

Introduction

All computers, beginning with Babbage's unconstructed "analytical machine"¹ and ending with the Cray, are based on the very same principles. From the logical point of view a computer consists of bits (variables, taking the values 0 and 1), and a program — that is, a sequence of operations, each using some bits. Of course, the newest computers work faster than old ones, but progress in this direction is limited. It is hard to imagine that the size of a transistor or another element will ever be smaller than 10^{-8} cm (the diameter of the hydrogen atom) or that the clock frequency will be greater than 10^{15} Hz (the frequency of atomic transitions), so that even the supercomputers of the future will not be able to solve computational problems having *exponential* complexity. Let us consider, for example, the problem of factoring an integer number x into primes. The obvious method is to attempt to divide x by all numbers from 2 to \sqrt{x} . If x has n digits (as written in the binary form), we need to go through $\sim \sqrt{x} \sim 2^{n/2}$ trials. There exists an ingenious algorithm that solves the same problem in approximately $\exp(cn^{1/3})$ steps (c is a constant). But even so, to factor a number of a million digits, a time equal to the age of the Universe would not suffice. (There may exist more effective algorithms, but it seems impossible to dispense with the exponential.)

There is, however, another way of speeding up the calculation process for several special classes of problems. The situation is such that ordinary computers do not employ all the possibilities that are offered to us by nature.

¹Charles Babbage began his work on the "analytical machine" project in 1833. In contrast to calculating devices already existed at the time, his was supposed to be a *universal* computer. Babbage devoted his whole life to its development, but was not successful in realizing his dream. (A simpler, nonuniversal machine was *partially* constructed. In fact, this smaller project could have been completed — in 1991 the machine was produced in accordance with Babbage's design.)

This assertion may seem extremely obvious: in nature there exists a multitude of processes that are unlike operations with zeros and ones. We might attempt to use those processes for the creation of an analog computer. For example, interference of light can be used to compute the Fourier transform. However, in most cases the gain in speed is not major, i.e., it depends weakly on the size of the device. The reason lies in the fact that the equations of *classical* physics (for example, Maxwell's equations) can be effectively solved on an ordinary digital computer. What does “effectively” mean? The calculation of an interference pattern may require more time than the real experiment by a factor of a million, because the speed of light is great and the wave length is small. However, as the size of the modelled physical system gets bigger, the required number of computational operations grows at a moderate rate — as the size raised to some power or, as is customarily said in complexity theory, *polynomially*. (As a rule, the number of operations is proportional to the quantity Vt , where V is the volume and t is the time.) Thus we see that classical physics is too “simple” from the computational point of view.

Quantum mechanics is more interesting from this perspective. Let us consider, for example, a system of n spins. Each spin has two so-called *basis states* ($0 = \text{“spin up”}$ and $1 = \text{“spin down”}$), and the whole system has 2^n basis states $|x_1, \dots, x_n\rangle$ (each of the variables x_1, \dots, x_n takes values 0 or 1). By a general principle of quantum mechanics, $\sum_{x_1, \dots, x_n} c_{x_1, \dots, x_n} |x_1, \dots, x_n\rangle$ is also a possible state of the system; here c_{x_1, \dots, x_n} are complex numbers called *amplitudes*. The summation sign must be understood as a pure formality. In fact, the “sum” (also called a *superposition*) represents a new mathematical object — a vector in a 2^n -dimensional complex vector space. Physically, $|c_{x_1, \dots, x_n}|^2$ is the probability to find the system in the basis state $|x_1, \dots, x_n\rangle$ by a measurement of the values of the variables x_j . (We note that such a measurement destroys the superposition.) For this to make sense, the formula $\sum_{x_1, \dots, x_n} |c_{x_1, \dots, x_n}|^2 = 1$ must hold. Therefore, the general state of the system (i.e., a superposition) is a unit vector in the 2^n -dimensional complex space. A state change over a specified time interval is described by a unitary matrix of size $2^n \times 2^n$. If the time interval is very small ($\ll \hbar/J$, where J is the energy of spin-spin interaction and \hbar is Planck's constant), then this matrix is rather easily constructed; each of its elements is easily calculated knowing the interaction between the spins. If, however, we want to compute the change of the state over a large time interval, then it is necessary to multiply such matrices. For this purpose an exponentially large number of operations is needed. Despite much effort, no method has been found to simplify this computation (except for some special cases). Most plausibly, simulation of quantum mechanics is indeed an exponentially hard computational problem. One may think this is unfortunate, but let us take

a different point of view: quantum mechanics being *hard* means it is *powerful*. Indeed, a quantum system effectively “solves” a complex computational problem — it models its very self.

Can we use quantum systems for solving other computational problems? What would be a mathematical model of a quantum computer that is just as independent of physical realization as are models of classical computation?² It seems that these questions were first posed in 1980 in the book by Yu. I. Manin [49]. They were also discussed in the works of R. Feynman [23, 24] and other authors. In 1985 D. Deutsch [20] proposed a concrete mathematical model — the quantum Turing machine, and in 1989 an equivalent but more convenient model — the quantum circuit [21] (the latter was largely based on Feynman’s ideas).

What exactly is a quantum circuit? Suppose that we have N spins, each located in a separate numbered compartment and completely isolated from the surrounding world. At each moment of time (as a computer clock ticks) we choose, at our discretion, any two spins and act on them with an arbitrary 4×4 unitary matrix. A sequence of such operations is called a *quantum circuit*. Each operation is determined by a pair of integers, indexing the spins, and sixteen complex numbers (the matrix entries). So a quantum circuit is a kind of computer program, which can be represented as text and written on paper. The word “quantum” refers to the way this program is executed.

Let us try to use a quantum circuit for calculating a function $F : \mathbb{B}^n \rightarrow \mathbb{B}^m$, where $\mathbb{B} = \{0, 1\}$ is the set of values of a classical bit.³ It is necessary to be able to enter the initial data, perform the computations, and read out the result. Input into a quantum computer is a sequence (x_1, \dots, x_n) of zeros and ones — meaning that we prepare an initial state $|x_1, \dots, x_n, 0, \dots, 0\rangle$. (The amount of initial data, n , is usually smaller than the overall number of “memory cells,” i.e., of spins, N . The remaining cells are filled with zeros.) The initial data are fed into a quantum circuit, which depends on the *problem being solved*, but not on the specific initial data. The circuit turns the initial state into a new quantum state,

$$|\psi(x_1, \dots, x_n)\rangle = \sum_{y_1, \dots, y_N} c_{y_1, \dots, y_N}(x_1, \dots, x_n) |y_1, \dots, y_N\rangle,$$

²The standard mathematical model of an ordinary computer is the Turing machine. Most other models in use are polynomially equivalent to this one and to each other, i.e., a problem, that is solvable in L steps in one model, will be solvable in cL^k steps in another model, where c and k are constants.

³Any computational problem can be posed in this way. For example, if we wish to solve the problem of factoring an integer into primes, then $(x_1, \dots, x_n) = x$ (in binary notation) and $F(x)$ is a list of prime factors (in some binary code).

which depends on (x_1, \dots, x_n) . It is now necessary to read out the result. If the circuit were classical (and correctly designed to compute F), we would expect to find the answer in the first m bits of the sequence (y_1, \dots, y_N) , i.e., we seek $(y_1, \dots, y_m) = F(x_1, \dots, x_n)$. To determine the actual result in the quantum case, the values of all spins should be *measured*. The measurement may produce *any* sequence of zeros and ones (y_1, \dots, y_N) , the probability of obtaining such a sequence being equal to $|c_{y_1, \dots, y_N}(x_1, \dots, x_n)|^2$. A quantum circuit is “correct” for a given function F if the correct answer $(y_1, \dots, y_m) = F(x_1, \dots, x_n)$ is obtained with *probability* that is sufficiently close to 1. By repeating the computation several times and choosing the answer that is encountered most frequently, we can reduce the probability of an error to be as small as we want.

We have just formulated (omitting some details) a mathematical model of quantum computation. Now, two questions arise naturally.

1. For which problems does quantum computation have an advantage in comparison with classical?
2. What system can be used for the physical realization of a quantum computer? (This does not necessarily have to be a system of spins.)

With regard to the first question we now know the following. First, on a quantum computer it is possible to model an arbitrary quantum system in *polynomially* many steps. This will allow us (when quantum computers become available) to predict the properties of molecules and crystals and to design microscopic electronic devices, say, 100 atoms in size. Presently such devices lie at the edge of technological possibility, but in the future they will likely be common elements of ordinary computers. So, a quantum computer will not be a thing to have in every home or office, but it will be used to make such things.

A second example is factoring integers into primes and analogous number-theoretic problems. In 1994 P. Shor [62] found a quantum algorithm⁴ which factors an n -digit integer in about n^3 steps. This beautiful result could have an outcome that is more harmful than useful: factoring allows one to break the most commonly used cryptosystem (RSA), to forge electronic signatures, etc. (But anyway, building a quantum computer is such a difficult task that cryptography users may have good sleep — at least, for the next 10 years.) The method at the core of Shor’s algorithms deals with Abelian groups. Some non-Abelian generalizations have been found, but it remains to be seen if they can be applied to any practical problem.

⁴Without going into detail, a quantum algorithm is much the same thing as a quantum circuit. The difference lies in the fact that a circuit is defined for problems of fixed size ($n = \text{const}$), whereas an algorithm applies to any n .

A third example is a search for a needed entry in an unsorted database. Here the gain is not so significant: to locate one entry in N we need about \sqrt{N} steps on a quantum computer, compared to N steps on a classical one. As of this writing, these are all known examples — not because quantum computers are useless for other problems, but because their theory has not been worked out yet. We can hope that there will soon appear new mathematical ideas that will lead to new quantum algorithms.

The physical realization of a quantum computer is an exceedingly interesting, but difficult problem. Only a few years ago doubts were expressed about its solvability in principle. The trouble is that an arbitrary unitary transformation can be realized only with certain accuracy. Apart from that, a system of spins or a similar quantum system cannot be fully protected from the disturbances of the surrounding environment. All this leads to errors that accumulate in the computational process. In $L \sim \delta^{-1}$ steps (where δ is the precision of each unitary transformation) the probability of an error will be of the order of 1, which renders the computation useless. In part this difficulty can be overcome using *quantum error-correcting codes*. In 1996 P. Shor [65] proposed a scheme of error correction in the quantum computing process (fault-tolerant quantum computation). The original method was not optimal but it was soon improved by a number of authors. The end result amounts to the following. There exists some threshold value δ_0 such that for any precision $\delta < \delta_0$ arbitrarily long quantum computation is possible. However, for $\delta > \delta_0$ errors accumulate faster than we can succeed in correcting them. By various estimates, δ_0 lies in the interval from 10^{-6} to 10^{-2} (the exact value depends on the character of the disturbances and the circuit that is used for error correction).

So, there are no obstacles in principle for the realization of a quantum computer. However, the problem is so difficult that it can be compared to the problem of controlled thermonuclear synthesis. In fact, it is essential to simultaneously satisfy several almost contradictory demands:

1. The elements of a quantum computer — quantum bits (spins or something similar) — must be isolated from one another and from the environment.
2. It is essential to have the possibility to act selectively on each pair of quantum bits (at least, on each neighboring pair). Generally, one needs to implement several types of elementary operations (called *quantum gates*) described by different unitary operators.
3. Each of the gates must be realized with precision $\delta < \delta_0$ (see above).
4. The quantum gates must be sufficiently nontrivial, so that any other operator is, in a certain sense, expressible in terms of them.

At the present time there exist several approaches to the problem of realizing a quantum computer.

1. Individual atoms or ions. This first-proposed and best-developed idea exists in several variants. For representing a quantum bit one can employ both the usual electron levels and the levels of fine and superfine structures. There is an experimental technique for keeping an individual ion or atom in the trap of a steady magnetic or alternating electric field for a reasonably long time (of the order of 1 hour). The ion can be “cooled down” (i.e., its vibrational motion eliminated) with the aid of a laser beam. Selecting the duration and frequency of the laser pulses, it is possible to prepare an arbitrary superposition of the ground and excited states. In this way it is rather easy to control individual ions. Within the trap, one can also place two or more ions at distances of several microns one from another, and control each of them individually. However, it is rather difficult to choreograph the interactions between the ions. To this end it has been proposed that collective vibrational modes (ordinary mechanical vibrations with a frequency of several MHz) be used. Dipole-dipole interactions could also be used, with the advantage of being a lot faster. A second method (for neutral atoms) is as follows: place atoms into separate electromagnetic resonators that are coupled to one another (at the moment it is unclear how to achieve this technically). Finally, a third method: using several laser beams, one can create a periodic potential (“optical lattice”) which traps unexcited atoms. However, an atom in an excited state can move freely. Thus, by exciting one of the atoms for a certain time, one lets it move around and interact with its neighbors. This field of experimental physics is now developing rapidly and seems to be very promising.

2. Nuclear magnetic resonance. In a molecule with several different nuclear spins, an arbitrary unitary transformation can be realized by a succession of magnetic field pulses. This has been tested experimentally at room temperature. However, for the preparation of a suitable initial state, a temperature $< 10^{-3}$ K is required. Apart from difficulties with the cooling, undesirable interactions between the molecules increase dramatically as the liquid freezes. In addition, it is nearly impossible to address a given spin selectively if the molecule has several spins of the same kind.

3. Superconducting granules and “quantum dots”. Under supercool temperatures, the unique degree of freedom of a small (submicron size) superconducting granule is its charge. It can change in magnitude by a multiple of two electron charges (since electrons in a superconductor are bound in pairs). Changing the external electric potential, one can achieve a situation where two charge states have almost the same energy. These two states can be used as basis states of a quantum bit. The granules interact with each other by means of Josephson junctions and mutual electric

capacitance. This interaction can be controlled. A quantum dot is a microstructure which can contain few electrons or even a single electron. The spin of this electron can be used as a qubit. The difficulty is that one needs to control each granule or quantum dot individually with high precision. This seems harder than in the case of free atoms, because all atoms of the same type are identical while parameters of fabricated structures fluctuate. This approach may eventually succeed, but a new technology is required for its realization.

4. Anyons. Anyons are quasi-particles (excitations) in certain two-dimensional quantum systems, e.g. in a two-dimensional electron liquid in magnetic field. What makes them special is their *topological properties*, which are stable to moderate variation of system parameters. One of the authors (A.K.) considers this approach especially interesting (in view of it being his own invention, cf. [35]), so that we will describe it in more detail. (At a more abstract level, the connection between quantum computation and topology was discussed by M. Freedman [26].)

The fundamental difficulty in constructing a quantum computer is the necessity for realizing unitary transformations with precision $\delta < \delta_0$, where δ_0 is between 10^{-2} and 10^{-6} . To achieve this it is necessary, as a rule, to control the parameters of the system with still greater precision. However, we can imagine a situation where high precision is achieved automatically, i.e., where error correction occurs on the physical level. An example is given by two-dimensional systems with anyonic excitations.

All particles in three-dimensional space are either bosons or fermions. The wave function of bosons does not change if the particles are permuted. The wave function of fermions is multiplied by -1 under a transposition of two particles. In any case, the system is unchanged when each of the particles is returned to its prior position. In two-dimensional systems, more complex behavior is possible. Note, however, that the discussion is not about fundamental particles, such as an electron, but about excitations (“defects”) in a two-dimensional electron liquid. Such excitations can move, transform to each other, etc., just like “genuine” particles.⁵ However, excitations in the two-dimensional electron liquid display some unusual properties. An excitation can have a fractional charge (for example, $1/3$ of the charge of an electron). If one excitation makes a full turn *around another*, the state of the surrounding electron liquid changes in a precisely defined manner that depends on the types of the excitations and on the *topology* of the path, but not on the specific trajectory. In the simplest case, the wave function gets multiplied by a number (which is equal to $e^{2\pi i/3}$ for anyons in

⁵Fundamental particles can also be considered as excitations in the vacuum which is, actually, a nontrivial quantum system. The difference is that the vacuum is unique, whereas the electron liquid and other “quantum media” can be designed to meet our needs.

the two-dimensional electron liquid in a magnetic field at the filling factor $1/3$). Excitations with such properties are called *Abelian anyons*. Another example of Abelian anyons is described (in a mathematical language) in Section 15.11.

More interesting are *non-Abelian anyons*, which have not yet been observed experimentally. (Theory predicts their existence in a two-dimensional electron liquid in a magnetic field at the filling factor $5/2$.) In the presence of non-Abelian anyons, the state of the surrounding electron liquid is degenerate, the multiplicity of the degeneracy depending on the number of anyons. In other words, there exist not one, but many states, which can display arbitrary quantum superpositions. It is utterly impossible to act on such a superposition without moving the anyons, so the system is ideally protected from perturbations. If one anyon is moved around another, the superposition undergoes a certain unitary transformation. This transformation is *absolutely precise*. (An error can occur only if the anyon “gets out of hand” as a result of quantum tunneling.)

At first glance, the design using anyons seems least realistic. Firstly, Abelian anyons will not do for quantum computation, and non-Abelian ones are still awaiting experimental discovery. But in order to realize a quantum computer, it is necessary to control (i.e., detect and drag by a specified path) *each* excitation in the system, which will probably be a fraction of a micron apart from each other. This is an exceedingly complex technical problem. However, taking into account the high demands for precision, it may not be at all easier to realize any of the other approaches we have mentioned. Beyond that, the idea of *topological quantum computation*, lying at the foundation of the anyonic approach, might be expedited by other means. For example, the quantum degree of freedom protected from perturbation, might shoot up at the end of a “quantum wire” (a one-dimensional conductor with an odd number of propagating electronic modes, placed in contact with a three-dimensional superconductor).

Thus, the idea of a quantum computer looks so very attractive, and so very unreal. It is likely that the design of an ordinary computer was perceived in just that way at the time of Charles Babbage, whose invention was realized only a hundred years later. We may hope that in our time the science and the industry will develop faster, so that we will not have to wait that long. Perhaps a couple of fresh ideas plus a few years for working out a new technology will do.

Classical Computation

1. Turing machines

Note. In this section we address the abstract notion of computability, of which we only need a few basic properties. Therefore our exposition here is very brief. For the most part, the omitted details are simply exercises in programming a Turing machine, which is but a primitive programming language. A little of programming experience (in any language) suffices to see that these tasks are doable but tedious.

Informally, an *algorithm* is a set of instructions; using it, “we need only to carry out what is prescribed as if we were robots: neither understanding, nor cleverness, nor imagination is required of us” [39]. Applying an algorithm to its *input* (initial data) we get some *output* (result). (It is quite possible that computation never terminates for some inputs; in this case we get no result.)

Usually inputs and outputs are strings. A *string* is a finite sequence of symbols (characters, letters) taken from some finite *alphabet*. Therefore, before asking for an algorithm that, say, factors polynomials with integer coefficients, we should specify the encoding, i.e., specify some alphabet A and the representation of polynomials by strings over A . For example, each polynomial may be represented by a string formed by digits, letter x , signs $+$, $-$ and \times . In the answer, two factors can be separated by a special delimiter, etc.

One should be careful here because sometimes the encoding becomes really important. For example, if we represent large integers as bit strings (in binary), it is rather easy to compare them (to find which of two given integers is larger), but multiplication is more difficult. On the other hand, if an

integer is represented by its remainders modulo different primes p_1, p_2, \dots, p_n (using the Chinese remainder theorem; see Theorem A.5 in Appendix A), it is easy to multiply them, but comparison is more difficult. So we will specify the encoding in case of doubt.

We now give a formal definition of an algorithm.

1.1. Definition of a Turing machine.

Definition 1.1. A Turing machine (TM) consists of the following components:

- a finite set S called the *alphabet*;
- an element $\sqcup \in S$ (blank symbol);
- a subset $A \subset S$ called the *external alphabet*; we assume that the blank symbol does not belong to A ;
- a finite set Q whose elements are called *states* of the TM;
- an *initial state* $q_0 \in Q$;
- a *transition function*, specifically, a partial function

$$(1.1) \quad \delta : Q \times S \rightarrow Q \times S \times \{-1, 0, 1\}.$$

(The term “*partial function*” means that the domain of δ is actually a subset of $Q \times S$. A function that is defined everywhere is called *total*.)

Note that there are infinitely many Turing machines, each representing a particular algorithm. Thus the above components are more like a computer program. We now describe the “hardware” such programs run on.

A Turing machine has a *tape* that is divided into *cells*. Each cell carries one symbol from the machine alphabet S . We assume that the tape is infinite to the right. Therefore, the content of the tape is an infinite sequence $\sigma = s_0, s_1, \dots$ (where $s_i \in S$).

A Turing machine also has a read-write *head* that moves along the tape and changes symbols: if we denote its position by $p = 0, 1, 2, \dots$, the head can read the symbol s_p and write another symbol in its place.

Position of head	∇				
Cells	s_0	s_1	\dots	s_p	\dots
Cell numbers	0	1		p	

The behavior of a Turing machine is determined by a control device, which is a finite-state automaton. At each step of the computation this device is in some state $q \in Q$. The state q and the symbol s_p under the head determine the action performed by the TM: the value of the transition function, $\delta(q, s_p) = (q', s', \Delta p)$, contains the new state q' , the new symbol

s' , and the shift Δp (for example, $\Delta p = -1$ means that the head moves to the left).

More formally, the *configuration* of a TM is a triple $\langle \sigma; p; q \rangle$, where σ is an infinite sequence s_0, \dots, s_n, \dots of elements of S , p is a nonnegative integer, and $q \in Q$. At each step the TM changes its configuration $\langle \sigma; p; q \rangle$ as follows:

- (a) it reads the symbol s_p ;
- (b) it computes the value of the transition function: $\delta(q, s_p) = (q', s', \Delta p)$ (if $\delta(q, s_p)$ is undefined, the TM stops);
- (c) it writes the symbol s' in cell p of the tape, moves the head by Δp , and passes to state q' . In other words, the new configuration of the TM is the triple $\langle s_0, \dots, s_{p-1}, s', s_{p+1}, \dots; p + \Delta p; q' \rangle$. (If $p + \Delta p < 0$, the TM stops.)

Perhaps everyone would agree that these actions require neither cleverness, nor imagination.

It remains to define how the input is given to the TM and how the result is obtained. Inputs and outputs are strings over A . An input string α is written on the tape and is padded by blanks. Initially the head is at the left end of the tape; the initial state is q_0 . Thus the initial configuration is $\langle \alpha \sqcup \dots; 0; q_0 \rangle$. Subsequently, the configuration is transformed step by step using the rules described above, and we get the sequence of configurations

$$\langle \alpha \sqcup \dots; 0; q_0 \rangle, \langle \sigma_1; p_1; q_1 \rangle, \langle \sigma_2; p_2; q_2 \rangle, \dots$$

As we have said, this process terminates if δ is undefined or the head bumps into the (left) boundary of the tape ($p + \Delta p < 0$). After that, we read the tape from left to right (starting from the left end) until we reach some symbol that does not belong to A . The string before that symbol will be the output of the TM.

1.2. Computable functions and decidable predicates. Every Turing machine M computes a partial function $\varphi_M: A^* \rightarrow A^*$, where A^* is the set of all strings over A . By definition, $\varphi_M(\alpha)$ is the output string for input α . The value $\varphi_M(\alpha)$ is undefined if the computation never terminates.

Definition 1.2. A partial function f from A^* to A^* is *computable* if there exists a Turing machine M such that $\varphi_M = f$. In this case we say that f is *computed by* M .

Not all functions are computable because the set of all functions of type $A^* \rightarrow A^*$ is uncountable, while the set of all Turing machines is countable. For concrete examples of noncomputable functions see Problems 1.3–1.5.

By a *predicate* we mean a function with Boolean values: 1 (“true”) or 0 (“false”). Informally, a predicate is a property that can be true or false. Normally we consider predicates whose domain is the set A^* of all strings over some alphabet A . Such predicates can be identified with subsets of A^* : a predicate P corresponds to the set $\{x : P(x)\}$, i.e., the set of strings x for which $P(x)$ is true. Subsets of A^* are also called *languages* over A .

As has been said, a predicate P is a function $A^* \rightarrow \{0, 1\}$. A predicate is called *decidable* if this function is computable. In other words, a predicate P is decidable if there exists a Turing machine that answers question “is $P(\alpha)$ true?” for any $\alpha \in A^*$, giving either 1 (“yes”) or 0 (“no”). (Note that this machine must terminate for any $\alpha \in A^*$.)

The notions of a computable function and a decidable predicate can be extended to functions and predicates in several variables in a natural way. For example, we can fix some separator symbol $\#$ that does not belong to A and consider a Turing machine M with external alphabet $A \cup \{\#\}$. Then a partial function $\varphi_{M,n} : (A^*)^n \rightarrow A^*$ is defined as follows:

$$\varphi_{M,n}(\alpha_1, \dots, \alpha_n) = \text{output of } M \text{ for the input } \alpha_1\#\alpha_2\#\dots\#\alpha_n.$$

The value $\varphi_{M,n}(\alpha_1, \dots, \alpha_n)$ is undefined if the computation never terminates or the output string does not belong A^* .

Definition 1.3. A partial function f from $(A^*)^n$ to A^* is *computable* if there is a Turing machine M such that $\varphi_{M,n} = f$.

The definition of a decidable predicate can be given in the same way.

We say that a Turing machine works in time $T(n)$ if it performs at most $T(n)$ steps for any input of size n . Analogously, a Turing machine M works in space $s(n)$ if it visits at most $s(n)$ cells for any computation on inputs of size n .

1.3. Turing’s thesis and universal machines. Obviously a TM is an algorithm in the informal sense. The converse assertion is called the *Turing thesis*:

“Any algorithm can be realized by a Turing machine.”

It is called also the Church thesis because Church gave an alternative definition of computable functions that is formally equivalent to Turing’s definition. Note that the Church-Turing thesis is not a mathematical theorem, but rather a statement about our informal notion of algorithm, or the physical reality this notion is based upon. Thus the Church-Turing thesis cannot be proved, but it is supported by empirical evidence. At the early age of mathematical computation theory (1930’s), different definitions of

algorithm were proposed (Turing machine, Post machine, Church's lambda-calculus, Gödel's theory of recursive functions), but they all turned out to be equivalent to each other. The reader can find a detailed exposition of the theory of algorithms in [5, 39, 40, 48, 54, 60, 61].

We make some informal remarks about the capabilities of Turing machines. A Turing machine behaves like a person with a restricted memory, pencil, eraser, and a notebook with an infinite number of pages. Pages are of fixed size; therefore there are finitely many possible variants of filling a page, and these variants can be considered as letters of the alphabet of a TM. The person can work with one page at a time but can then move to the previous or to the next page. When turning a page, the person has a finite amount of information (corresponding to the state of the TM) in her head.

The input string is written on several first pages of the notebook (one letter per page); the output should be written in a similar way. The computation terminates when the notebook is closed (the head crosses the left boundary) or when the person does not know what to do next (δ is undefined).

Think about yourself in such a situation. It is easy to realize that by memorizing a few letters near the head you can perform any action in a fixed-size neighborhood of the head. You can also put extra information (in addition to letters from the external alphabet) on pages. This means that you extend the tape alphabet by taking the Cartesian product with some other finite set that represents possible notes. You can leaf through the notebook until you find a note that is needed. You can create a free cell by moving all information along the tape. You can memorize symbols and then copy them onto free pages of the notebook. Extra space on pages may also be used to store auxiliary strings of arbitrary length (like the initial word, they are written one symbol per page). These auxiliary strings can be processed by "subroutines". In particular, auxiliary strings can be used to implement counters (integers that can be incremented and decremented). Using counters, we can address a memory cell by its number, etc.

Note that in the definition of computability for functions of type $A^* \rightarrow A^*$ we restrict neither the number of auxiliary tape symbols (the set of symbols S could be much bigger than A) nor the number of states. It is easy to see, however, that one auxiliary symbol (the blank) is enough. Indeed, we can represent each letter from $S \setminus A$ by a combination of blanks and nonblanks. (The details are left to the reader as an exercise.)

Since a Turing machine is a finite object (according to Definition 1.1), it can be encoded by a string. (Note that Turing machines with arbitrary numbers of states and alphabets of any size can be encoded by strings over a fixed alphabet.) Then for any fixed alphabet A we can consider a *universal*

Turing machine U . Its input is a pair $([M], x)$, where $[M]$ is the encoding of a machine M with external alphabet A , and x is a string over A . The output of U is $\varphi_M(x)$. Thus U computes the function u defined as follows:

$$u([M], x) = \varphi_M(x).$$

This function is *universal* for the class of computable functions of type $A^* \rightarrow A^*$ in the following sense: for any computable function $f : A^* \rightarrow A^*$, there exists some M such that $u([M], x) = f(x)$ for all $x \in A^*$. (The equality actually means that either both $u([M], x)$ and $f(x)$ are undefined, or they are defined and equal. Sometimes the notation $u([M], x) \simeq f(x)$ is used to stress that both expressions can be undefined.)

The existence of a universal machine U is a consequence of the Church-Turing thesis since our description of Turing machines was algorithmic. But, unlike the Church-Turing thesis, this is also a mathematical theorem: the machine U can be constructed explicitly and proved to compute the function u . The construction is straightforward but boring. It can be explained as follows: the notebook begins with pages where instructions (i.e., $[M]$) are written; the input string x follows the instructions. The universal machine interprets the instructions in the following way: it marks the current page, goes back to the beginning of the tape, finds the instruction that matches the current state and the current symbol, then returns to the current page and performs the action required. Actually, the situation is a bit more complex: both the current state and the current symbol of M have to be represented in U by several symbols on the tape (because the number of states of the universal machine U is fixed whereas the alphabet and the number of states of the simulated machine M are arbitrary). Therefore, we need subroutines to move strings along the tape, compare them with instructions, etc.

1.4. Complexity classes. The computability of a function does not guarantee that we can compute it in practice: an algorithm computing it can require too much time or space. So from the practical viewpoint we are interested in *effective* algorithms.

The idea of an effective algorithm can be formalized in different ways, leading to different complexity classes. Probably the most important is the class of *polynomial* algorithms.

We say that a function $f(n)$ is of *polynomial growth* if $f(n) \leq cn^d$ for some constants c, d and for all sufficiently large n . (Notation: $f(n) = \text{poly}(n)$.)

Let \mathbb{B} be the set $\{0, 1\}$. In the sequel we usually consider functions and predicates defined on \mathbb{B}^* , i.e., on binary strings.

Definition 1.4. A function F on \mathbb{B}^* is *computable in polynomial time* if there exists a Turing machine that computes it in time $T(n) = \text{poly}(n)$,

where n is the length of the input. If F is a predicate, we say that it is *decidable in polynomial time*.

The class of all functions computable in polynomial time, or all predicates decidable in polynomial time (sometimes we call them “polynomial predicates” for brevity), is denoted by P . (Some other complexity classes considered below are only defined for predicates.) Note that if F is computable in polynomial time, then $|F(x)| = \text{poly}(|x|)$, since the output length cannot exceed the maximum of the input length and the number of computation steps. (Here $|t|$ stands for the length of the string t .)

The computability in polynomial time is still a theoretic notion: if the degree of the polynomial is large (or the constant c is large), an algorithm running in polynomial time may be quite impractical.

One may use other computational models instead of Turing machines to define the class P . For example, we may use a usual programming language dealing with integer variables, if we require that all integers used in the program have at most $\text{poly}(n)$ bits.

In speaking about polynomial time computation, one should be careful about encoding. For example, it is easy to see that the predicate that is true for all *unary* representations of prime numbers (i.e., strings $1 \dots 1$ whose length N is a prime number) is polynomial. Indeed, the obvious algorithm that tries to divide N by all numbers $\leq \sqrt{N}$ runs in polynomial time, namely, $\text{poly}(N)$. On the other hand, we do not know whether the predicate $P(x) = “x \text{ is a binary representation of a prime number}”$ is polynomial or not. For this to be true, there should exist an algorithm with running time $\text{poly}(n)$, where $n = \lfloor \log_2 N \rfloor$ is the length of the binary string x . (A probabilistic polynomial algorithm for this problem is known; see below.)

Definition 1.5. A function (predicate) F on \mathbb{B}^* is computable (decidable) in polynomial space if there exists a Turing machine that computes F and runs in space $s(n) = \text{poly}(n)$, where n is the length of the input.

The class of all functions (predicates) computable (decidable) in polynomial space is called PSPACE.

Note that any machine that runs in polynomial time also runs in polynomial space, therefore $P \subseteq \text{PSPACE}$. Most experts believe that this inclusion is strict, i.e., $P \neq \text{PSPACE}$, although nobody has succeeded in proving it so far. This is a famous open problem.

Problems

- [1] **1.1.** Construct a Turing machine that reverses its input (e.g., produces “0010111” from “1110100”).

[1] **1.2.** Construct a Turing machine that adds two numbers written in binary. (Assume that the numbers are separated by a special symbol “+” that belongs to the external alphabet of the TM.)

[3!] **1.3** (“The halting problem is undecidable”). Prove that there is no algorithm that determines, for given Turing machine and input string, whether the machine terminates at that input or not.

[2] **1.4.** Prove that there is no algorithm that enumerates all Turing machines that do not halt when started with the empty tape.

(Informally, enumeration is a process which produces one element of a set after another so that every element is included in this list. Exact definition: a set $X \subseteq A^*$ is called *enumerable* if it is the set of all possible outputs of some Turing machine E .)

[3] **1.5.** Let $T(n)$ be the maximum number of steps performed by a Turing machine with $\leq n$ states and $\leq n$ symbols before it terminates starting with the empty tape. Prove that the function $T(n)$ grows faster than any computable total function $b(n)$, i.e., $\lim_{n \rightarrow \infty} T(n)/b(n) = \infty$.

The mode of operation of Turing machines is rather limited and can be extended in different ways. For example, one can consider *multitape* Turing machines that have a finite number of tapes. Each tape has its own head that can read and write symbols on the tape. There are two special tapes: an *input* read-only tape, and an *output* write-only tape (after writing a symbol the head moves to the right). A k -tape machine has an input tape, an output tape, and k work tapes.

At each step the machine reads symbols on all of the tapes (except for the output tape), and its action depends upon these symbols and the current state. This action is determined by a transition function that says what the next state is, what symbols should be written on each work tape, what movement is prescribed for each head (except for the output one), and what symbol should be written on the output tape (it is possible also that no symbol is written; in this case the output head does not move).

Initially all work tapes are empty, and the input string is written on the input tape. The output is the content of the output tape after the TM halts (this happens when the transition function is undefined or when one of the heads moves past the left end of its tape).

More tapes allow Turing machine to work faster; however, the difference is not so great, as the following problems show.

[2] **1.6.** Prove that a 2-tape Turing machine working in time $T(n)$ for inputs of length n can be simulated by an ordinary Turing machine working in time $O(T^2(n))$.

[3] **1.7.** Prove that a 3-tape Turing machine working in time $T(n)$ for inputs of length n can be simulated by a 2-tape machine working in time $O(T(n) \log T(n))$.

[3] **1.8.** Let M be a (single-tape) Turing machine that duplicates the input string (e.g., produces “blabla” from “bla”). Let $T(n)$ be its maximum running time when processing input strings of length n . Prove that $T(n) \geq \varepsilon n^2$ for some ε and for all n . What can be said about $T'(n)$, the minimum running time for inputs of length n ?

[2] **1.9.** Consider a programming language that includes 100 integer variables, the constant 0, increment and decrement statements, and conditions of type “variable = 0”. One may use **if-then-else** and **while** constructs, but recursion is not allowed. Prove that any computable function of type $\mathbb{Z} \rightarrow \mathbb{Z}$ has a program in this language.

2. Boolean circuits

2.1. Definitions. Complete bases. A *Boolean circuit* is a representation of a given Boolean function as a composition of other Boolean functions.

By a *Boolean function* of n variables we mean a function of type $\mathbb{B}^n \rightarrow \mathbb{B}$. (For $n = 0$ we get two constants 0 and 1.) Assume that some set \mathcal{A} of Boolean functions (*basis*) is fixed. It may contain functions with different arity (number of arguments).

A circuit C over \mathcal{A} is a sequence of assignments. These assignments involve n *input* variables x_1, \dots, x_n and several *auxiliary* variables y_1, \dots, y_m ; the j -th assignment has the form $y_j := f_j(u_1, \dots, u_r)$. Here f_j is some function from \mathcal{A} , and each of the variables u_1, \dots, u_r is either an input variable or an auxiliary variable that precedes y_j . The latter requirement guarantees that the right-hand side of the assignment is defined when we perform it (we assume that the values of the input variables are defined at the beginning; then we start defining y_1, y_2 , etc.).

The value of the last auxiliary variable is the *result* of computation. A circuit with n input variables x_1, \dots, x_n computes a Boolean function $F : \mathbb{B}^n \rightarrow \mathbb{B}$ if the result of computation is equal to $F(x_1, \dots, x_n)$ for any values of x_1, \dots, x_n .

If we select m auxiliary variables (instead of one) to be the output, we get the definition of the computation of a function $F : \mathbb{B}^n \rightarrow \mathbb{B}^m$ by a circuit.

A circuit can also be represented by an acyclic directed graph (as in Figure 2.1), in which vertices of in-degree 0 (*inputs* — we put them on top of the figure) are labeled by input variables; all other vertices (*gates*) are labeled with functions from \mathcal{A} in such a way that the in-degree of a vertex matches the arity of the function placed at that vertex, and each incoming

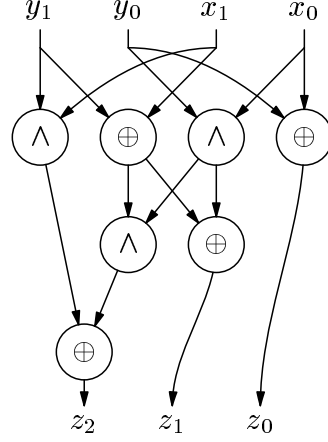


Fig. 2.1. Circuit over the basis $\{\wedge, \oplus\}$ for the addition of two 2-digit numbers: $\overline{z_2 z_1 z_0} = \overline{x_1 x_0} + \overline{y_1 y_0}$.

edge is linked to one of the function arguments. Some vertices are called *output* vertices. Since the graph is acyclic, any value assignment to the input variables can be extended (uniquely) to a consistent set of values for all vertices. Therefore, the set of values at the output vertices is a function of input values. This function is computed by a circuit. It is easy to see that this representation of a circuit can be transformed into a sequence of assignments, and vice versa. (We will not use this representation much, but it explains the name “circuit”.)

A circuit for a Boolean function is called a *formula* if each auxiliary variable, except the last one, is used (i.e., appears on the right-hand side of an assignment) exactly once. The graph of a formula is a tree whose leaves are labeled by input variables; each label may appear any number of times. (In a general circuit an auxiliary variable may be used more than once, in which case the out-degree of the corresponding vertex is more than 1.)

Why the name “formula”? If each auxiliary variable is used only once, we can replace it by its definition. Performing all these “inline substitutions”, we get an expression for f that contains only input variables, functions from the basis, and parentheses. The size of this expression approximately equals the total length of all assignments. (It is important that each auxiliary variable is used only once; otherwise we would need to replace all occurrences of each auxiliary variable by their definitions, and the size might increase exponentially.)

A basis \mathcal{A} is called *complete* if, for any Boolean function f , there is a circuit over \mathcal{A} that computes f . (It is easy to see that in this case any function of type $\mathbb{B}^n \rightarrow \mathbb{B}^m$ can be computed by an appropriate circuit.)

The most common basis contains the following three functions:

$$NOT(x) = \neg x, \quad OR(x_1, x_2) = x_1 \vee x_2, \quad AND(x_1, x_2) = x_1 \wedge x_2$$

(negation, disjunction, conjunction). Here are the value tables for these functions:

x	$\neg x$	x_1	x_2	$x_1 \vee x_2$	x_1	x_2	$x_1 \wedge x_2$
0	1	0	0	0	0	0	0
1	0	0	1	1	0	1	0
		1	0	1	1	0	0
		1	1	1	1	1	1

Theorem 2.1. *The basis $\{NOT, OR, AND\} = \{\neg, \vee, \wedge\}$ is complete.*

Proof. Any Boolean function of n arguments is determined by its value table, which contains 2^n rows. Each row contains the values of the arguments and the corresponding value of the function.

If the function takes value 1 only once, it can be computed by a conjunction of *literals*; each literal is either a variable or the negation of a variable. For example, if $f(x_1, x_2, x_3)$ is true (equals 1) only for $x_1 = 1, x_2 = 0, x_3 = 1$, then

$$f(x_1, x_2, x_3) = x_1 \wedge \neg x_2 \wedge x_3$$

(the conjunction is associative, so we omit parentheses; the order of literals is also unimportant).

In the general case, a function f can be represented in the form

$$(2.1) \quad f(x) = \bigvee_{\{u: f(u)=1\}} \chi_u(x),$$

where $u = (u_1, \dots, u_n)$, and χ_u is the function such that $\chi_u(x) = 1$ if $x = u$, and $\chi_u(x) = 0$ otherwise. \square

A representation of type (2.1) is called a *disjunctive normal form* (DNF). By definition, a DNF is a disjunction of conjunctions of literals. Later we will also need the conjunctive normal form (CNF) — a conjunction of disjunctions of literals. Any Boolean function can be represented by a CNF. This fact is dual to Theorem 2.1 and can be proved in a dual way (we start with functions that have only one zero in the table). Or we can represent $\neg f$ by a DNF and then get a CNF for f by negation using De Morgan's identities

$$x \wedge y = \neg(\neg x \vee \neg y), \quad x \vee y = \neg(\neg x \wedge \neg y).$$

These identities show that the basis $\{\neg, \vee, \wedge\}$ is redundant: the subsets $\{\neg, \vee\}$ and $\{\neg, \wedge\}$ also constitute complete bases. Another useful example of a complete basis is $\{\wedge, \oplus\}$.

The number of assignments in a circuit is called its *size*. The minimal size of a circuit over \mathcal{A} that computes a given function f is called the *circuit*

complexity of f (with respect to the basis \mathcal{A}) and is denoted by $c_{\mathcal{A}}(f)$. The value of $c_{\mathcal{A}}(f)$ depends on \mathcal{A} , but the transition from one finite complete basis to another changes the circuit complexity by at most a constant factor: if \mathcal{A}_1 and \mathcal{A}_2 are two finite complete bases, then $c_{\mathcal{A}_1}(f) = O(c_{\mathcal{A}_2}(f))$ and vice versa. Indeed, each \mathcal{A}_2 -assignment can be replaced by $O(1)$ \mathcal{A}_1 -assignments since \mathcal{A}_1 is a complete basis.

We are interested in asymptotic estimates for circuit complexity (up to an $O(1)$ -factor); therefore the particular choice of a complete basis is not important. We use the notation $c(f)$ for the circuit complexity of f with respect to some finite complete basis.

[2] **Problem 2.1.** Construct an algorithm that determines whether a given set of Boolean functions \mathcal{A} constitutes a complete basis. (Functions are represented by tables.)

[2!] **Problem 2.2.** Let c_n be the maximum complexity $c(f)$ for Boolean functions f in n variables. Prove that $1.99^n < c_n < 2.01^n$ for sufficiently large n .

2.2. Circuits versus Turing machines. Any predicate F on \mathbb{B}^* can be restricted to strings of fixed length n , giving rise to the Boolean function

$$F_n(x_1, \dots, x_n) = F(x_1x_2 \cdots x_n).$$

Thus F may be regarded as the sequence of Boolean functions F_0, F_1, F_2, \dots

Similarly, in most cases of practical importance a (partial) function of type $F : \mathbb{B}^* \rightarrow \mathbb{B}^*$ can be represented by a sequence of (partial) functions $F_n : \mathbb{B}^n \rightarrow \mathbb{B}^{p(n)}$, where $p(n)$ is a polynomial with integer coefficients. We will focus on predicates for simplicity, though.

Definition 2.1. A predicate F belongs to the class P/poly (“nonuniform P”) if

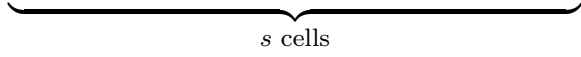
$$c(F_n) = \text{poly}(n).$$

(The term “nonuniform” indicates that a separate procedure, i.e., a Boolean circuit, is used to perform computation with input strings of each individual length.)

Theorem 2.2. $P \subset P/\text{poly}$.

Proof. Let F be a predicate decidable in polynomial time. We have to prove that $F \in P/\text{poly}$. Let M be a TM that computes F and runs in polynomial time (and therefore in polynomial space). The computation by M on some input x of length n can be represented as a space-time diagram Γ that is a rectangular table of size $T \times s$, where $T = \text{poly}(n)$ and $s = \text{poly}(n)$.

$t = 0$		$\Gamma_{0,1}$		
$t = 1$				
$t = j$		$\Gamma_{j,k-1}$	$\Gamma_{j,k}$	$\Gamma_{j,k+1}$
$t = j + 1$			$\Gamma_{j+1,k}$	
$t = T$				



 $s \text{ cells}$

In this table the j -th row represents the configuration of M after j steps: $\Gamma_{j,k}$ corresponds to cell k at time j and consists of two parts: the symbol on the tape and the state of the TM if its head is in k -th cell (or a special symbol Λ if it is not). In other words, all $\Gamma_{j,k}$ belong to $S \times (\{\Lambda\} \cup Q)$. (Only one entry in a row can have the second component different from Λ .) For simplicity we assume that after the computation stops all subsequent rows in the table repeat the last row of the computation.

There are local rules that determine the contents of a cell $\Gamma_{j+1,k}$ if we know the contents of three neighboring cells in row j , i.e., $\Gamma_{j,k-1}$, $\Gamma_{j,k}$ and $\Gamma_{j,k+1}$. Indeed, the head speed is at most one cell per step, so no other cell can influence $\Gamma_{j+1,k}$. Rules for boundary cells are somewhat special; they take into account that the head cannot be located outside the table.

Now we construct a circuit that computes $F(x)$ for inputs x of length n . The contents of each table cell can be encoded by a constant (i.e., independent of n) number of Boolean variables. These variables (for all cells) will be the auxiliary variables of the circuit.

Each variable encoding the cell $\Gamma_{j+1,k}$ depends only on the variables that encode $\Gamma_{j,k-1}$, $\Gamma_{j,k}$, and $\Gamma_{j,k+1}$. This dependence is a Boolean function with a constant number of arguments. These functions can be computed by circuits of size $O(1)$. Combining these circuits, we obtain a circuit that computes all of the variables which encode the state of every cell. The size of this circuit is $O(sT)O(1) = \text{poly}(n)$.

It remains to note that the variables in row 0 are determined by the input string, and this dependence leads to additional $\text{poly}(n)$ assignments. Similarly, to find out the result of M it is enough to look at the symbol written in the 0-th cell of the tape at the end of the computation. So the output is a function of $\Gamma_{T,0}$ and can be computed by an additional $O(1)$ -size circuit. Finally we get a $\text{poly}(n)$ -size circuit that simulates the behavior of M for inputs of length n and therefore computes the Boolean function F_n . \square

Remark 2.1. The class P/poly is bigger than P. Indeed, let $\varphi : \mathbb{N} \rightarrow \mathbb{B}$ be an arbitrary function (maybe even a noncomputable one). Consider the predicate F_φ such that $F_\varphi(x) = \varphi(|x|)$, where $|x|$ stands for the length of string x . The restriction of F_φ to strings of length n is a constant function (0 or 1), so the circuit complexity of $(F_\varphi)_n$ is $O(1)$. Therefore F_φ for any φ belongs to P/poly, although for a noncomputable φ the predicate F_φ is not computable and thus does not belong to P.

Remark 2.2. That said, P/poly seems to be a good approximation of P for many purposes. Indeed, the class P/poly is relatively small: out of 2^{2^n} Boolean functions in n variables only $2^{\text{poly}(n)}$ functions have polynomial circuit complexity (see solution to Problem 2.2). The difference between uniform and nonuniform computation is more important for bigger classes. For example, EXPTIME, the class of predicates decidable in time $2^{\text{poly}(n)}$, is a nontrivial computational class. However, the nonuniform analog of this class includes all predicates!

The arguments used to prove Theorem 2.2 can also be used to prove the following criterion:

Theorem 2.3. *F belongs to P if and only if these conditions hold:*

- (1) $F \in \text{P/poly}$;
- (2) *the functions F_n are computed by polynomial-size circuits C_n with the following property: there exists a TM that for each positive integer n runs in time $\text{poly}(n)$ and constructs the circuit C_n .*

A sequence of circuits C_n with this property is called *polynomial-time uniform*.

Note that the TM mentioned in (2) is *not* running in polynomial time since its running time is polynomial in n but not in $\log n$ (the number of bits in the binary representation of n). Note also that we implicitly use some natural encoding for circuits when saying “TM constructs a circuit”.

Proof. \Rightarrow The circuit for computing F_n constructed in Theorem 1.2 has regular structure, and it is clear that the corresponding sequence of assignments can be produced in polynomial time when n is known.

\Leftarrow This is also simple. We compute the size of the input string x , then apply the TM to construct a circuit $C_{|x|}$ that computes $F_{|x|}$. Then we perform the assignments indicated in $C_{|x|}$, using x as the input, and get $F(x)$. All these computations can be performed in polynomial (in $|x|$) time. \square

[1] **Problem 2.3.** Prove that there exists a decidable predicate that belongs to P/poly but not to P.

2.3. Basic algorithms. Depth, space and width. We challenge the reader to study these topics by working on problems. (Solutions are also provided, of course.) In Problems 2.9–2.16 we introduce some basic algorithms which are used universally throughout this book. The algorithms are described in terms of uniform (i.e., effectively constructed) sequences of circuits. In this book we will be satisfied with *polynomial-time uniformity*; cf. Theorem 2.3. [This property is intuitive and usually easy to check. Another useful notion is *logarithmic-space uniformity*: the circuits should be constructed by a Turing machine with work space $O(\log n)$ (machines with limited work space are defined below; see 2.3.3). Most of the circuits we build satisfy this stronger condition, although the proof might not be so easy.]

2.3.1. Depth. In practice (e.g., when implementing circuits in hardware), size is not the only circuit parameter that counts. Another important parameter is *depth*. Roughly, it is the time that is needed to carry out all assignments in the circuit, if we can do more than one *in parallel*. Interestingly enough, it is also related to the space needed to perform the computation (see Problems 2.17 and 2.18). In general, there is a trade-off between size and depth. In our solutions we will be willing to increase the size of a circuit a little to gain a considerable reduction of the depth (see e.g., Problem 2.14). As a result, we come up with circuits of polynomial size and poly-logarithmic depth. (With a certain notion of uniformity, the functions computed by such circuits form the so-called class NC, an interesting subclass of P.)

More formally, the *depth* of a Boolean circuit is the maximum number of gates on any path from the input to the output. The depth is $\leq d$ if and only if one can arrange the gates into d layers, so that the input bits of any gate at layer j come from the layers $1, \dots, j-1$. For example, the circuit in Figure 2.1 has depth 3.

Unless stated explicitly otherwise, we assume that all gates have bounded *fan-in*, i.e., the number of input bits. (This is always the case when circuits are built over a finite basis. Unbounded fan-in can occur, for example, if one uses *OR* gates with an arbitrary number of inputs.) We also assume that the *fan-out* (the number of times an input or an auxiliary variable is used) is bounded.¹ If it is necessary to use the variable more times, one may insert additional “trivial gates” (identity transformations) into the circuit, at the cost of some increase in size and depth. Note that a formula is a circuit in which all auxiliary variables have fan-out 1, whereas the fan-out of the input variables is unbounded.

¹This restriction is needed to allow conversion of Boolean circuits into quantum circuits without extra cost (see Section 7). However, it is in no way standard: in most studies in computational complexity, unbounded fan-out is assumed.

[1] **2.4.** Let C be an $O(\log n)$ -depth circuit which computes some function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$. Prove that after eliminating all extra variables and gates in C (those which are not connected to the output), we get a circuit of size $\text{poly}(n + m)$.

[1!] **2.5.** Let C be a circuit (over some basis B) which computes a function $f : \mathbb{B}^n \rightarrow \mathbb{B}$. Prove that C can be converted into an equivalent (i.e., computing the same function) formula C' of the same depth over the same basis. (It follows from the solution that the size of C' does not exceed c^d , where d is the depth and c is the maximal fan-in.)

[3] **2.6** (“Balanced formula”). Let C be a formula of size L over the basis $\{NOT, OR, AND\}$ (with fan-in ≤ 2). Prove that it can be converted into an equivalent formula of depth $O(\log L)$ over the same basis.

(Therefore, it does not matter whether we define the formula complexity of a function in terms of size or in terms of depth. This is not true for the circuit complexity.)

[1] **2.7.** Show that any function can be computed by a circuit of depth ≤ 3 with gates of type *NOT*, *AND*, and *OR*, if we allow *AND*- and *OR*-gates with arbitrary fan-in and fan-out.

[2] **2.8.** By definition, the function *PARITY* is the sum of n bits modulo 2. Suppose it is computed by a circuit of depth 3 containing *NOT*-gates, *AND*-gates and *OR*-gates with arbitrary fan-in. Show that the size of the circuit is exponential (at least c^n for some $c > 1$ and for all n).

2.3.2. Basic algorithms.

[3!] **2.9.** *COMPARISON*. Construct a circuit of size $O(n)$ and depth $O(\log n)$ that tells whether two n -bit integers are equal and if they are not, which one is greater.

[2] **2.10.** Let $n = 2^l$. Construct circuits of size $O(n)$ and depth $O(\log n)$ for the solution of the following problems.

a) *ACCESS BY INDEX*. Given an n -bit string $x = x_0 \cdots x_{n-1}$ (a table) and an l -bit number j (an index), find x_j .

b) *SEARCH*. Evaluate the disjunction $y = x_0 \vee \cdots \vee x_{n-1}$, and if it equals 1, find the smallest j such that $x_j = 1$.

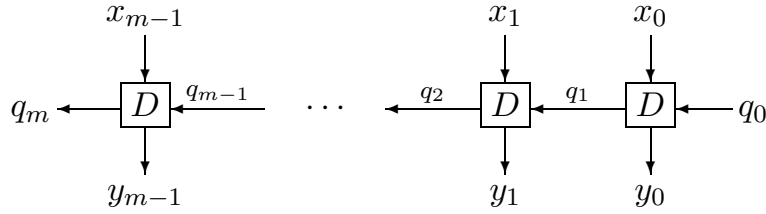
We now describe one rather general method of parallelizing computation. A *finite-state automaton* is a device with an input alphabet A' , an output alphabet A'' , a set of states Q and a transition function $D : Q \times A' \rightarrow Q \times A''$ (recall that such a device is a part of a Turing machine). It is initially set to some state $q_0 \in Q$. Then it receives input symbols $x_0, \dots, x_{m-1} \in A'$ and

changes its state from q_0 to q_1 to q_2 , etc., according to the rule²

$$(q_{j+1}, y_j) = D(q_j, x_j).$$

The iterated application of D defines a function $D^m : (q_0, x_0, \dots, x_{m-1}) \mapsto (q_m, y_0, \dots, y_{m-1})$. We may assume without loss of generality that $Q = \mathbb{B}^r$, $A' = \mathbb{B}^{r'}$, $A'' = \mathbb{B}^{r''}$; then $D : \mathbb{B}^{r+r'} \rightarrow \mathbb{B}^{r+r''}$ whereas $D^m : \mathbb{B}^{r+r'm} \rightarrow \mathbb{B}^{r+r''m}$.

The work of the automaton can be represented by this diagram:



[3!] **2.11** (“Parallelization of iteration”). Let the integers r, r', r'' and m be fixed; set $k = r + r' + r''$. Construct a circuit of size $\exp(O(k))m$ and depth $O(k \log m)$ that receives a transition function $D : \mathbb{B}^{r+r'} \rightarrow \mathbb{B}^{r+r''}$ (as a value table), an initial state q_0 and input symbols x_0, \dots, x_{m-1} , and produces the output $(q_m, y_0, \dots, y_{m-1}) = D^m(q_0, x_0, \dots, x_{m-1})$.

[2!] **2.12.** ADDITION. Construct a circuit of size $O(n)$ and depth $O(\log n)$ that adds two n -bit integers.

[3!] **2.13.** The following two problems are closely related.

a) ITERATED ADDITION. Construct a circuit of size $O(nm)$ and depth $O(\log n + \log m)$ for the addition of m n -digit numbers.

b) MULTIPLICATION. Construct a circuit with the same parameters for the multiplication of an n -digit number by an m -digit number.

[3!] **2.14.** DIVISION. This problem also comes in two variants.

a) Compute the inverse of a real number $x = \overline{1.x_1x_2\cdots} = 1 + \sum_{j=1}^{\infty} 2^{-j}x_j$ with precision 2^{-n} . By definition, this requires to find a number z such that $|z - x^{-1}| \leq 2^{-n}$. For this to be possible, x must be known with precision 2^{-n} or better; let us assume that x is represented by an $n+1$ -digit number x' such that $|x' - x| \leq 2^{-(n+1)}$. Construct an $O(n^2 \log n)$ -size, $O((\log n)^2)$ -depth circuit for the solution of this problem.

²Our definition of an automaton is not standard. We require that the automaton reads *and* outputs one symbol at each step. Traditionally, an automaton is allowed to either read a symbol, or output a symbol, or both, depending on the current state. The operation of such a general automaton can be represented as the application of an automaton in our sense (with a suitable output alphabet B) followed by substitution of a word for each output symbol.

b) Divide two integers with remainder: $(a, b) \mapsto (\lfloor a/b \rfloor, (a \bmod b))$, where $0 \leq a < 2^k b$ and $0 < b < 2^n$. In this case, construct a circuit of size $O(nk + k^2 \log k)$ and depth $O(\log n + (\log k)^2)$.

[2!] **2.15. MAJORITY.** The majority function $\text{MAJ} : \mathbb{B}^n \rightarrow \mathbb{B}$ equals 1 for strings in which the number of 1s is greater than the number of 0s, and equals 0 elsewhere. Construct a circuit of size $O(n)$ and depth $O(\log n)$ that computes the majority function.

[3!] **2.16. CONNECTING PATH.** Construct a circuit of size $O(n^3 \log n)$ and depth $O((\log n)^2)$ that checks whether two fixed vertices of an undirected graph are connected by a path. The graph has n vertices, labeled $1, \dots, n$; there are $m = n(n-1)/2$ input variables x_{ij} (where $i < j$) indicating whether there is an edge between i and j .

2.3.3. Space and width. In the solution to the above problems we strove to provide parallel algorithms, which were described by circuits of poly-logarithmic depth. We now show that, if the circuit size is not taken into account, then uniform computation with poly-logarithmic depth³ is equivalent to computation with poly-logarithmic space.

We are going to study computation with very limited space — so small that the input string would not fit into it. So, let us assume that the input string $x = x_0 \cdots x_{n-1}$ is stored in a supplementary read-only memory, and the Turing machine can access bits of x by their numbers. We may think of the input string as a function $X : j \mapsto x_j$ computed by some external agent, called “oracle”. The length of an “oracle query” j is included into the machine work space, but the length of x is not. This way we can access all input bits with space $O(\log n)$ (but no smaller).

Definition 2.2. Let $X : A^* \rightarrow A^*$ be a partial function. A *Turing machine with oracle X* is an ordinary TM with a supplementary tape, in which it can write a string z and have the value of $X(z)$ available for inspection at the next computation step.

In our case $X(z) = x_j$, where z is the binary representation of j ($0 \leq j \leq n-1$) by $\lceil \log_2 n \rceil$ digits; otherwise $X(z)$ is undefined.

The computation of a function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ by such a machine is defined as follows. Let x ($|x| = n$) be the input. We write n and another number k ($0 \leq k < m$) on the machine tape and run the machine. We allow it to query bits of x . When the computation is complete, the first cell of the tape must contain the k -th bit of $f(x)$. Note that if the work space is limited by s , then $n \leq 2^s$ and $m \leq 2^s$. The computation time is also bounded:

³As is usual, we consider circuits over a finite complete basis; the fan-in and fan-out are bounded.

the machine either stops within $\exp(O(s))$ steps, or enters a cycle and runs forever.

[2!] **2.17** (“Small depth circuits can be simulated in small space”). Prove that there exists a Turing machine that evaluates the output variables of a circuit of depth d over a fixed basis, using work space $O(d + \log m)$ (where m is the number of the output variables). The input to the machine consists of a description of the circuit and the values of its input variables.

[3!] **2.18** (“Computation with small space is parallelizable”). Let M be a Turing machine. For each choice of n, m, s , let $f_{n,m,s} : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be the function computed by the machine M with space s (it may be a partial function). Prove that there exists a family of $\exp(O(s))$ -size, $O(s^2)$ -depth circuits $C_{n,m,s}$ which compute the functions $f_{n,m,s}$.

(These circuits can be constructed by a TM with space $O(s)$, but we will not prove that.)

The reader might wonder why we discuss the space restriction in terms of Turing machines while the circuit language is apparently more convenient. So, let us ask this question: what is the circuit analog of the computation space? The obvious answer is that it is the circuit width.

Let C be a circuit whose gates are arranged into d layers. The *width* of C is the maximum amount of information carried between the layers, *not including the input variables*. More formally, for each $l = 1, \dots, d$ we define w_l to be the number of auxiliary variables from layers $1, \dots, l$ that are output variables or connected to some variables from layers $l+1, \dots, d$ (i.e., used in the right-hand side of the corresponding assignments). Then the width of C is $w = \max\{w_1, \dots, w_d\}$.

But here comes a little surprise: any Boolean function can be computed by a circuit of bounded width (see Problem 2.19 below). Therefore the width is rather meaningless parameter, unless we put some other restrictions on the circuit. To characterize computation with limited space (e.g., the class PSPACE), one has to use either Turing machines, or some class of circuits with regular structure.

[3] **2.19** (Barrington [8]). Let C be a formula of depth d that computes a Boolean function $f(x_1, \dots, x_n)$. Construct a circuit of size $\exp(O(d))$ and width $O(1)$ that computes the same function.

3. The class NP: Reducibility and completeness

3.1. Nondeterministic Turing machines. NP is the class of predicates recognizable in polynomial time by “nondeterministic Turing machines.” (The word “nondeterministic” is not appropriate but widely used.)

The class NP is defined only for predicates. One says, for example, that “the property that a graph has a Hamiltonian cycle belongs to NP”. (A *Hamiltonian cycle* is a cycle that traverses all vertices exactly once.)

We give several definitions of this class. The first uses *nondeterministic Turing machines*. A nondeterministic Turing machine (NTM) resembles an ordinary (deterministic) machine, but can nondeterministically choose one of several actions possible in a given configuration. More formally, a transition function of an NTM is multivalued: for each pair (state, symbol) there is a set of possible actions. Each action has a form (new state, new symbol, shift). If the set of possible actions has cardinality at most 1 for each state-symbol combination, we get an ordinary Turing machine.

A *computational path* of an NTM is determined by a choice of one of the possible transitions at each step; different paths are possible for the same input.

Definition 3.1. A predicate L belongs to the class NP if there exists an NTM M and a polynomial $p(n)$ such that

$$\begin{aligned} L(x) = 1 &\Rightarrow \text{there exists a computational path that gives answer} \\ &\quad \text{“yes” in time not exceeding } p(|x|); \\ L(x) = 0 &\Rightarrow \text{(version 1) there is no path with this property;} \\ &\quad \text{(version 2) ... and, moreover, there is no path (of any} \\ &\quad \text{length) that gives answer “yes”.} \end{aligned}$$

Remark 3.1. Versions 1 and 2 are equivalent. Indeed, let an NTM M_1 satisfy version 1 of the definition. To exclude “yes” answers for long computational paths, it suffices to simulate M_1 while counting its steps, and to abort the computation after $p(|x|)$ steps.

Remark 3.2. The argument in Remark 3.1 has a subtle error. If the coefficients of the polynomial p are noncomputable, difficulties may arise when we have to compare the number of steps with the value of the polynomial. In order to avoid this complication we will add to Definition 3.1 an additional requirement: $p(n)$ has integer coefficients.

Remark 3.3. By definition $P \subseteq NP$. Is this inclusion strict? Rather intense although unsuccessful attempts have been made over the past 30 years to prove the strictness. Recently S. Smale included the $P \stackrel{?}{=} NP$ problem in the list of most important mathematical problems for the new century (the other problems are the Riemann hypothesis and the Poincaré conjecture). More practical people can dream of \$1,000,000 that Clay Institute offers for the solution of this problem.

Now we give another definition of the class NP, which looks more natural. It uses the notion of a polynomially decidable predicate of two variables:

a predicate $R(x, y)$ (where x and y are strings) is *polynomially decidable* (decidable in polynomial time) if there is a (deterministic) TM that computes it in time $\text{poly}(|x|, |y|)$ (which means $\text{poly}(|x| + |y|)$ or $\text{poly}(\max\{|x|, |y|\})$ — these two expressions are equivalent).

Definition 3.2. A predicate L belongs to the class NP if it can be represented as

$$L(x) = \exists y \left((|y| < q(|x|)) \wedge R(x, y) \right),$$

where q is a polynomial (with integer coefficients), and R is a predicate of two variables decidable in polynomial time.

Remark 3.4. Let $R(x, y) = “y \text{ is a Hamiltonian cycle in the graph } x”$. More precisely, we should say: “ x is a binary encoding of some graph, and y is an encoding of a Hamiltonian cycle in that graph”. Take $q(n) = n$. Then $L(x)$ means that graph x has a Hamiltonian cycle. (We assume that the encoding of any cycle in a graph is shorter than the encoding of the graph itself.)

Theorem 3.1. *Definitions 3.1 and 3.2 are equivalent.*

Proof. *Definition 3.1 \Rightarrow Definition 3.2.* Let M be an NTM and let $p(n)$ be the polynomial of the first definition. Consider the predicate $R(x, y) = “y \text{ is a description of a computational path that starts with input } x, \text{ ends with answer ‘yes’, and takes at most } p(|x|) \text{ steps}”$. Such a description has length proportional to the computation time if an appropriate encoding is used (and even if we use a table as in the proof of Theorem 2.2, the description length is at most quadratic). Therefore for $q(n)$ in the second definition we can take $O(p(n))$ (or $O(p^2(n))$ if we use less efficient encoding).

It remains to prove that predicate R belongs to P. This is almost obvious. We must check that we are presented with a valid description of some computational path (this is a polynomial task), and that this path starts with x , takes at most $p(|x|)$ steps, and ends with “yes” (that is also easy).

Definition 3.2 \Rightarrow Definition 3.1. Let R, q be as in Definition 3.2. We construct an NTM M for Definition 3.1. M works in two stages.

First, M nondeterministically guesses y . More precisely, this means that M goes to the end of the input string, moves one cell to the right, writes $\#$, moves once more to the right, and then writes some string y (M ’s nondeterministic rules allow it to write any symbol and then move to the right, or finish writing). After that, the tape has the form $x\#y$ for some y , and M goes to the second stage.

At the second stage M checks that $|y| < q(|x|)$ (note that M can write a very long y for any given x) and computes $R(x, y)$ (using the polynomial algorithm that exists according to Definition 3.2). If $x \in L$, then there

is some y such that $|y| < q(|x|)$ and $R(x, y)$. Therefore M has, for x , a computational path of polynomial length ending with “yes”. If $x \notin L$, no computational path ends with “yes”. \square

Now we proceed to yet another description of NP that is just a reformulation of Definition 3.2, but which has a form that can be used to define other complexity classes.

Definition 3.3. Imagine two persons: King **Arthur**, whose mental abilities are polynomially bounded, and a wizard **Merlin**, who is intellectually omnipotent and knows everything. **A** is interested in some property $L(x)$ (he wants, for example, to be sure that some graph x has a Hamiltonian cycle). **M** wants to convince **A** that $L(x)$ is true. But **A** does not trust **M** (“he is too clever to be loyal”) and wants to make sure $L(x)$ is true, not just believe **M**.

So Arthur arranges that, after both he and Merlin see input string x , **M** writes a note to **A** where he “proves” that $L(x)$ is true. Then **A** verifies this proof by some polynomial proof-checking procedure.

The proof-checking procedure is a polynomial predicate

$$R(x, y) = \text{“}y \text{ is a proof of } L(x)\text{”}.$$

It should satisfy two properties:

$$\begin{aligned} L(x) = 1 &\Rightarrow \text{M can convince A that } L(x) \text{ is true by presenting some} \\ &\quad \text{proof } y \text{ such that } R(x, y); \\ L(x) = 0 &\Rightarrow \text{whatever M says, A is not convinced: } R(x, y) \text{ is false for} \\ &\quad \text{any } y. \end{aligned}$$

Moreover, the proof y should have polynomial (in $|x|$) length, otherwise **A** cannot check $R(x, y)$ in polynomial (in $|x|$) time.

In this way, we arrive precisely at Definition 3.2.

3.2. Reducibility and NP-completeness. The notion of reducibility allows us to verify that a predicate is at least as difficult as some other predicate.

Definition 3.4 (Karp reducibility). A predicate L_1 is *reducible* to a predicate L_2 if there exists a function $f \in P$ such that $L_1(x) = L_2(f(x))$ for any input string x .

We say that f *reduces* L_1 to L_2 . Notation: $L_1 \propto L_2$.

Karp reducibility is also called “polynomial reducibility” (or just “reducibility”).

Lemma 3.2. Let $L_1 \propto L_2$. Then

$$(a) \ L_2 \in P \Rightarrow L_1 \in P;$$

- (b) $L_1 \notin P \Rightarrow L_2 \notin P$;
(c) $L_2 \in NP \Rightarrow L_1 \in NP$.

Proof. To prove (a) let us note that $|f(x)| = \text{poly}(|x|)$ for any $f \in P$. Therefore, we can decide $L_1(x)$ in polynomial time as follows: we compute $f(x)$ and then compute $L_2(f(x))$.

Part (b) follows from (a).

Part (c) is also simple. It can be explained in various ways. Using the Arthur–Merlin metaphor, we say that Merlin communicates to Arthur a proof that $L_2(f(x))$ is true (if it is true). Then Arthur computes $f(x)$ by himself and checks whether $L_2(f(x))$ is true, using Merlin’s proof.

Using Definition 3.2, we can explain the same thing as follows:

$$\begin{aligned} L_1(x) &\Leftrightarrow L_2(f(x)) \Leftrightarrow \exists y \left((|y| < q(|f(x)|)) \wedge R(f(x), y) \right) \\ &\Leftrightarrow \exists y \left((|y| < r(|x|)) \wedge R'(x, y) \right). \end{aligned}$$

Here $R'(x, y)$ stands for $(|y| < q(|f(x)|)) \wedge R(f(x), y)$, and $r(n) = q(h(n))$, where $h(n)$ is a polynomial bound for the time needed to compute $f(x)$ for any string x of length n (and, therefore, $|f(x)| \leq h(|x|)$ for any x). \square

Definition 3.5. A predicate $L \in NP$ is NP-*complete* if any predicate in NP is reducible to it.

If some NP-complete predicate can be computed in time $T(n)$, then any NP-predicate can be computed in time $\text{poly}(n) + T(\text{poly}(n))$. Therefore, if some NP-complete predicate belongs to P, then $P = NP$. Put it this way: if $P \neq NP$ (which is probably true), then no NP-complete predicate belongs to P.

If we measure running time “up to a polynomial”, then we can say that NP-complete predicates are the “most difficult” ones in NP.

The key result in computational complexity says that NP-complete predicates do exist. Here is one of them, called *satisfiability*: $SAT(x)$ means that x is a propositional formula (containing Boolean variables and operations \neg , \wedge , and \vee) that is satisfiable, i.e., true for some values of the variables.

Theorem 3.3 (Cook, Levin).

- (1) $SAT \in NP$;
- (2) SAT is NP-complete.

Corollary. If $SAT \in P$, then $P = NP$.

Proof of Theorem 3.3. (1) To convince Arthur that a formula is satisfiable, Merlin needs only to show him the values of the variables that make it true. Then Arthur can compute the value of the whole formula by himself.

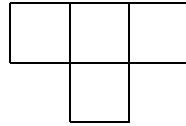
(2) Let $L(x)$ be an NP-predicate and

$$L(x) = \exists y \left((|y| < q(|x|)) \wedge R(x, y) \right)$$

for some polynomial q and some predicate R decidable in polynomial time.

We need to prove that L is reducible to SAT . Let M be a Turing machine that computes R in polynomial time. Consider the computation table (see the proof of Theorem 2.2) for M working on some input $x\#y$. We will use the same variables as in the proof of Theorem 2.2. These variables encode the contents of cells in the computation table.

Now we write a formula that says that values of variables form an encoding of a successful computation (with answer “yes”), starting with the input $x\#y$. To form a valid computation table, values should obey some local rules for each four cells configured as follows:



These local rules can be written as formulas in $4t$ variables (if t variables are needed to encode one cell). We write the conjunction of these formulas and add formulas saying that the first line contains the input string x followed by $\#$ and some binary string y , and that the last line contains the answer “yes”.

The satisfying assignment for our formula will be an encoding of a successful computation of M on input $x\#y$ (for some binary string y). On the other hand, any successful computation that uses at most S tape cells and requires at most T steps (where $T \times S$ is the size of the computation table that is encoded) can be transformed into a satisfying assignment.

Therefore, if we consider a computational table that is large enough to contain the computation of $R(x, y)$ for any y such that $|y| < q(|x|)$, and write the corresponding formula as explained above, we get a polynomial-size formula that is satisfiable if and only if $L(x)$ is true. Therefore L is reducible to SAT . \square

Other examples of NP-complete problems (predicates) can be obtained using the following lemma.

Lemma 3.4. *If $SAT \propto L$ and $L \in \text{NP}$, then L is NP-complete. More generally, if L_1 is NP-complete, $L_1 \propto L_2$, and $L_2 \in \text{NP}$, then L_2 is NP-complete.*

Proof. The reducibility relation is transitive: if $L_1 \propto L_2$ and $L_2 \propto L_3$, then $L_1 \propto L_3$. (Indeed, the composition of two functions from P belongs to P).

According to the hypothesis, any NP-problem is reducible to L_1 , and L_1 is reducible to L_2 . Therefore any NP-problem is reducible to L_2 . \square

Now let us consider the satisfiability problem restricted to 3-CNF. Recall that a CNF (conjunctive normal form) is a conjunction of *clauses*; each clause is a disjunction of literals; each literal is either a variable or a negation of a variable. If each clause contains at most three literals, we get a 3-CNF.

By 3-SAT we denote the following predicate:

$$3\text{-SAT}(x) = x \text{ is a satisfiable 3-CNF.}$$

Evidently, 3-SAT is reducible to SAT (because any 3-CNF is a formula). The next theorem shows that the reverse is also true: SAT is reducible to 3-SAT. Therefore 3-SAT is NP-complete (by Lemma 3.4).

Theorem 3.5. $SAT \propto 3\text{-SAT}$.

Proof. For any propositional formula (and even for any circuit over the standard basis $\{AND, OR, NOT\}$), we construct a 3-CNF that is satisfiable if and only if the given formula is satisfiable (the given circuit produces output 1 for some input).

Let x_1, \dots, x_n be input variables of the circuit, and let y_1, \dots, y_s be auxiliary variables (see the definition of a circuit). Each assignment involves at most three variables (1 on the left-hand side, and 1 or 2 on the right-hand side).

Now we construct a 3-CNF that has variables $x_1, \dots, x_n, y_1, \dots, y_s$ and is true if and only if the values of all y_j are correctly computed (i.e., they coincide with the right-hand sides of the assignments) and the last variable is true. To this end, we replace each assignment by an equivalence (of Boolean expressions) and represent this equivalence as a 3-CNF:

$$\begin{aligned} (y \Leftrightarrow (x_1 \vee x_2)) &= (x_1 \vee x_2 \vee \neg y) \wedge (\neg x_1 \vee x_2 \vee y) \wedge (x_1 \vee \neg x_2 \vee y) \\ &\quad \wedge (\neg x_1 \vee \neg x_2 \vee y), \\ (y \Leftrightarrow (x_1 \wedge x_2)) &= (x_1 \vee x_2 \vee \neg y) \wedge (\neg x_1 \vee x_2 \vee \neg y) \wedge (x_1 \vee \neg x_2 \vee \neg y) \\ &\quad \wedge (\neg x_1 \vee \neg x_2 \vee y), \\ (y \Leftrightarrow \neg x) &= (x \vee y) \wedge (\neg x \vee \neg y). \end{aligned}$$

Finally, we take the conjunction of all these 3-CNF's and the variable y_s (the latter represents the condition that the output of the circuit is 1).

Let us assume that the resulting 3-CNF is satisfied by some $x_1, \dots, x_n, y_1, \dots, y_s$. If we plug the same values of x_1, \dots, x_n into the circuit, then the auxiliary variables will be equal to y_1, \dots, y_s , so the circuit output will be $y_s = 1$. Conversely, if the circuit produces 1 for some inputs, then the

3-CNF is satisfied by the same values of x_1, \dots, x_n and appropriate values of the auxiliary variables.

So our transformation (of a circuit into a 3-CNF) is indeed a reduction of *SAT* to 3-*SAT*. \square

Here is another simple example of reduction.

ILP (integer linear programming). Given a system of linear inequalities with integer coefficients, is there an integer solution? (In other words, is the system consistent?)

In this problem the input is the coefficient matrix and the vector of the right-hand sides of the inequalities. It is not obvious that *ILP* \in NP. Indeed, the solution might exist, but Merlin might not be able to communicate it to Arthur because it is not immediately clear that the number of bits needed to represent the solution is polynomial.

However, it is in fact true that, if a system of inequalities with integer coefficients has an integer solution, then it has an integer solution whose binary representation has size bounded by a polynomial in the bit size of the system; see [55, vol. 2, §17.1]. Therefore, *ILP* is in NP.

Now we reduce 3-*SAT* to *ILP*. Assume that a 3-CNF is given. We construct a system of inequalities that has integer solutions if and only if the 3-CNF is satisfiable. For each Boolean variable x_i we consider an integer variable p_i . The negation $\neg x_i$ corresponds to the expression $1 - p_i$. Each clause $X_j \vee X_k \vee X_m$ (where X_* are literals) corresponds to the inequality $P_j + P_k + P_m \geq 1$, where P_j, P_k, P_m are the expressions corresponding to X_j, X_k, X_m . It remains to add the inequalities $0 \leq p_i \leq 1$ for all i , and we get a system whose solutions are satisfying assignments for the given 3-CNF.

Remark 3.5. If we do not require the solution to be integer-valued, we get the standard linear programming problem. Polynomial algorithms for the solution of this problem (due to Khachiyan and Karmarkar) are described, e.g., in [55, vol. 1, §§13, 15.1].

An extensive list of NP-complete problems can be found in [28]. Usually NP-completeness is proved by some reduction. Here are several examples of NP-complete problems.

3-COLORING. For a given graph G determine whether it admits a 3-coloring. (By a 3-coloring we mean coloring of the vertices with 3 colors such that each edge has endpoints of different colors.)

(It turns out that a similar 2-coloring problem can be solved in polynomial time.)

CLIQUE. For a graph G and an integer k determine whether the graph has a k -clique (a set of k vertices such that every two of its elements are connected by an edge).

Problems

[3] **3.1.** Prove that one can check the satisfiability of a 2-CNF (a conjunction of disjunctions, each containing two literals) in polynomial time.

[2] **3.2.** Prove that the problem of the existence of an Euler cycle in an undirected graph (an Euler cycle is a cycle that traverses each edge exactly once) belongs to P.

[1!] **3.3.** Suppose we have an NP-*oracle* — a magic device that can immediately solve any instance of the SAT problem for us. In other words, for any propositional formula the oracle tells whether it is satisfiable or not. Prove that there is a polynomial-time algorithm that finds a satisfying assignment to a given formula by making a polynomial number of queries to the oracle. (A similar statement is true for the Hamiltonian cycle: finding a Hamiltonian cycle in a graph is at most polynomially harder than checking for its existence.)

[3!] **3.4.** There are n boys and n girls. It is known which boys agree to dance with which girls and vice versa. We want to know whether there exists a *perfect matching* (the boys and the girls can dance in pairs so that everybody is satisfied). Prove that this problem belongs not only to NP (which is almost evident), but also to P.

3.5. Construct

[2!] (a) a polynomial reduction of the 3-SAT problem to the clique problem;

[3] (b) a polynomial reduction of 3-SAT to CLIQUE that conserves the number of solutions (if a 3-CNF F is transformed into a graph H and an integer k , then the number of satisfying assignments for F equals the number of k -cliques in H).

3.6. Construct

[2!] (a) a polynomial reduction of 3-SAT to 3-COLORING;

[3] (b) the same as for (a), with the additional requirement that the number of satisfying assignments is one sixth of the number of 3-colorings of the corresponding graph.

[2] **3.7.** A *tile* is a (1×1) -square whose sides are marked with letters. We want to cover an $(n \times n)$ -square with n^2 tiles; it is known which letters are allowed at the boundary of the $n \times n$ -square and which letters can be adjacent.

The tiling problem requires us to find, for a given set of tile types (containing at most $\text{poly}(n)$ types) and for given restrictions, whether or not there exists a tiling of the $(n \times n)$ -square.

Prove that the tiling problem is NP-complete.

[1] **3.8.** Prove that the predicate “ x is the binary representation of a composite integer” belongs to NP.

[3] **3.9.** Prove that the predicate “ x is the binary representation of a prime integer” belongs to NP.

4. Probabilistic algorithms and the class BPP

4.1. Definitions. Amplification of probability. A probabilistic Turing machine (PTM) is somewhat similar to a nondeterministic one; the difference is that choice is produced by coin tossing, not by guessing. More formally, some (state, symbol) combinations have two associated actions, and the choice between them is made probabilistically. Each instance of this choice is controlled by a random bit. We assume that each random bit is 0 or 1 with probability $1/2$ and that different random bits are independent.

(In fact we can replace $1/2$ by, say, $1/3$ and get almost the same definition; the class BPP (see below) remains the same. However, if we replace $1/2$ by some noncomputable real p , we get a rather strange notion which is better avoided.)

For a given input string a PTM generates not a unique output string, but a probability distribution on the set of all strings (different values of the random bits lead to different computation outputs, and each possible output has a certain probability).

Definition 4.1. Let ε be a constant such that $0 < \varepsilon < 1/2$. A predicate L belongs to the class BPP if there exist a PTM M and a polynomial $p(n)$ such that the machine M running on input string x always terminates after at most $p(|x|)$ steps, and

$$L(x) = 1 \Rightarrow M \text{ gives the answer “yes” with probability } \geq 1 - \varepsilon;$$

$$L(x) = 0 \Rightarrow M \text{ gives the answer “no” with probability } \leq \varepsilon.$$

In this definition the admissible error probability ε can be, say, 0.49 or 10^{-10} — the class BPP will remain the same. Why? Assume that the PTM has probability of error at most $\varepsilon < 1/2$. Take k copies of this machine, run them all for the same input (using independent random bits) and let them vote. Formally, what we do is applying the majority function MAJ (see Problem 2.15) to k individual outputs. The “majority opinion” will be

wrong with probability

$$\begin{aligned}
 p_{\text{error}} &\leq \sum_{\substack{S \subseteq \{1, \dots, k\}, \\ |S| \leq k/2}} (1 - \varepsilon)^{|S|} \varepsilon^{k-|S|} \\
 (4.1) \quad &= ((1 - \varepsilon)\varepsilon)^{k/2} \sum_{\substack{S \subseteq \{1, \dots, k\}, \\ |S| \leq k/2}} \left(\frac{\varepsilon}{1 - \varepsilon} \right)^{k/2 - |S|} \\
 &< \left(\sqrt{(1 - \varepsilon)\varepsilon} \right)^k 2^k = \lambda^k, \quad \text{where } \lambda = 2\sqrt{\varepsilon(1 - \varepsilon)} < 1.
 \end{aligned}$$

If k is big enough, the effective error probability will be as small as we wish. This is called *amplification of probability*. To make the quantity p_{error} smaller than a given number ε' , we need to set $k = \Theta(\log(1/\varepsilon'))$. (Since ε and ε' are constants, k does not depend on the input. Even if we require that $\varepsilon' = \exp(-p(n))$ for an arbitrary polynomial p , the composite TM still runs in polynomial time.)

Let us we give an equivalent definition of the class BPP using predicates in two variables (this definition is similar to Definition 3.2).

Definition 4.2. A predicate L belongs to BPP if there exist a polynomial p and a predicate R , decidable in polynomial time, such that

$$\begin{aligned}
 L(x) = 1 &\Rightarrow \text{the fraction of strings } r \text{ of length } p(|x|) \text{ satisfying } R(x, r) \\
 &\quad \text{is greater than } 1 - \varepsilon; \\
 L(x) = 0 &\Rightarrow \text{the fraction of strings } r \text{ of length } p(|x|) \text{ satisfying } R(x, r) \\
 &\quad \text{is less than } \varepsilon.
 \end{aligned}$$

Theorem 4.1. *Definitions 4.1 and 4.2 are equivalent.*

Proof. *Definition 4.1 \Rightarrow Definition 4.2.* Let $R(x, r)$ be the following predicate: “ M says ‘yes’ for the input x using r_1, \dots, r_{p_n} as the random bits” (we assume that a coin is tossed at each step of M). It is easy to see that the requirements of Definition 4.2 are satisfied.

Definition 4.2 \Rightarrow Definition 4.1. Assume that p and R are given. Consider a PTM that (for input x) randomly chooses a string r of length $p(|x|)$, making $p(|x|)$ coin tosses, and then computes $R(x, r)$. This machine satisfies Definition 4.1 (with a different polynomial p' instead of p). \square

Definition 4.2 is illustrated in Figure 4.1. We represent a pair (x, y) of strings as a point and draw the set $S = \{(x, y) : (|y| = p(|x|)) \wedge R(x, y)\}$. For each x we consider the x -section of S defined as $S_x = \{y : (x, y) \in S\}$. The set S is rather special in the sense that, for any x , either S_x is large (contains at least $1 - \varepsilon$ fraction of all strings of length $p(|x|)$) or is small

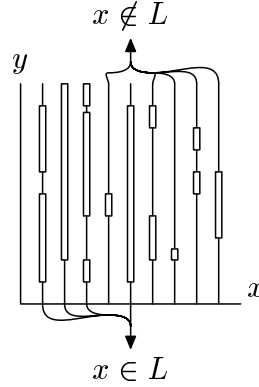


Fig. 4.1. The characteristic set of the predicate $R(x, y)$. Vertical lines represent the sets S_x .

(contains at most ε fraction of them). Therefore all values of x are divided into two categories: for one of them $L(x)$ is true and for the other $L(x)$ is false.

Remark 4.1. Probabilistic Turing machines (unlike nondeterministic ones, which depend on almighty Merlin for guessing the computational path) can be considered as real computing devices. Physical processes like the Nyquist noise or radioactive decay are believed to provide sources of random bits; in the latter case it is guaranteed by the very nature of quantum mechanics.

4.2. Primality testing. A classic example of a BPP problem is checking whether a given integer q (represented by $n = \lceil \log_2 q \rceil$ bits) is prime or not. We call this problem PRIMALITY. We will describe a probabilistic primality test, called *Miller–Rabin test*. For reader’s convenience all necessary facts from number theory are given in Appendix A.

4.2.1. Main idea. We begin with a much simpler “Fermat test”, though its results are generally inconclusive. It is based on Fermat’s little theorem (Theorem A.9), which says that

$$\text{if } q \text{ is prime, then } a^{q-1} \equiv 1 \pmod{q} \text{ for } x \in \{1, \dots, q-1\}.$$

We may regard a as a $(\text{mod } q)$ -residue and simply write $a^{q-1} = 1$, assuming that arithmetic operations are performed with residues rather than integers.

So, the test is this: we pick a random a and check whether $a^{q-1} = 1$. If this is true, then q *may* be a prime; but if this is false, then q is *not* a prime. Such a can be called a *witness* saying that a is composite. This kind of evidence is indirect (it does not give us any factor of q) but usually easy to find: it often suffices to check $a = 2$. But we will do a better job if we sample a from the uniform distribution over the set $\{1, \dots, q-1\}$ (i.e., each element of this set is taken with probability $1/(q-1)$).

Suppose q is composite. We want to know if the test actually shows that with nonnegligible probability. Consider two cases.

- 1) $\gcd(a, q) = d \neq 1$. Then $a^{q-1} \equiv 0 \not\equiv 1 \pmod{d}$, therefore $a^{q-1} \not\equiv 1 \pmod{q}$. The test detects that q is composite. Unfortunately, the probability to get such an a is usually small.
- 2) $\gcd(a, q) = 1$, i.e. $a \in (\mathbb{Z}/q\mathbb{Z})^*$ (where $(\mathbb{Z}/q\mathbb{Z})^*$ denotes the group of invertible \pmod{q} -residues). This is the typical case; let us consider it more closely.

Lemma 4.2. *If $a^{q-1} \neq 1$ for some element $a \in (\mathbb{Z}/q\mathbb{Z})^*$, then the Fermat test detects the compositeness of q with probability $\geq 1/2$.*

Proof. Let $G = (\mathbb{Z}/q\mathbb{Z})^*$. For any integer k define the following set:

$$(4.2) \quad G_{(m)} = \{x \in G : x^m = 1\}.$$

This is a subgroup in G (due to the identity $a^m b^m = (ab)^m$ for elements of an Abelian group). If $a^{q-1} \neq 1$ for some a , then $a \notin G_{(q-1)}$, therefore $G_{(q-1)} \neq G$. By Lagrange's theorem, the ratio $|G|/|G_{(m)}|$ is an integer, hence $|G|/|G_{(m)}| \geq 2$. It follows that $a^{q-1} \neq 1$ for at least half of $a \in (\mathbb{Z}/q\mathbb{Z})^*$. And, as we already know, $a^{q-1} \neq 1$ for all $a \notin (\mathbb{Z}/q\mathbb{Z})^*$. \square

Is it actually possible that q is composite but $a^{q-1} = 1$ for all invertible residues a ? Such numbers q are rare, but they exist (they are called *Carmichael numbers*). Example: $q = 561 = 3 \cdot 11 \cdot 17$. Note that the numbers $3 - 1$, $11 - 1$ and $17 - 1$ divide $q - 1$. Therefore $a^{q-1} = 1$ for any $a \in (\mathbb{Z}/q\mathbb{Z})^* \cong \mathbb{Z}_{3-1} \times \mathbb{Z}_{11-1} \times \mathbb{Z}_{17-1}$.

We see that the Fermat test alone is not sufficient to detect a composite number. The Miller–Rabin test uses yet another type of witnesses for the compositeness: if $b^2 \equiv 1 \pmod{q}$, and $b \not\equiv \pm 1 \pmod{q}$ for some b , then q is composite. Indeed, in this case $b^2 - 1 = (b - 1)(b + 1)$ is a multiple of q but $b - 1$ and $b + 1$ are not, therefore q has nontrivial factors in common with both $b - 1$ and $b + 1$.

4.2.2. Required subroutines and their complexity. Addition (or subtraction) of n -bit integers is done by an $O(n)$ -size circuit; multiplication and division are performed by $O(n^2)$ -size circuits. These estimates refer to the standard algorithms learned in school, though they are not the most efficient for large integers. In the solutions to Problems 2.12, 2.13 and 2.14 we described alternative algorithms, which are much better in terms of the circuit depth, but slightly worse in terms of the size. If only the size is important, the standard addition algorithm is optimal, but the ones for the multiplication and division are not. For example, an $O(n \log n \log \log n)$ -size circuit for the multiplication exists; see [5, Sec. 7.5] or [43, vol. 2, Sec. 4.3.3].

Euclid's algorithm for $\gcd(a, b)$ has complexity $O(n^3)$, but there is also a so-called "binary" gcd algorithm (see [43, vol. 2, Sec. 4.5.2]) of complexity $O(n^2)$. It does not solve the equation $xa + yb = \gcd(a, b)$, though.

We will also use modular arithmetic. It is clear that the addition of $(\text{mod } q)$ -residues is done by a circuit of size $O(n)$, whereas modular multiplication can be reduced to integer multiplication and division; therefore it is performed by a circuit of size $O(n^2)$ (by the standard technique). To invert a residue $a \in (\mathbb{Z}/q\mathbb{Z})^*$, we need to find an integer x such that $xa \equiv 1 \pmod{q}$, i.e., $xa + yq = 1$. This is done by extended Euclid's algorithm, which has complexity $O(n^3)$.

It is also possible to compute $(a^m \text{ mod } q)$ by a polynomial time algorithm. (Note that we speak about an algorithm that is polynomial in the length n of its input (a, m, q) ; therefore performing m multiplications is out of the question. Note also that the size of the integer a^m is exponential.) But we can compute $(a^{2^k} \text{ mod } q)$ for $k = 1, 2, \dots, \lfloor \log_2 m \rfloor$ by repeated squaring, applying the $(\text{mod } q)$ reduction at each step. Then we multiply some of the results in such a way that the powers, i.e., the numbers 2^k , add to m (using the binary representation of m). This takes $O(\log m) = O(n)$ modular multiplications, which translates to circuit size $O(n^3)$.

4.2.3. The algorithm. Assume that a number q is given.

Step 1. If q is even (and $q \neq 2$), then q is composite. If q is odd, we proceed to Step 2.

Step 2. Let $q - 1 = 2^k l$, where $k > 0$, and l is odd.

Step 3. We choose a random $a \in \{1, \dots, q - 1\}$.

Step 4. We compute $a^l, a^{2l}, a^{4l}, \dots, a^{2^{kl}} = a^{q-1}$ modulo q .

Test 1. If $a^{q-1} \neq 1$ (where modular arithmetic is assumed), then q is composite.

Test 2. If the sequence $a^l, a^{2l}, \dots, a^{2^{kl}}$ (Step 4) contains a 1 that is preceded by anything except ± 1 , then q is composite. In other words, if there exists j such that $a^{2^j l} \neq \pm 1$ but $a^{2^{j+1} l} = 1$, then q is composite.

In all other cases the algorithm says that " q is prime" (though in fact it is not guaranteed).

Theorem 4.3. *If q is prime, then the algorithm always (with probability 1) gives the answer "prime".*

If q is composite, then the answer "composite" is given with probability at least $1/2$.

Remark 4.2. To get a probabilistic algorithm in sense of Definition 4.1, we repeat this test twice: the probability of an error (a composite number being undetected twice) is at most $1/4 < 1/2$.

Proof of Theorem 4.3. As we have seen, the algorithm always gives the answer “prime” for prime q .

Assume that q is composite (and odd). If $\gcd(a, q) > 1$ then Test 1 shows that q is composite. So, we may assume that a is uniformly distributed over the group $G = (\mathbb{Z}/q\mathbb{Z})^*$. We consider two major cases.

Case A: $q = p^\alpha$, where p is an odd prime, and $\alpha > 1$. We show that there is an invertible (mod q)-residue x such that $x^{q-1} \neq 1$, namely $x = (1 + p^{\alpha-1}) \bmod q$. Indeed, $x^{-1} = (1 - p^{\alpha-1}) \bmod q$, and⁴

$$\begin{aligned} x^{q-1} &\equiv (1 + p^{\alpha-1})^{q-1} = 1 + (q-1)p^{\alpha-1} + \text{higher powers of } p \\ &\equiv 1 - p^{\alpha-1} \not\equiv 1 \pmod{q}. \end{aligned}$$

Therefore Test 1 detects the compositeness of q with probability $\geq 1/2$ (due to Lemma 4.2).

Case B: q has at least two distinct prime factors. Then $q = uv$, where u and v are odd numbers, $u, v > 1$, and $\gcd(u, v) = 1$. The Chinese remainder theorem (Theorem A.5) says that the group $G = (\mathbb{Z}/q\mathbb{Z})^*$ is isomorphic to the direct product $U \times V$, where $U = (\mathbb{Z}/u\mathbb{Z})^*$ and $V = (\mathbb{Z}/v\mathbb{Z})^*$, and that each element $x \in G$ corresponds to the pair $((x \bmod u), (x \bmod v))$.

For any m we define the following subgroup (cf. formula (4.2)):

$$G^{(m)} = \{x^m : x \in G\} = \text{Im } \varphi_m, \quad \text{where } \varphi_m : x \mapsto x^m.$$

Note that $G^{(m)} = \{1\}$ if and only if $G_{(m)} = G$; this is yet another way to formulate the condition that $a^m = 1$ for all $a \in G$. Also note that if a is uniformly distributed over G , then a^m is uniformly distributed over $G^{(m)}$. Indeed, the map $\varphi_m : x \mapsto x^m$ is a group homomorphism; therefore the number of pre-images is the same for all elements of $G^{(m)}$. It is clear that $G^{(m)} \cong U^{(m)} \times V^{(m)}$. Now we have two subcases.

Case 1. $U^{(q-1)} \neq \{1\}$ or $V^{(q-1)} \neq \{1\}$. This condition implies that $G^{(q-1)} \neq \{1\}$, so that Test 1 detects q being composite with probability at least $1/2$.

Case 2. $U^{(q-1)} = \{1\}$ and $V^{(q-1)} = \{1\}$. In this case Test 1 is always passed, so we have to study Test 2. Let us define two sequences of subgroups:

$$U^{(l)} \supseteq U^{(2l)} \supseteq \dots \supseteq U^{(2^k l)} = \{1\}, \quad V^{(l)} \supseteq V^{(2l)} \supseteq \dots \supseteq V^{(2^k l)} = \{1\}.$$

⁴A similar argument is used to prove that the group $(\mathbb{Z}/p^\alpha\mathbb{Z})^*$ is cyclic; see Theorem A.11.

Note that $U^{(l)} \neq \{1\}$ and $V^{(l)} \neq \{1\}$. Specifically, both $U^{(l)}$ and $V^{(l)}$ contain the residues that correspond to -1 . Indeed, both U and V contain -1 , and $(-1)^l = -1$ since l is odd.

Going from right to left, we find the first place where one of the sets $U^{(m)}, V^{(m)}$ contains an element different from 1. In other words, we find $t = 2^s l$ such that $0 \leq s < k$, $U^{(2^t)} \neq \{1\}$, $V^{(2^t)} \neq \{1\}$, and either $U^{(t)} \neq \{1\}$ or $V^{(t)} \neq \{1\}$.

We will prove that Test 2 shows (with probability at least $1/2$) that q is composite. By our assumption $a^{2^t} = 1$, so we need to know the probability of the event $a^t \neq \pm 1$. Let us consider two possibilities.

Case 2a. One of the sets $U^{(t)}, V^{(t)}$ equals $\{1\}$ (for example, let $U^{(t)} = \{1\}$). This means that for any a the pair $((a^t \bmod u), (a^t \bmod v))$ has 1 as the first component. Therefore $a^t \neq -1$, since -1 is represented by the pair $(-1, -1)$.

At the same time, $V^{(t)} \neq \{1\}$; therefore the probability that a^t has 1 in the second component is at most $1/2$ (by Lagrange's theorem; cf. proof of Lemma 4.2). Thus Test 2 says “composite” with probability at least $1/2$.

Case 2b. Both sets $U^{(t)}$ and $V^{(t)}$ contain at least two elements: $|U^{(t)}| = c \geq 2$, $|V^{(t)}| = d \geq 2$. In this case a^t has 1 in the first component with probability $1/c$ (there are c equiprobable possibilities) and has 1 in the second component with probability $1/d$. These events are independent due to the Chinese remainder theorem, so the probability of the event $a^t = 1$ is $1/cd$. For similar reasons the probability of the event $a^t = -1$ is either 0 or $1/cd$. In any case the probability of the event $a^t = \pm 1$ is at most $2/cd \leq 1/2$. \square

4.3. BPP and circuit complexity.

Theorem 4.4. $\text{BPP} \subseteq \text{P/poly}$.

Proof. Let $L(x)$ be a BPP-predicate, and M a probabilistic TM that decides $L(x)$ with probability at least $1 - \varepsilon$. By running M repeatedly we can decrease the error probability. Recall that a polynomial number of repetitions leads to the exponential decrease. Therefore we can construct a polynomial probabilistic TM M' that decides $L(x)$ with error probability less than $\varepsilon' < 1/2^n$ for inputs x of length n .

The machine M' uses a random string r (one random bit for each step). For each input x of length n , the fraction of strings r that lead to an incorrect answer is less than $1/2^n$. Therefore the overall fraction of “bad” pairs (x, r) among all such pairs is less than $1/2^n$. [If one represents the set of all pairs (x, r) as a unit square, the “bad” subset has area $< 1/2^n$.] It follows that there exists $r = r_*$ such that the fraction of bad pairs (x, r) is less than

$1/2^n$ among all pairs with $r = r_*$. However, there are only 2^n such pairs (corresponding to 2^n possibilities for x). The only way the fraction of bad pairs can be smaller than $1/2^n$ is that there are no bad pairs at all!

Thus we conclude that there is a particular string r_* that produces correct answers for all x of length n .

The machine M' can be transformed into a polynomial-size circuit with input (x, r) . Then we fix the value of r (by setting $r = r_*$) and obtain a polynomial-size circuit with input x that decides $L(x)$ for all n -bit strings x . \square

This is a typical nonconstructive existence proof: we know that a “good” string r_* exists (by probability counting) but have no means of finding it, apart from exhaustive search.

Remark 4.3. It might well be the case that $\text{BPP} = \text{P}$. Let us explain briefly the motivation of this conjecture and why it is hard to prove.

Speaking about proved results, it is clear that $\text{BPP} \subseteq \text{PSPACE}$. Indeed, the algorithm that counts all values of the string r that lead to the answer “yes” runs in polynomial space. Note that the running time of this algorithm is $2^N \text{poly}(n)$, where $N = |r|$ is the number of random bits.

On the other hand, there is empirical evidence that probabilistic algorithms usually work well with pseudo-random bits instead of truly random ones. So attempts have been made to construct a mathematical theory of pseudo-random numbers. The general idea is as follows. A *pseudo-random generator* is a function $g : \mathbb{B}^l \rightarrow \mathbb{B}^L$ which transforms short truly random strings (of length l , which can be as small as $O(\log L)$) into much longer pseudo-random strings (of length L). “Pseudo-random” means that any computational device with limited resources (say, any Boolean circuit of a given size N computing a function $F : \mathbb{B}^L \rightarrow \mathbb{B}$) is unable to distinguish between truly random and pseudo-random strings of length L . Specifically, we require that

$$\left| \Pr[F(g(x)) = 1] - \Pr[F(y) = 1] \right| \leq \delta, \quad x \in \mathbb{B}^l, \ y \in \mathbb{B}^L$$

for some constant $\delta < 1/2$, where x and y are sampled from the uniform distributions. (In this definition the important parameters are l and N , while L should fit the number of input bits of the circuit. For simplicity let us require that $L = N$: it will not hurt if the pseudo-random generator produces some extra bits.)

It is easy to show that pseudo-random generators $g : \mathbb{B}^{O(\log L)} \rightarrow \mathbb{B}^L$ exist: if we choose the function g randomly, it fulfills the above condition with high probability. What we actually need is a sequence of efficiently computable pseudo-random generators $g_L : \mathbb{B}^{l(L)} \rightarrow \mathbb{B}^L$, where $l(L) = O(\log L)$.

If such pseudo-random generators exist, we can use their output instead of truly random bits in any probabilistic algorithm. The definition of pseudo-randomness guarantees that this will work, provided the running time of the algorithm is limited by $c\sqrt{L}$ (for a suitable constant c) and the error probability ε is smaller than $1/2 - \delta$. (With pseudo-random bits the error probability will be $\varepsilon + \delta$, which is still less than $1/2$. The estimate $c\sqrt{L}$ comes from the simulation of a Turing machine by Boolean circuits, see Theorem 2.2.) Thus we decrease the number of needed genuine random bits from L to l . Then we can *derandomize* the algorithm by trying all 2^l possibilities. If $l = O(\log L)$, the resulting computation has polynomial complexity.

We do not know whether efficiently computable pseudo-random generators exist. The trouble is that the definition has to be satisfied for “any Boolean circuit of a given size”; this condition is extremely hard to deal with. But we may try to reduce this problem to a more fundamental one — obtaining lower bounds for the circuit complexity of Boolean functions. Even this idea, which sets the most difficult part of the problem aside, took many years to realize. Much work in this area was done in 1980’s and early 1990’s, but the results were not as strong as needed. Recently there has been dramatic progress leading to more efficient constructions of pseudo-random generators and new derandomization techniques. It has been proved that $\text{BPP} = \text{P}$ if there is an algorithm with running time $\exp(O(n))$ that computes a sequence of functions with circuit complexity $\exp(\Omega(n))$ [32]. We also mention a new work [69] in which pseudo-random generators are constructed from arbitrary hard functions in an optimal (up to a polynomial) way.

5. The hierarchy of complexity classes

Recall that we identify languages (sets of strings) and predicates (and $x \in L$ means $L(x) = 1$).

Definition 5.1. Let A be some class of languages. The dual class $\text{co-}A$ consists of the complements of all languages in A . Formally,

$$L \in \text{co-}A \Leftrightarrow (\mathbb{B}^* \setminus L) \in A.$$

It follows immediately from the definitions that $\text{P} = \text{co-P}$, $\text{BPP} = \text{co-BPP}$, $\text{PSPACE} = \text{co-PSPACE}$.

5.1. Games machines play. Consider a game with two players called White (W) and Black (B). A string x is shown to both players. After that, players alternately choose binary strings: W starts with some string w_1 , B

replies with b_1 , then W says w_2 , etc. Each string has length polynomial in $|x|$. Each player is allowed to see the strings already chosen by his opponent.

The game is completed after some prescribed number of steps, and the referee, who knows x and all the strings and who acts according to a polynomial-time algorithm, declares the winner. In other words, there is a predicate $W(x, w_1, b_1, w_2, b_2, \dots)$ that is true when W is the winner, and we assume that this predicate belongs to P . If this predicate is false, B is the winner (there are no ties). This predicate (together with polynomial bounds for the length of strings and the number of steps) determines the game.

Let us note that in fact the termination rule can be more complicated, but we always assume that the number of moves is bounded by a polynomial. Therefore we can “pad” the game with dummy moves that are ignored by the referee and consider only games where the number of moves is known in advance and is a polynomial in the input length.

Since this game is finite and has no ties, for each x either B or W has a winning strategy. (One can formally prove this using induction over the number of moves.) Therefore, any game determines two complementary sets,

$$\begin{aligned} L_w &= \{x : W \text{ has a winning strategy}\}, \\ L_b &= \{x : B \text{ has a winning strategy}\}. \end{aligned}$$

Many complexity classes can be defined as classes formed by the sets L_w (or L_b) for some classes of games. Let us give several examples.

P : the sets L_w (or L_b) for games of zero length (the referee declares the winner after he sees the input)

NP : the sets L_w for games that are finished after W 's first move. In other words, NP -sets are sets of the form

$$\{x : \exists w_1 W(x, w_1)\}.$$

$co-NP$: the sets L_b for games that are finished after W 's first move. In other words, $co-NP$ -sets are sets of the form

$$\{x : \forall w_1 B(x, w_1)\}$$

(here $B = \neg W$ means that B wins the game.)

Σ_2 : the sets L_w for games where W and B make one move each and then the referee declares the winner. In other words, Σ_2 -sets are sets of the form

$$\{x : \exists w_1 \forall b_1 W(x, w_1, b_1)\}.$$

(W can make a winning move w_1 after which any move b_1 of B makes B lose).

Π_2 : the sets L_b for the same class of games, i.e., the sets of the form

$$\{x : \forall w_1 \exists b_1 B(x, w_1, b_1)\}.$$

...

Σ_k : the sets L_w for games of length k (the last move is made by W if k is odd or by B if k is even), i.e., the sets

$$\{x : \exists w_1 \forall b_1 \dots \mathbf{Q}_k y_k W(x, w_1, b_1, \dots)\}$$

(if k is even, then $\mathbf{Q}_k = \forall$, $y_k = b_{k/2}$; if k is odd, then $\mathbf{Q}_k = \exists$, $y_k = w_{(k+1)/2}$).

Π_k : the sets L_b for the same class of games, i.e., the sets

$$\{x : \forall w_1 \exists b_1 \dots \mathbf{Q}_k y_k B(x, w_1, b_1, \dots)\}$$

(if k is even, then $\mathbf{Q}_k = \exists$, $y_k = b_{k/2}$; if k is odd, then $\mathbf{Q}_k = \forall$, $y_k = w_{(k+1)/2}$).

...

Evidently, complements of Σ_k -sets are Π_k -sets and vice versa: $\Sigma_k = \text{co-}\Pi_k$, $\Pi_k = \text{co-}\Sigma_k$.

Theorem 5.1 (Lautemann [46]). $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$.

Proof. Since $\text{BPP} = \text{co-BPP}$, it suffices to show that $\text{BPP} \subseteq \Sigma_2$.

Let us assume that $L \in \text{BPP}$. Then there exist a predicate R (computable in polynomial time) and a polynomial p such that the fraction $|S_x|/2^N$, where

$$(5.1) \quad S_x = \{y \in \mathbb{B}^N : R(x, y)\}, \quad N = p(|x|),$$

is either large (greater than $1 - \varepsilon$ for $x \in L$) or small (less than ε for $x \notin L$).

To show that $L \in \Sigma_2$, we need to reformulate the property “ X is a large subset of G ” (where G is the set of all strings y of length N) using existential and universal quantifiers.

This could be done if we impose a group structure on G . Any group structure will work if the group operations are polynomial-time computable. For example, we can consider an additive group formed by bit strings of a given length with bit-wise addition modulo 2.

The property that distinguishes large sets from small ones is the following: “several copies of X shifted by some elements cover G ”, i.e.,

$$(5.2) \quad \exists g_1, \dots, g_m \left(\bigcup_i (g_i + X) = G \right),$$

where “+” denotes the group operation. To choose an appropriate value for m , let us see when (5.2) is guaranteed to be true (or false).

It is obvious that condition (5.2) is false if

$$(5.3) \quad m|X| < |G|.$$

On the other hand, (5.2) is true if for independent random $g_1, \dots, g_m \in G$ the probability of the event $\bigcup_i (g_i + X) = G$ is positive; in other words, if $\Pr[\bigcup_i (g_i + X) \neq G] < 1$. Let us estimate this probability.

The probability that a random shift $g + X$ does not contain a fixed element $u \in G$ (for a given X and random g) is $1 - |X|/|G|$. When g_1, \dots, g_m are chosen independently, the corresponding sets $g_1 + X, \dots, g_m + X$ do not cover u with probability $(1 - |X|/|G|)^m$. Summing these probabilities over all $u \in G$, we see that the probability of the event $\bigcup_i (g_i + X) \neq G$ does not exceed $|G|(1 - |X|/|G|)^m$.

Thus condition (5.2) is true if

$$(5.4) \quad |G|(1 - |X|/|G|)^m < 1.$$

Let us now apply these results to the set $X = S_x$ (see formula (5.1)). We want to satisfy (5.3) and (5.4) when $|S_x|/2^N < \varepsilon$ (i.e., $x \notin L$) and when $|S_x|/2^N > 1 - \varepsilon$ (i.e., $x \in L$), respectively. Thus we get the inequalities $\varepsilon m < 1$ and $2^N \varepsilon^m < 1$, which should be satisfied simultaneously by a suitable choice of m . This is not always possible if N and ε are fixed. Fortunately, we have some flexibility in the choice of these parameters. Using “amplification of probability” by repeating the computation k times, we increase N by factor of k , while decreasing ε exponentially. Let the initial value of ε be a constant, and λ given by (4.1). The amplification changes N and ε to $N' = kN$ and $\varepsilon' = \lambda^k$. Thus we need to solve the system

$$\lambda^k m < 1, \quad 2^{kN} \lambda^{km} < 1$$

by adjusting m and k . It is obvious that there is a solution with $m = O(N)$ and $k = O(\log N)$.

We have proved that $x \in L$ is equivalent to the following Σ_2 -condition:

$$\exists g_1, \dots, g_m \forall y \left((|y| = p'(|x|)) \Rightarrow ((y \in g_1 + S'_x) \vee \dots \vee (y \in g_m + S'_x)) \right).$$

Here $p'(n) = kp(|x|)$ (k and m also depend on $|x|$), whereas S'_x is the “amplified” version of S_x .

In other words, we have constructed a game where W names m strings (group elements) g_1, \dots, g_m , and B chooses some string y . If y is covered by some $g_i + S'_x$ (which is easy to check: it means that $y - g_i$ belongs to S'_x), then W wins; otherwise B wins. In this game W has a winning strategy if and only if S'_x is big, i.e., if $x \in L$. \square

5.2. The class PSPACE. This class contains predicates that can be computed by a TM running in polynomial (in the input length) space. The class PSPACE also has a game-theoretic description:

Theorem 5.2. *$L \in \text{PSPACE}$ if and only if there exists a polynomial game such that*

$$L = \{x: W \text{ has a winning strategy for input } x\}.$$

By a polynomial game we mean a game where the number of moves is bounded by a polynomial (in the length of the input), players' moves are strings of polynomial length, and the referee's algorithm runs in polynomial time.

Proof. \Leftarrow We show that a language determined by a game belongs to PSPACE. Let the number of turns be $p(|x|)$. We construct a sequence of machines $M_1, \dots, M_{p(|x|)}$. Each M_k gets a prefix x, w_1, b_1, \dots of the play that includes k moves and determines who has the winning strategy in the remaining game.

The machine $M_{p(|x|)}$ just computes the predicate $W(x, w_1, \dots)$ using referee's algorithm. The machine M_k tries all possibilities for the next move and consults M_{k+1} to determine the final result of the game for each of them. Then M_k gives an answer according to the following rule, which says whether W wins. If it is W's turn, then it suffices to find a single move for which M_{k+1} declares W to be the winner. If it is B's turn, then W needs to win after all possible moves of B.

The machine M_0 says who is the winner before the game starts and therefore decides $L(x)$. Each machine in the sequence $M_1, \dots, M_{p(|x|)}$ uses only a small (polynomially bounded) amount of memory, so that the composite machine runs in polynomial space. (Note that the computation time is exponential since each of the M_k calls M_{k+1} many times.)

\Rightarrow Let M be a machine that decides the predicate L and runs in polynomial space s . We may assume that computation time is bounded by $2^{O(s)}$. Indeed, there are $2^{O(s)}$ different configurations, and after visiting the same configuration twice the computation repeats itself, i.e., the computation becomes cyclic.

[To see why there are at most $2^{O(s)}$ configurations note that configuration is determined by head position (in $\{0, 1, \dots, s\}$), internal state (there are $|Q|$ of them) and the contents of the s cells of the tape ($|A|^s$ possibilities where A is the alphabet of TM); therefore the total number of configurations is $|A|^s \cdot |Q| \cdot s = 2^{O(s)}$.]

Therefore we may assume without loss of generality that the running time of M on input x is bounded by 2^q , where $q = \text{poly}(|x|)$.

In the description of the game given below we assume that TM keeps its configuration unchanged after the computation terminates.

During the game, W claims that M 's result for an input string x is “yes”, and B wants to disprove this. The rules of the game allow W to win if $M(x)$ is indeed “yes” and allow B to win if $M(x)$ is not “yes”.

In his first move W declares the configuration of M after 2^{q-1} steps dividing the computation into two parts. B can choose any of the parts: either the time interval $[0, 2^{q-1}]$ or the interval $[2^{q-1}, 2^q]$. (B tries to catch W by choosing the interval where W is cheating.) Then W declares the configuration of M at the middle of the interval chosen by B and divides this interval into two halves, B selects one of the halves, W declares the configuration of M at the middle, etc.

The game ends when the length of the interval becomes equal to 1. Then the referee checks whether the configurations corresponding to the ends of this interval match (the second is obtained from the first according to M 's rules). If they match, then W wins; otherwise B wins.

If M 's output on x is really “yes”, then W wins if he is honest and declares the actual configuration of M . If M 's output is “no”, then W is forced to cheat: his claim is incorrect for (at least) one of the halves. If B selects this half at each move, then B can finally catch W “on the spot” and win. \square

[2!] **Problem 5.1.** Prove that any predicate $L(x)$ that is recognized by a nondeterministic machine in space $s = \text{poly}(|x|)$ belongs to PSPACE. (A predicate L is recognized by an NTM M in space $s(|x|)$ if for any $x \in L$ there exists a computational path of M that gives the answer “yes” using at most $s(|x|)$ cells and, for each $x \notin L$, no computational path of M ends with “yes”.)

Theorem 5.2 shows that all the classes Σ_k , Π_k are subsets of PSPACE. Relations between these classes are represented by the inclusion diagram in Figure 5.1. The smallest class is P (games of length 0); P is contained in both classes NP and co-NP (which correspond to games of length 1); classes NP and co-NP are contained in Σ_2 and Π_2 (games with two moves) and so on. We get the class PSPACE, allowing the number of moves in a game be polynomial in $|x|$.

We do not know whether the inclusions in the diagram are strict. Computer scientists have been working hard for several decades trying to prove at least something about these classes, but the problem remains open. It is possible that $P = \text{PSPACE}$ (though this seems very unlikely). It is also possible that $\text{PSPACE} = \text{EXPTIME}$, where EXPTIME is the class of languages decidable in time $2^{\text{poly}(n)}$. Note, however, that $P \neq \text{EXPTIME}$ — one can

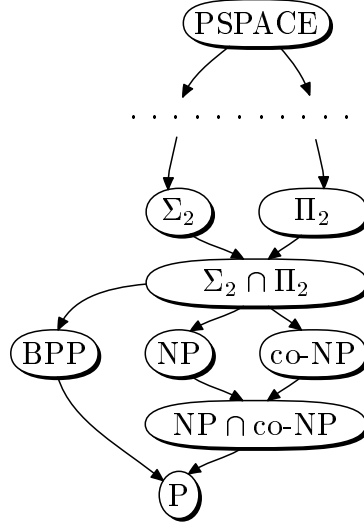


Fig. 5.1. Inclusion diagram for computational classes. An arrow from A to B means that B is a subset of A .

prove this by a rather simple “diagonalization argument” (cf. solution to Problem 1.3).

[3!] **Problem 5.2.** A *Turing machine with oracle for language L* uses a decision procedure for L as an external subroutine (cf. Definition 2.2). The machine has a supplementary *oracle tape*, where it can write strings and then ask the “oracle” whether the string written on the oracle tape belongs to L .

Prove that any language that is decidable in polynomial time by a TM with oracle for some $L \in \Sigma_k$ (or $L \in \Pi_k$) belongs to $\Sigma_{k+1} \cap \Pi_{k+1}$.

The class PSPACE has complete problems (to which any problem from PSPACE is reducible). Here is one of them.

The *TQBF* Problem is given by the predicate

$TQBF(x) \Leftrightarrow x$ is a True Quantified Boolean Formula, i.e., a true statement of type $Q_1 y_1 \dots Q_n y_n F(y_1, \dots, y_n)$, where variables y_i range over $\mathbb{B} = \{0, 1\}$, F is some propositional formula (involving $y_1, \dots, y_n, \neg, \wedge, \vee$), and Q_i is either \forall or \exists .

By definition, $\forall y A(y)$ means $(A(0) \wedge A(1))$ and $\exists y A(y)$ means $(A(0) \vee A(1))$.

Theorem 5.3. *TQBF is PSPACE-complete.*

Proof. We reduce an arbitrary language $L \in \text{PSPACE}$ to *TQBF*. Using Theorem 5.2, we construct a game that corresponds to L . Then we convert a TM that computes the result of the game (a predicate W) into a circuit. Moves of the players are encoded by Boolean variables. Then the existence of the winning strategy for W can be represented by a quantified Boolean

formula

$$\exists w_1^1 \exists w_1^2 \dots \exists w_1^{p(|x|)} \forall b_1^1 \dots \forall b_1^{p(|x|)} \exists w_2^1 \dots \exists w_2^{p(|x|)} \dots S(x, w_1^1, w_1^2, \dots),$$

where $S(\cdot)$ denotes the Boolean function computed by the circuit. (Boolean variables $w_1^1, \dots, w_1^{p(|x|)}$ encode the first move of W , variables $b_1^1, \dots, b_1^{p(|x|)}$ encode B 's answer, $w_2^1, \dots, w_2^{p(|x|)}$ encode the second move of W , etc.)

In order to convert S into a Boolean formula, recall that a circuit is a sequence of assignments $y_i := R_i$ that determine the values of auxiliary Boolean variables y_i . Then we can replace $S(\cdot)$ by a formula

$$\exists y_1, \dots, \exists y_s ((y_1 \Leftrightarrow R_1) \wedge \dots \wedge (y_s \Leftrightarrow R_s) \wedge y_s),$$

where s is the size of the circuit.

After this substitution we obtain a quantified Boolean formula which is true if and only if $x \in L$. \square

Remark 5.1. Note the similarity between Theorem 5.3 (which is about polynomial space computation) and Problems 2.17 and 2.18 (which are basically about poly-logarithmic space computation). Also note that a polynomial-size quantified Boolean formula may be regarded as a polynomial depth circuit (though of very special structure): the \forall and \exists quantifiers are similar to the \wedge and \vee gates. It is not surprising that the solutions are based on much the same ideas. In particular, the reduction from NTM to $TQBF$ is similar to the parallel simulation of a finite-state automaton (see Problem 2.11). However, in the case of $TQBF$ we could afford reasoning at a more abstract level: with greater amount of computational resources we were sure that all bookkeeping constructions could be implemented. This is one of the reasons why “big” computational classes (like PSPACE) are popular among computer scientists, in spite of being apparently impractical. In fact, it is sometimes easier to prove something about big classes, and then scale down the problem parameters while fixing some details.