3.3 the nondeterministic communication complexity of C is therefore at most $s-1$. Hence

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

Example 10.3.2 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 10.3.2 also shows that $\kappa_{ND}(C)$ and $\kappa_{ND}(\overline{C})$ can be very different. On the basis of the previous remarks, we have

$$\max\{1+\kappa_{ND}(C),\ 1+\kappa_{ND}(\overline{C})\}\le\kappa(C).$$

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.

(10.3.4) **Theorem**

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

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,

$$\varrho(C) \le \mathrm{rk}(C),$$

and Lemma 10.3.3 implies

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

The following inequality therefore implies theorem 10.3.4.

(10.3.5) **Theorem**

$$\kappa(C) \le 1 + \log \varrho(C)(\kappa_{ND}(\overline{C}) + 2).$$

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

(10.3.6) **Claim**

$$\varrho(A_i) + \varrho(B_i) \le \varrho(C).$$

10.4 **Exercise** Prove this claim. $\diamondsuit$

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 $\varrho(A_i) \le \frac{1}{2}\varrho(C)$. 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 $\varrho(B_i) \le \frac{1}{2}\varrho(C)$. 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 $\varrho(C') \le \frac{1}{2}\varrho(C)$. 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 10.1.1):

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

10.5 **Exercise** Show that in Theorems 10.3.4 and 10.3.5 and in Corollary 10.3.7, with more careful planning, the factor $(2 + \kappa_{ND})$ can be replaced with $(1 + \kappa_{ND})$. ♢

To show the power of Theorems 10.3.4 and 10.3.5 consider the examples treated in Subsection 10.2. If C is the matrix corresponding to Example 10.2.1 (in which 1 means that the subtrees are disjoint) then $\kappa_{ND}(\overline{C}) \le \lceil \log n \rceil$ (it is sufficient to name a common vertex). It is also easy to obtain that $\kappa_{ND}(C) \le 1 + \lceil \log(n-1) \rceil$ (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 10.3.4 and 10.3.5 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 10.2.

Let now C be the matrix corresponding to Example 10.2.2. It is again true that $\kappa_{ND}(\overline{C}) \le \lceil \log n \rceil$, 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 10.3.5, 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 $\kappa_{ND} \le \kappa$.

(10.3.8) **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 $\mathcal{H} \in \text{co}-\mathrm{NP}^{\mathrm{comm}}$ if the matrix set $\{\bar{\ }{:}C \in \mathcal{H}\}$ is in $\mathrm{NP}^{\mathrm{comm}}$. Then Example 10.3.2 shows that

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

and Theorem 10.3.4 implies

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

♢

10.4 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.

(10.4.1) **Protocol** Alice chooses a random prime number p in the interval $1 \le p \le N$ 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 $x \ne y$. If yes then he concludes that $x = y$. $\diamondsuit$

First we note that this protocol uses only $2 \log N$ bits since $1 \le r \le p \le N$. 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 $x \ne y$ 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

$$d \ge p_1 \cdots p_k \ge 2 \cdot 3 \cdot 5 \cdot \cdots \cdot q,$$

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 $k \le \pi(n)$ (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:

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

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.

(10.4.2) **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. $\diamondsuit$

11 An application of complexity: cryptography

The complexity of a phenomenon can be the main obstacle of finding out about it. Our book—we hope—proves that complexity is not only an obstacle to research but also its important and exciting subject. It goes, however, beyond this: it has applications in which we exploit the complexity of a phenomenon. In the present section, we treat such a subject: **cryptography**, i.e. the science of secret codes. We will see that, precisely by the application of the results of complexity theory, secret codes go beyond the well-known (military, intelligence) applications and find a number of uses in civil life.

11.1 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).

(11.1.1) **Remark** This, so-called "one-time pad" method is very safe. It was used e.g. 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; moreover, it may be possible to agree on a key even without actually passing it. $\diamondsuit$

11.2 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 e.g. withdraw money using an automaton. 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 even written on his card), so if he takes care that nobody else can find it out 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 in the knowledge of the complete password-checking program? This seemingly self-contradictory requirement is satisfiable!

Solution: the client takes up n points 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 graph to the bank (without marking the Hamiltonian circuit in it).

If somebody shows up at the bank in the name of the client the bank checks about the given password whether it is a Hamiltonian circuit of the graph in store there. If yes the password will be accepted; if not, it will be rejected. It can be seen that even if somebody learns the graph without authorization she must still solve the problem of finding a Hamiltonian circuit in a graph. An this is NP-hard!

(11.2.1) **Remarks**

1. Instead of the Hamiltonian circuit problem, we could have based the system, of course, 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 $m = n \log n$ seems at least hard.

♢

12 Public-key cryptography

In this subsection, we describe a system that improves on the methods of classical cryptography in several points. Let us note first of all that the system wishes to serve primarily civil rather than military goals. For using electronic mail, 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 know only by her. There is further a generally known coding/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 have

$$f(f(x, e_i), d_i) = f(f(x, d_i), e_i) = x. \tag{12.0.1}$$

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 work, it must satisfy the following complexity conditions.

(C1) $f(x, e_i)$ can be computed efficiently from x and e_i.

(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 understand in what follows polynomial time but the system makes sense also under other resource-bounds. A function with the above properties will be called a **trapdoor function**.

Condition (C1) makes sure that if participant i sends a message to participant j then she can compute it in polynomial time and the addressee can also solve 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.

(12.0.2) *Only j can solve a message addressed to j.*

Proof Assume that a band $k_1, \ldots, k_r$ of unauthorized participants finds the message $f(f(x, d_i), e_j)$ (possibly, even who sent it to whom) and 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 also compute x by the formula $x = f(z, d_i)$, which contradicts condition (C2). ∎

The following can be verified by similar reasoning:

(12.0.3) *Nobody can forge a message in the name of i, i.e. participant j can be sure that the message could have been sent only by i.*

(12.0.4) *j can prove to a third person (e.g. 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.*

(12.0.5) *j cannot change the message (and have it accepted e.g. in a court as coming from i) or send it in i's name to somebody else.*

It is not at all clear, of course, whether a trapdoor function exists. By now, it has been possible to give such systems only under certain number-theoretic complexity conditions. (Some of the proposed systems turned out later to be insecure—the corresponding complexity conditions were not true.) In the next subsection, we present a system that, to our current knowledge, is secure.

12.1 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 $1 \le e_i < m$ that is relatively prime to $(p-1)$ and $(q-1)$. (It can do this by generating a random e_i between 0 and $(p-1)(q-1)$ and checking by the Euclidean algorithm whether it is relatively prime to $(p-1)(q-1)$. If it is not it tries a new number. It is easy to see that after $\log n$ repetitions, it finds a good number e_i with high probability.) Then, using the Euclidean algorithm, it finds a number d_i with $1 \le d_i < m$ such that

$$e_i d_i \equiv 1 \pmod{(p-1)(q-1)}.$$

(here $(p-1)(q-1) = \varphi(m)$, the number of positive integers smaller than m and relatively prime to it). The public key is the number e_i, the secret key is the number d_i. The message x itself is considered a natural number with $0 \le x < m$ (if it is longer then it will be cut into pieces). The encoding function is defined by the formula

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

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

The inverse relation between coding and decoding (formula 12.0.1) 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 Subsection 4.1. Condition (C2) holds only in the following, weaker form:

(B2') $f(x,d_i)$ 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$\neq$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 solve 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 get information. The main problem is the following: *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 $s e_k = tu + 1$. Then

$$z^s \equiv y^{s e_k} = y(y^u)^t \equiv y \pmod{m}$$

and hence

$$x \equiv y^{e_i} \equiv z^{e_i s}.$$

Thus, participant j can also compute x.

12.1 **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. $\diamondsuit$

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.

12.2 Pseudo-randomness

A long sequence generated by a short program is not random, according to the notion of randomness introduced using information complexity. For various reasons, we are still forced to use algorithms that generate random-looking sequences but, as one of the first mathematicians to recommend the use of these, Von Neumann 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.

Why cannot we use real random sequences instead of computer-generated ones? In most cases, the reason is convenience: our computers are very fast and we do not have a cheap physical device giving the equivalent of unbiased coin-tosses at this rate. There are, however, other reasons. We often want to repeat some computation for various purposes, including error checking. In this case, if our source of random numbers was a real one then the only way to use the same random numbers again is to store them, using a lot of space. The most important reason is that there are applications, in cryptography, where what we want is only that the sequence should "look random" to somebody who does not know how it was generated.

(12.2.1) **Example** For the purpose of generating sequences that just "look random", the following method has been used extensively: it is called a *linear congruential generator*. It uses a "seed" consisting of the integers a, b, m and x_0 and the pseudo-random numbers are generated by the recurrence $x_{i+1} = ax_i + b \bmod m$. This sequence is, of course, periodic, but it will still be random-looking if the period is long enough (comparable to the size of the modulus).

Knuth's book "Seminumerical Algorithms" studies the periods of these generators extensively. It turns out that some simple criteria will guarantee long periods. These generators fail, however, the more stringent test that they should not be predictable by someone who does not know the seed. New methods of number theory (e.g. the basis reduction algorithm of Lenstra, Lenstra and Lovász for lattices, [8]) gave a polynomial algorithm always for the prediction of such sequences. This does not mean that linear congruential generators became useless, but their use should be restricted to applications which are unlikely to "know" about this (sophisticated) prediction algorithm. $\diamondsuit$

(12.2.2) **Example** The following random number generator was suggested by L. BLUM, M. BLUM and M. SHUB. Find random prime numbers p, q of the form $4k - 1$ and form $m = pq$. Choose them big enough so that the factoring of n by somebody not knowing p, q should seem a hopeless task. Pick also a random positive integer $x < m$; the pair (x, n) is the seed. Let $x_0 = x$, $x_{i+1} = x_i^2 \bmod n$ for $i > 0$. Let $y_i = x_i \bmod 2$ be the i-th pseudorandom bit.

Thus, we choose a modulus n that seems hard to factor. Creating random numbers happens by repeatedly squaring a certain number x_i modulo n. The last bits $y_i = x_i \bmod 2$ of the numbers x_i are hoped to look random. Later in this section, we will see that this method produces bits that remain random-looking even to a user familiar with all currently known algorithms. $\diamondsuit$

In general, a pseudo-random bit generator transforms a small, truly random seed into a longer sequence that still looks random from certain points of view. Let $G_n(s)$ be a function, our intended random-number generator. Its inputs are *seeds* s of length n, and its outputs are sequences $x = G_n(s)$ of length $N = n^k$ for some fixed k. The success of using x in place of a random sequence depends on how severely the randomness of x is tested by the application. If the application has the ability to test all possible seeds of length n that might have generated x then it finds the 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 x from truly random strings. It turns out that it is enough to require that every bit of x should be highly unpredictable from the other bits, as long as the prediction algorithm cannot use too much time.

Let $F_N(i, y)$ be some algorithm (we even permit it to be randomized) that takes a string y of length $N - 1$ and a natural number $i \le N$ and outputs a bit. Now, if x is truly random then the probability that

$$x_i = F_N(i, x_1, \dots, x_{i-1}, x_{i+1} \dots, x_N)$$

(the success probability) is exactly $1/2$. We will call G_n a *pseudo-random bit generator* if, assuming $x = G_n(s)$ where s is a truly random seed, for any constant c, for all polynomially computable functions F, for sufficiently large n, the success probability is still less than $1/2 + n^{-c}$.

This requirement is so strong that it is still unknown whether pseudo-random bit generators exist. (If they do then P $\neq$ NP, as we will see shortly.)

Other, similarly reasonable definitions of the notion of pseudo-number bit generator were found to be equivalent to this one. The following theorem shows that from the point of view of an application computing a yes-no answer, pseudo-random bit generators do not differ too much from random bit generators.

(12.2.3) **Yao's Theorem** *Let $G_n(s)$ be a generator. Let $T(x)$ be polynomial-time algorithm computing a 0 or 1 from strings of length N. Let $u(n)$ be the probability that $T(y)$ produces a 1 (accepts) when the string y is truly random, and $v(n)$ the probability that $T(x)$ produces 1 when $x = G_n(s)$ with a truly random seed s. Suppose that for some constant c, for some large n, we have $|u(n) - v(n)| > n^{-c}$. Then a polynomial-time algorithm will find some i and guess x_i from $x_1, \dots, x_{i-1}$ with probability of success $\ge 1/2 + 2n^{-c}/N$.*

Proof Assume that, contrary to the statement of the theorem, there is a constant c such that for infinitely many n_j we have $|u(n_j) - v(n_j)| > n_j^{-c}$. Let n be such an n_j. Without loss of generality, let us assume that $u(n) - v(n) > n^{-c}$ (otherwise, we consider the negation of B). For an arbitrary sequence $x_1, \dots, x_N$, we use the notation

$$x_i^j = x_i, x_{i+1}, \dots, x_j.$$

Let $X_1, \dots, X_N$ be the random variables that are the bits of $G_n(s)$ where s is a random seed. Let $R_1, \dots, R_N$ be new random bits chosen by an independent coin-tossing. We will consider a gradual transition $X_1^i R_{i+1}^N$ from one of these two sequences into the other one, taking the first i bits from $X_1, \dots, X_N$ and the rest from $R_1, \dots, R_N$. Let p_i be the probability of acceptance by T for $X_1^i R_{i+1}^N$. Then $p_0 = u(n)$ and $p_N = v(n)$. According to our assumption, $p_N - p_0 > n^{-c}$. Then, $p_i - p_{i-1} > n^{-c}/N$ for some i. Let us fix this i.

Let the prediction algorithm of X_i from $X_1, \dots, X_{i-1}$ work as follows. First it chooses the random bits $R_i, \dots, R_N$. Then it computes $T(X_1^{i-1} R_i^N)$. It predicts $X_i = R_i$ iff this value is 1.

For $b = 0, 1$, $a = a_1, \dots, a_N$, let $P_b(a)$ be the probability of acceptance of $a_1^{i-1} b R_{i+1}^N$. Let

$$q_1(a) = \operatorname{Prob}\{X_i = 1 \mid X_1^{i-1} = a_1^{i-1}\}$$

be the conditional probability that $X_i = b$ provided the first $i - 1$ pseudo-random bits were $a_1, \dots, a_{i-1}$. Then the success probability of our prediction is the expected value of the following expression (we delete the argument a from $P_b(a), q_b(a)$):

$$\begin{aligned} &\operatorname{Prob}\{R_i = 1\}\big(P_1 q_1 + (1 - P_1)(1 - q_1)\big) + \operatorname{Prob}\{R_i = 0\}\big(P_0(1 - q_1) + (1 - P_0)q_1\big) \\ &= \ 1/2 + 2(P_1 - P_0)(q_1 - 1/2) \end{aligned} \tag{12.2.4}$$

We can also express p_i: it is the expected value of the following expression:

$$q_1 P_1 + q_0 P_0.$$

On the other hand, p_{i-1} is the expected value of the following expression:

$$\operatorname{Prob}\{R_i = 1\} P_1 + \operatorname{Prob}\{R_i = 0\} P_0 = (P_1 + P_0)/2.$$

Subtracting these, we find that $p_i - p_{i-1}$ is the expected value of the difference, which is

$$(P_1 - P_0)(q_1 - 1/2).$$

We found that this expected value is at least n^{-c}/N. It follows then from the estimate 12.2.4 that the probability of success in our prediction is at least $1/2 + 2n^{-c}/N$. ∎

The security of some pseudo-random bit generators can be derived from some unproved assumptions that are nevertheless rather plausible. These are discussed below.

12.3 One-way functions

An important property of any pseudo-random bit generator $G_n(s)$ is that it turns the seed s into a sequence $x = G_n(s)$ in polynomial time but the inverse operation, finding a seed s from x, seems hard. Every search problem connected with an NP problem can be formulated similarly. In the original formulation, what is given is a string x of some length N, and the problem is to find some witness y such that the string $x\&y$ is in some polynomial language $\mathcal{L}$. This problem can be considered an inversion problem for a function F defined as follows: $F(x\&y) = x$ if $x\&y$ is in $\mathcal{L}$ and, say, the empty string otherwise. The difficulty of NP-problems is therefore the same as the difficulty of the inversion problem for polynomial-time functions. If P $\ne$ NP, as many believe, then there are polynomial-time functions whose inversion problem is not solvable in polynomial time. A pseudo-random bit generator is, however, not just a function that is hard to invert. An inversion algorithm is not only required to fail, it is required to fail to find some s' with $G_n(s') = G_n(s)$ on a significant fraction of all inputs s. (It is even required to fail in the easier problem of predicting any bit of $G_n(s)$ from the rest.)

(12.3.1) **Definition** Let $f(x, n)$ be a function on strings computable in time polynomial in n. We say that f is *one-way* if for all randomized algorithms A running in time polynomial in n, constants $c > 0$, for all sufficiently large n, the following holds. If x of length n is chosen randomly and n, $y = f(x)$ is given as input to A then the output of A differs from x with probability at least n^{-c}. $\diamondsuit$

(12.3.2) **Remark** Note an important feature of this definition: the inversion algorithm must fail with significant probability. But the probability distribution used here is not uniform over all its inputs y; rather, it is the distribution of $y = f(x)$ when x is chosen uniformly. $\diamondsuit$

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 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.

The quadratic residuosity problem. This problem, also a classical difficult problem, is not exactly an inversion problem.

For a modulus m, a residue r is called *quadratic* if there is an x with $x^2 \equiv r \pmod{m}$. The mapping $x \to x^2$ is certainly not one-to-one since $x^2 = (-x)^2$. Therefore there must be residues that are not quadratic. For a composite modulus, it seems to be very difficult to decide about a certain residue whether it is quadratic: no polynomial algorithm was found for this problem though it has been investigated for almost two centuries.

If the modululus is a prime then squaring corresponds to the multiplying of the discrete logarithm (index) by 2 modulo $p - 1$. Thus, the quadratic residues are exactly the residues with an even index. Squaring is multiplication by 2 of the index modulo $p-1$, which is clearly a one-to-one operation iff $(p-1)/2$ is odd. Such primes p will be called here *Blum primes*, (in honor of M. Blum who first used this distinction for the purposes of cryptography). Certain things are easier for Blum primes, so we restrict ourselves to them. Numbers of the form $m = pq$ where p, q are Blum primes are called Blum numbers. The squaring operation is also one-to-one on the quadradic residues of a Blum number. It follows that exactly one of the two square roots of a quadratic residue is a quadratic residue. Let us call it the *principal squareroot.*

For prime moduli, it is easy to decide whether a certain residue is quadratic.

(12.3.3) **Theorem** *Let p be a prime number. A residue x is quadratic modulo p iff $x^{(p-1)/2} \equiv 1 \pmod{p}$.*

12.2 **Exercise** Prove this theorem. $\diamondsuit$

This theorem implies that p is a Blum prime iff -1 is a quadratic nonresidue.

Let us make some remarks on the square root problem. If p is a Blum prime then it is easy to find the square roots for each quadratic residue. Let $t = (p-1)/2$. If $b \equiv a^2 \pmod{p}$ then $c = b^{(t+1)/2}$ is a square root of b. Indeed, $c^2 \equiv b^{t+1} \equiv a^{2t}b \equiv b$. Similarly, if m is a product of Blum primes p, q then it is similarly easy to find a squareroot with $t = (p-1)(q-1)/4$. This algorithm can be strengthened into a polynomial probabilistic algorithm to find squareroots modulo an arbitrary prime but we do not go into this.

We found that if m is a Blum number pq then on the set of quadratic residues, squaring is an operation that is easy to invert provided somebody gives us $(p-1)(q-1)$. If we do not know the factorization of m then we do not know $(p-1)(q-1)$ which leaves us apparently without a clue to find a squareroot.

Chance amplification. Quadratic residuosity has the curious property that algorithms that guess the right answer just 51% of the time when averaged over all residues, can be "amplified" to guess correctly 99% of the time. (Notice that the 51% success probability is known to hold when we average over all residues. For some residues, it can be much worse.) To be more exact, let us assume that an algorithm is known to guess correctly with probability $1/2 + \varepsilon$ provided its inputs are quadratic residues or their negatives, chosen with uniform probability.

Suppose that we have to decide whether x is a quadratic residue and we know already that either x or $-x$ is. Then we choose some random residues $r_1, r_2, \ldots$ and form the integers

$y_i = r_i^2 x \bmod m$. It is easy to see that if x is a quadratic residue then the numbers y_i are uniformly distributed over the quadratic residues; otherwise, they are uniformly distributed over the negatives of quadratic residues. It follows that our test on y_i gives the same answer as on x, correctly with probability $1/2+\varepsilon$ in each case. But now we can repeat the experiment for a large number y_i (say, c/ε^2 times where c is an appropriate constant) and take the answer given in the majority of cases. The law of large numbers shows (details in an exercise) that the error probability can be driven below ε this way.

A hard-core bit and its use for pseudorandom bit generation. If the modulus m has the property that it is a product of two Blum primes the quadratic residue problem still seems just as difficult as its general case. If the modulus is a Blum prime then it is easy to see that for each primitive residue x, exactly one of the two numbers $x, -x$ is a quadratic residue. For a modulus that is the product of two Blum primes, at most one of the two numbers $x, -x$ can be quadratic residue since the operation $x \to x^2$ is bijective on quadrative residues. There is a polynomial algorithm to decide whether at least one of them is quadratic residue (compute the so-called Jacobi symbol, using the generalized quadratic reciprocity theorem). However, even if we learn that one of them is a quadratic residue it still remains difficult whether x is the one.

It follows that no polynomial algorithm is known for finding out, given a quadratic residue y modulo m, the *parity of its principal squareroot*. Indeed, if there was such an algorithm A then taking a pair $x, -x$ of numbers about which it is known that one of them is a quadratic residue, we can form $y = x^2$ and applying the algorithm, we could find the parity of its principal squareroot, which is one of the the numbers $x, -x$. Since the parity of $-x \equiv m - x$ differs from that of x, knowing the parity would identify the quadratic residue among them, which we agreed is a difficult problem. Taking the "chance amplification" property of the quadratic residue problem into account, we can assert that it is difficult to predict the parity of the principal squareroot even with probability just slightly above $1/2$.

We can say that the parity function is a *hard-core bit* with respect to the operation of squaring over quadratic residues modulo m: this one bit of information is (seemingly) lost by the squaring operation.

Let us return to Example 12.2.2. For a random Blum number $m = pq$ and a random positive integer $x < m$, we chose $x_0 = x$, $x_{i+1} = x_i^2 \bmod n$ for $i > 0$. The bits $y_i = x_i \bmod 2$ were outputted. Suppose that this bit generator is not pseudorandom. Then there is a way to predict, from the bits $y_{i+1}, y_{i+1}, \ldots$, the bit y_i with probability $1/2 + \varepsilon$. From the way the numbers x_i are generated it is clear that they are uniformly distributed over the set of quadratic residues. Suppose that somebody gives us x_{i+1}. Then we know all the bits $y_{i+1}, y_{i+2}, \ldots$. According to our assumption, we can find y_i with probability $1/2 + \varepsilon$. But this is the parity of the principal squareroot of the uniformly distributed quadratic residue x_{i+1}, which we assumed cannot be done.

Generalizations. The above construction of a pseudo-random bit generator can be generalized. As a result of the work of a number of researchers (Blum, Goldreich, Goldwasser, Impagliazzo, Levin, Luby, Micali, Yao), every one-way function can be used, in a very simple way, to construct a pseudo-random generator. (Moreover, it is even enough to know that one-way functions exist, in order to define such a generator.)

12.4 Application of pseudo-number generators to cryptography

(12.4.1) **Example** At the beginning of the present section, we have mentioned one-time pads, with their advantages and disadvantages. A pseudo-random bit generator G_n can be used to produce a *pseudo one-time pad* instead. Now, the shared key can be much shorter. If it has length n the parties can use it to encode N-bit strings x as $x \oplus G_n(k)$, where N is much larger than n. $\diamondsuit$

(12.4.2) **Example** In this realization of public-key cryptography, the encryption is probabilistic, but otherwise, it will fit into the above model. The encryption key consists of a Blum number $m = pq$. The decryption key is the pair (p, q) of Blum primes. The key idea is that Alice can encrypt a bit b of information in the following way. She chooses a random residue r mod m and sends $x = (-1)^b r^2$. Now, x is a quadratic residue if and only if $b = 0$. Bob can decide this since he knows p and q. But without this, the problem seems hopeless. For sending the next bit, Alice can choose a new random r.

A more economical encryption along similar ideas is possible using the Blum-Blum-Shub pseudo-random bit generator defined above. For encryption, one chooses a random residue s_0 and computes the pseudo-random sequence $b = (b_1, \dots, b_N)$ by $s_{i+1} = s_i^2 \pmod m$, $b_i = s_i \bmod 2$. The encoded message will be the pair $(s_N, b \oplus x)$. The decryptor knows p, q and therefore can compute $s_N, s_{N-1}, \dots$ from s_N, which eventually gives him b and x. Due to the difficulty of finding the principal squareroots discussed above, to the interceptor, $b \oplus x$ will be indistinguishable from random strings, even when knowing m and s_N. $\diamondsuit$

Other applications Pseudo-random generators and trap-door functions have many other exciting applications. Using them, it is possible to implement all kinds of complicated protocols involving the interaction of several participants, with sharp requirements on who is permitted to know what. E.g., using a few interactions involving randomization and encryption, it is possible for a fairly clever Alice to convince Bob that a certain graph has a Hamiltonian circuit, in a way that Bob will get absolutely no extra information out of the communication that would help him finding the circuit. This sort of proof is called a "zero-knowledge proof". Another interesting application (by Goldreich, Micali, Widgerson, 1987) helps a number of engineers jealously protecting their private know-how (say, a string x_i) who have to cooperate in building something (say, computing a function $f(x_1, \dots, x_n)$ in polynomial time). It is possible to arrange the cooperation in such a way that none of them (not even a small clique of them) can learn anything extra about the private information of the others, besides the value of f.

12.3 **Exercise** Prove that the discrete logarithm problem has a chance-amplification property similar to the one of the quadratic residuosity problem discussed in the notes. $\diamondsuit$

12.4 **Exercise** Let $G_n(s)$ be a bit generator, giving out strings of length $N = n^k$. Suppose that there is a polynomial algorithm that computes a function $f_n(x)$ from strings of length N into strings of length $0.9N$ and another polynomial algorithm computing a function $g_n(p)$

from strings of length $0.9N$ to strings of length N, with the property that for all strings x of the form $G_n(s)$ we have $g_n(f_n(x)) = x$ (in other words, the strings generated by G_n can be compressed in polynomial time, with polynomial-time decompression). Prove that then G_n is not a pseudo-random bit generator. In short, pseudo-random strings do not have polynomial-time compression algorithms (Kolmogorov-codes). $\diamondsuit$

Literature

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

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

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

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

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

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

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

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

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

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