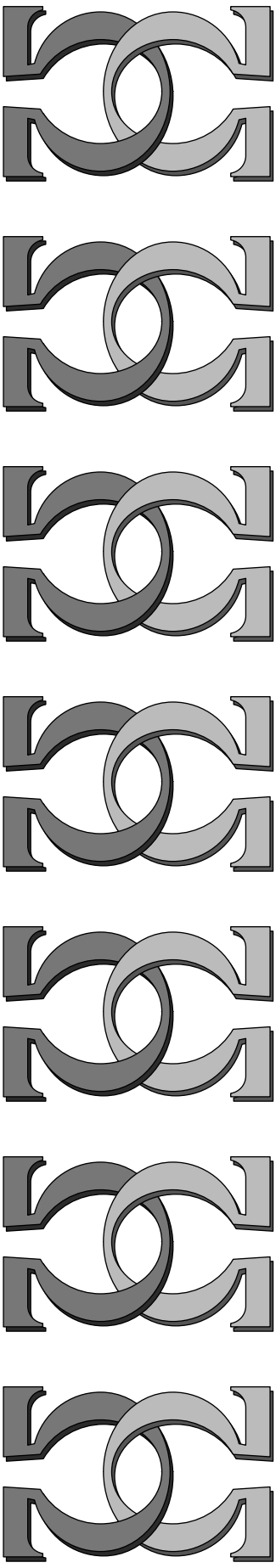

CDMTCS
Research
Report
Series

Complexity: A Language - -Theoretic Point of View

C. Calude, J. Hromkovič
University of Auckland, New Zealand
University of Kiel, Germany

CDMTCS-009
October 1995

Centre for Discrete Mathematics and
Theoretical Computer Science

G. Rozenberg, A. Salomaa (eds.)

HANDBOOK OF FORMAL LANGUAGES–VOLUME II

Cristian Calude and Juraj Hromkovič

COMPLEXITY: A LANGUAGE–THEORETIC POINT OF VIEW

Springer–Verlag

Contents

Chapter 1

COMPLEXITY: A LANGUAGE–THEORETIC POINT OF VIEW

Cristian Calude[1] **and Juraj Hromkovič** [2]

1.1 Introduction

The theory of computation and complexity theory are fundamental parts of current theoretical computer science. They study the borders between possible and impossible in the information processing, quantitative rules governing discrete computations (how much work (computational resources) has to be done (have to be used) and suffices (suffice) to algorithmically solve various computing problems), algorithmical aspects of complexity, optimization, approximation, reducibility, simulation, communication, knowledge representation, information, etc. Historically, theoretical computer science started in the 1930s with the theory of computation (computability theory) giving the exact formal border between algorithmically solvable computing problems and problems which cannot be solved by any program (algorithm). The birth of complexity theory can be set in the 1960s when computers started to be widely used and the inner difficulty of computing problems has started to be investigated. At that time people defined quantitative complexity measures enabling to compare the efficiency of computer programs and to study the computational hardness of computing problems as an inherent property of problems. The abstract part of complexity theory has tried to classify computing problems according to their hardness (computational complexity) while the algorithmic part of complexity theory has dealt with the development of methods for the design of effective algorithms for concrete problems.

The theory of computation and complexity theory provide a variety of concepts, methods, and tools building the fundamentals of theoretical computer science. The goal of this chapter is to concentrate on the intersection of formal language theory and computation (complexity) theory, on the methods developed in formal language theory and used in complexity theory as well as on the complexity of language recognition and generation. An effort in this direction is reasonable because the core formalism used in complexity and computation theory is based on words and languages, the class of algorithmically solvable problems is usually defined as a class of languages, the fundamental complexity classes are defined as classes of languages, etc.

In what follows we assume that the reader is familiar with the elementary notions and concepts of formal language theory (words, languages, automata, Turing machines, grammars and rewriting systems, etc.) and we review the basic concepts, results, and proof methods of the computation and complexity theory using the formalism of formal language theory.

[1]Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand.

[2]Institut of Informatics and Applied Mathematics, University of Kiel, Olshausenstrasse 40, 24098 Kiel, Germany.

This chapter is organized as follows. In Section 2 the fundamentals of computability theory (theory of computation) is given.

Section 3 is devoted to abstract (structural) complexity theory. First the definitions of time and space complexity as basic complexity measures are given and the corresponding complexity classes are defined. Using proof methods from computability and formal language theory, strong hierarchies of complexity measures (more time/space helps to recognize more languages) are proven. The problem of proving nontrivial lower bounds on the complexity of concrete problems is discussed and nondeterminism is used in order to obtain a new insight on the classification of the hardness of computing problems. Finally, probabilistic Turing machines and the corresponding probabilistic complexity classes are introduced.

Section 4 is devoted to program-size (or descriptional) complexity. We begin by contrasting dynamic and descriptional complexities, then revisit the halting problem; random strings and random languages will be introduced and studied. Recursive and regular languages are characterized in terms of descriptive complexity and, at the end, we review a few results—obtained by program-size methods—concerning the problem **P** versus **NP**.

The last section of this chapter is devoted to parallel data processing. Alternating Turing machines are used as a representant of a parallel computing model enabling to relate sequential complexity measures to parallel ones. A further extension to synchronized alternating Turing machines shows the importance of communication facilities in parallel computing. A formal language approach enabling to study and to compare the power of different communication structures as candidates for parallel architectures (interconnection networks) closes this section.

1.2 Theory of Computation

1.2.1 Computing Fallibilities

This section will describe a few tasks which appear to be beyond the capabilities of computers.

From Minimal Art to Minimal Programs

According to Gardner [66], minimal art[3]—painting[4], sculpture[5], music[6] —appears to be *minimal* in at least two senses:

- it requires minimal resources, i.e. time, space, cost, thought, talent, to produce, and
- it has *some*, but rather minimal, aesthetic value.

Let's imagine with [110] that we find ourselves in a large and crowded hall where ten thousand people are talking on a large variety of subjects. The loud hubbub generated by this environment is certainly very rich in information. However, it is totally beyond human feasibility to extract one single item from it. Pushing this experiment to extreme we reach the "white noise" where all sounds that have been made, that are being made or that will be ever made are put together. Similar experiments would consist in

- considering a canvas on which all colours are mixed to the extent that the whole painting becomes a uniform shade of grey, or
- mixing the matter and anti-matter until one reaches the quantum vacuum, or

[3]In painting, minimalism was characterized chiefly by the minimal presence of such standard "artistic" means as form and color and by the use of components that in themselves have no emotive or aesthetic significance. Minimal sculpture is often constructed by others, from the artist's plans, in commonplace industrial materials such as plastic or concrete. Music minimalism is based on the repetition of a musical phrase with subtle, slowly shifting tonalities and a rhythmic structure—if there is one. Minimal art works are not intended to embody any representational or emotional qualities but must be seen simply as what they are. See [5, 137].

[4]P. Mondrian (Composition 2, 1922), R. Tuttle (Silver Picture, 1964).

[5]C. Brâncuşi (Endless Column, 1937-8), C. Andre (Cedar Piece, 1959), Picasso (Chicago statue, 1967).

[6]La Monte Young (Trio for Strings, 1958), T. Riley (In C, 1964), S. Reich (Drumming, 1971), P. Glass (Akhnateon, 1984), J. Adams (Nixon in China, 1987).

- considering a lexicon containing all writings that have been written, that are being written or that will be ever written.

In all these experiments information tends to be so "dense" and "large" that it is impossible to conceive it as a human creation: it reaches the level of *randomness*.[7] What about computers and their "minimal programs"? Any computation can be done in infinitely many different ways. However, the programs of greatest interest are the smallest ones, i.e. the minimal ones.

What is the typical "behaviour" of such a program? Can we "compute" the smallest minimal programs?

To answer the first question we claim that *minimal programs should be random*. But what is a random program? According to the point of view of Algorithmic Information Theory (see [37, 24]), a random program is a program whose minimal program has roughly the same length as the generating program.[8]

Now, assume that x is a minimal program generating y. If x is not random, then there exists a program z generating x which is substantially smaller than x. To conclude, let us consider the program

from z calculate x, then from x calculate y.

This program is only a few letters longer than z, and thus it should be much shorter than x, so x is not minimal.

The answer to the second question is *negative* and the argument is related to the answer of the first question: minimal programs cannot be computed because they are random.

Like minimal art works, minimal programs tend to display an inner "randomness"; in contrast, minimal programs cannot be computed at all while minimal work arts appear to require very little resources.

Word Problems

Suppose that we have fixed a finite set of strings (words) over a fixed alphabet. These strings do not need to have in themselves any meaning, but a meaning will be assigned by considering certain "equalities" between strings. These equalities will be used to derive further equalities by making substitutions of strings from the initial list into other, normally much longer, strings which contain them as "portions". Each portion may be in turn replaced by another portion which is deemed to be equal to it according to the list.

For example, from the list

$$\begin{aligned} ATE &= A \\ CARP &= ME \\ EAT &= AT \\ PAN &= PILLOW \end{aligned}$$

we can derive, by successively substitutions,

LAP=LEAP

as

LAP=LATEP=LEATEP=LEAP.[9]

Is it possible to derive from the string CARPET the string MEAT? A possible way to get a negative answer is to notice that in every equality in our initial list the number of As plus the number of Ws plus the number of Ms is constant in each side. Computing this "invariant" for the strings above we get 1 for CARPET and 2 for MEAT, so we cannot get MEAT from CARPET.

[7] Xenakis [147] says that the amount of information conveyed in sounds gives the real value of music, and Eco [57] observes that a lexicon, no matter how complete or well constructed, has no poetic value.

[8] See Section 1.3.3.

[9] We have used, in order, the first, third, and again first equality.

What about the general decision problem, i.e. one in which we have an arbitrary initial (finite) list of strings, two fixed arbitrary strings x, y and we ask whether we can get from x to y by allowed substitutions? Clearly, by generating quasi-lexicographically all possible finite sequences of strings starting with x and ending with y, and then checking if such a list satisfy the required rules, one can establish equality between strings which are indeed equal. For some lists (e.g. the list displayed above) it is possible to design an algorithm to test whether two arbitrary strings are or are not equal.[10] Is it possible to do this in general? The answer is *negative* and here is an example (discovered by G. S. Tseitin and D. Scott in 1955 and modified by Gardner [65]) of an instance for which there is no single algorithm to test whether, for arbitrary strings x, y we can get from x to y:

$$\begin{aligned} ac &= ca \\ ad &= da \\ bc &= cd \\ bd &= db \\ abac &= abacc \\ eca &= ae \\ edb &= be. \end{aligned}$$

For more information on word problems see [2, 50].

Tilings

Consider a positive integer n and a $2^n \times 2^n$ square grid with only one square removed. Let us define an L-shaped tile to be a figure consisting of three squares arranged in the form of the letter L. Is it possible to cover the square grid with L-shaped tiles? The answer is *affirmative* and here is an inductive argument (see [152]). If $n = 1$, then the grid has the dimension 2×2 and one square has been removed: the figure is exactly an L–shaped tile. Suppose now that for a positive integer n, every $2^n \times 2^n$ square grid in which one square has been removed can be covered with L-shaped tiles. Let us consider a $2^{n+1} \times 2^{n+1}$ grid with one square removed. Cutting this grid in half both vertically and horizontally we get four $2^n \times 2^n$ square subgrids. The missing square comes from one of these four subgrids, so by applying the inductive hypothesis for that subgrid we deduce that it can be covered with L-shaped tiles. To cover the remainder subgrids, first place one L-shaped tile in the center so that it covers one square from each of the remaining subgrids. We have to cover an area which contains every square except one in each of the subgrids, so applying again the inductive hypothesis we get the desired result. Notice that the above proof can be easily converted in an algorithm constructing the required cover.

We can go one further step and ask if it possible to cover the Euclidean plane with polygonal shapes, i.e. we are given a finite number of shapes and we ask whether it is possible to cover the plane *completely, without gaps or overlaps* with just the selected shapes. Choosing only squares, or equilateral triangles, or regular hexagons, the answer is *affirmative*. In all these cases the tilings are *periodic*, in the sense that they are exactly repetitive in two independent directions. However, H. Wang has shown the existence of non-periodic tilings. In 1961 he has addressed the following question: Is there an algorithm for deciding whether or not a given finite set of different polygonal shapes will tile the Euclidean plane? Five years later, R. Berger proven that the answer is *negative*.[11]

The World of Polynomials

We shall use Diophantine equations, that is equations of the form

$$P(x_1, \ldots, x_n) = 0,$$

[10] The reader is encouraged to find such an algorithm.

[11] Berger used a set of 20 426 tiles; R. Robinson was able to reduce this number to six, and R. Penrose to two. See more in [74].

where P is a polynomial with integer coefficients in the variables $x_1, \ldots, x_n$, to define sets of positive integers. To every polynomial $P(x, y_1, y_2, \ldots, y_m)$ with integer coefficients one associates the set

$$D = \{x \in \mathbb{N} \mid P(x, y_1, y_2, \ldots, y_m) = 0, \text{for some } y_1, y_2, \ldots, y_m \in \mathbb{Z}\}.$$

Call a set *Diophantine* if it is of the above form.

For example, the set of composite numbers is Diophantine as it can be written in the form

$$\{x \in \mathbb{N} \mid x = (y+2)(z+2), \text{ for some } y, z \in \mathbb{Z}\}.$$

The language of Diophantine sets permits the use of existential quantifiers (by definition), as well as the logical connectives *and* and *or* (as the system $P_1 = 0$ and $P_2 = 0$ is equivalent to the equation $P_1^2 + P_2^2 = 0$, and the condition $P_1 = 0$ or $P_2 = 0$ can equivalently be written as $P_1 P_2 = 0$). Many complicated sets, including the set of all primes or the exponential set

$$\{2^{3^{\cdot^{\cdot^{\cdot^{n}}}}} \mid n > 1\},$$

are Diophantine.

Actually, the work of J. Robinson, M. Davis and Y. Matijasevič (see the recent book [109]) has shown that combining the fact that every possible computation can be represented by a suitable polynomial with the fact that the language of Diophantine sets permits neither the use of universal quantification nor the use of the logical negation, proves that the famous *Hilbert's tenth problem* "Does there exist an algorithm to tell whether or not an arbitrary given Diophantine equation has a solution" has a *negative* answer.

1.2.2 Turing Machines, Chaitin Computers, and Chomsky Grammars

Before the work of A. Church, S. Kleene, K. Gödel and A. Turing there was a great deal of uncertainty about whether the informal notion of an algorithm—used since Euclid and Archimedes—could ever be made mathematically rigorous.

Turing's approach was to think of algorithms as procedures for manipulating symbols in a purely deterministic way. He imagined a device, a *Turing machine*, as it has come to be called later, having a finite number of states. At every given moment the machine is scanning a single square on a long, thin tape and, depending upon the state and the scanned symbol, it writes a symbol (chosen from a finite alphabet), moves left or right, and enters a new, possibly the same, state. Despite the extreme conceptual simplicity Turing machines are very powerful.[12]

There are many different models of Turing machines. At this point we use the following variant: a Turing machine TM will have three tapes, an input tape, an output tape, and a scratch tape. Such a machine determines a partial function, φ_{TM} from the set of strings over an alphabet Σ into itself: $\varphi_{TM}(x) = y$ if TM started in its initial state, with scratch and output tapes blank, and x on its input tape, writes y on its output tape and then halts. The class of partial functions computed by Turing machines coincides with the class of partial recursive functions; the languages computed by Turing machines are the recursively enumerable languages, [21, 23, 84, 114, 128].

For information-theoretical reasons Chaitin [35] (and, independently, Levin [103]) has modified the standard notion of Turing machine by requiring that as we are reading a string, we are able to tell when we have read the entire string. More precisely, we require that the input tape reading head cannot move to the left: at the start of the computation, the input tape is positioned at the leftmost binary digits of x, and at the end of the computation, for $\varphi_{TM}(x)$ to be defined, we now require that the input head be positioned on the last digit of x. Thus, while reading x, TM was able to detect at which point the last digit of x has occurred. Such a special Turing machine is called *self-delimiting Turing machine* or *Chaitin computer* (cf. [135, 24]). The class of partial recursive functions having a prefix-free domain[13] is exactly the class of partial functions computed Chaitin computers. In terms of languages, Chaitin computers have the same capability as Turing machines.

[12] In fact, Turing conjectured that a symbolic procedure is algorithmically computable just in case we can design a Turing machine to carry on the procedure. This claim, known as *Church-Turing's Thesis*, will be discussed later.

[13] No string in the domain is a proper prefix of another string in the domain.

Chomsky type-0 grammars offer another way to generate languages. Before going to some details let us a fix a piece of notation. For an alphabet V we denote by V^* the free monoid generated by V (λ is the empty string); the elements of the Cartesian product $V^* \times V^*$ will be written in the form $\alpha \to \beta,\ \alpha, \beta \in V^*$.

A *Chomsky (type-0) grammar* is a system $G = (V_N, V_T, w, P)$, where V_N and V_T are disjoint alphabets, $w \in V^*$ and P is a finite subset of $V^*V_NV^* \times V^*$, where $V = V_N \cup V_T$. The elements of V_N and V_T are referred to as nonterminal and terminal letters, respectively, and w is called the start, or axiom, string.

A *derivation* of length n in G with domain a_1 and codomain a_{n+1} is a triple of finite sequences $x = (pr^x, r^x, k^x)$ such that

1. $pr^x = (a_1, a_2, \ldots, a_{n+1})$ is a sequence of $n+1$ strings over V,
2. $r^x = (r_1, r_2, \ldots, r_n)$ is a sequence of n elements in P,
3. $k^x = (\langle u_1, v_1\rangle, \ldots, \langle u_n, v_n\rangle)$ is a sequence of n pairs of strings over V, and
4. for each $1 \leq i \leq n,\ a_i = u_i\alpha v_i,\ a_{i+1} = u_i\beta v_i,$ and $r_i = \alpha \to \beta$.

A derivation with domain w and codomain in V_T^* is called *terminal*. The language generated by G is defined to be the set of all codomains of terminal derivations. The languages generated by Chomsky grammars are exactly the recursively enumerable languages.[14]

1.2.3 Universality

Is it possible to design a "machine" capable to simulate any other "machine"? If we replace the word "machine" by Turing machine or Chaitin computer or Chomsky grammar, then the answer is *affirmative*. Proofs of this extremely important result[15] can be found in Turing's seminal paper [144], and in many monographs and textbooks (e.g. [84, 128]). The result is true also for Chomsky grammars as well, and in the next paragraph we shall illustrate the universality with a construction of the universal Chomsky grammar.

The Universal Chomsky Grammar

First we have to make precise the notion of "simulation".

A *universal Chomsky grammar* is a Chomsky grammar $U = (V_N, V_T, w, P)$ with the following property: for every recursively enumerable language L over V_T there exists a string $w(L)$ (depending upon L) such that the language generated by the grammar $(V_N, V_T, w(L), P)$ coincides with L.

Theorem 1.1 [27] *There exists a universal Chomsky grammar.*

Proof. Let

$$V_N = \{A, B, C, D, E, F, H, R, S, T, X, Y\} \cup V_T \times \{1, 2, 3, 4, 5, 6, 7, 8, 9\},$$

P consists of the following rules (we have used also the set $\Sigma = \{S, X, Y\} \cup V_T$):

1. $C \to BT$,
2. $Tx \to xT,\ x \in \Sigma \cup \{D, E\}$,
3. $TDx \to (x, 2)D(x, 1),\ x \in \Sigma$,
4. $y(x, 2) \to (x, 2)y,\ x \in \Sigma\ y \in \Sigma \cup \{D, E\}$,

[14] Traditionally, see [128], the axiom is a nonterminal letter; using a string in V^* instead of a single letter in V_T does not modify the generative capacity of grammars.

[15] Which represents the mathematical fact justifying the construction of present day computers.

5. $B(x,2) \to (x,3)B,\ x \in \Sigma,$
6. $y(x,3) \to (x,3)y,\ x,y \in \Sigma,$
7. $x(x,3) \to (x,4),\ x \in \Sigma,$

8. $(x,1)y \to (y,5)(x,1)(y,1),\ x,y \in \Sigma,$
9. $x(y,5) \to (y,5)x,\ y \in \Sigma, x \in \Sigma \cup \{D,E\},$
10. $B(y,5) \to (y,6)B,\ y \in \Sigma,$
11. $x(y,6) \to (y,6)x,\ x,y \in \Sigma,$
12. $(x,4)y(y,6) \to (x,4),\ x,y \in \Sigma,$

13. $(x,1)Ey \to (x,7)E(y,9),\ x,y \in \Sigma,$
14. $(x,1)(y,7) \to (x,7)y,\ x,y \in \Sigma,$
15. $D(x,7) \to DX,\ x \in \Sigma,$
16. $(x,9)y \to (y,8)(x,9)(y,9),\ x,y \in \Sigma,$
17. $x(y,8) \to (y,8)x,\ y \in \Sigma, x \in \Sigma \cup \{D,E,B\},$
18. $(x,9)(y,8) \to (y,8)(x,9),\ x,y \in \Sigma,$
19. $(x,4)(y,8) \to y(x,4),\ x,y \in \Sigma,$

20. $(x,1)ED \to (x,7)RED,\ x,y \in \Sigma,$

21. $(x,9)D \to RxD,\ x \in \Sigma,$
22. $(x,9)R \to Rx,\ x \in \Sigma,$
23. $xR \to Rx,\ x \in \Sigma \cup \{D,E\},$
24. $BR \to RC,$
25. $(x,4)R \to \lambda,\ x \in \Sigma,$

26. $Ax \to xA,\ x \in V_T,$
27. $AC \to H,$
28. $Hx \to H,\ x \in \Sigma \cup \{D,E\},$
29. $HF \to \lambda.$

First we notice that every recursively enumerable language can be generated by a Chomsky grammar having at most three nonterminals. Indeed, if L is generated by the grammar $G = (V_N, V_T, S, P)$ and V_N contains more than three elements, say $V_N = \{S, X_1, \ldots, X_m\}$ with $m > 2$, then we define the morphism $h : V^* \to (V_T \cup \{S, A, B\})^*$ (here A, B are symbols not contained in V) by $h(S) = S$, $h(a) = a$, for all $a \in V_T$, and $h(X_i) = AB^i$, $1 \leq i \leq m$. Let $h(P') = \{h(u) \to h(v) \mid u \to v \in P\}$. It is easy to check that L is generated by the grammar $G' = (\{S, A, B\}, V_T, S, P')$.

To complete the proof we consider the language L generated by the grammar $G = (\{S, X, X\}, V_T, S, Q)$ and we put

$$w(L) = ASCD\alpha_1 D\alpha_2 E\beta_2 D \ldots D\alpha_k R\beta_k DF,$$

where the set of productions is $Q = \{\alpha_i \to \beta_i \mid 1 \leq i \leq k\}$.

We analyse now a derivation from a string

$$A\gamma CD\alpha_1 D\alpha_2 E\beta_2 D \ldots D\alpha_k R\beta_k DF. \tag{1.1}$$

The first group of rules[16] constructs the nonterminal T which selects the rule $\alpha_i \to \beta_i$ occurring in the right hand side of a nonterminal D (see rule 3). By the second group of rules, the first symbol x in α_i goes into $(x, 1)$ and the nonterminal $(x, 2)$ is translated into the left hand side of B, where it becomes $(x, 3)$. If in γ there exists a symbol x, then by the rule 7 we construct the nonterminal $(x, 4)$.

The third group of rules transforms all symbols x from α_i in $(x, 1)$; then, every such x is removed from the right hand side of $(x, 4)$ in case such elements do appear in the same order.

The fourth group transforms every symbol y from $\beta_i \neq \lambda$ into $(y, 9)$ and translates the symbol $(y, 8)$ on the left hand side. When $(y, 8)$ reaches $(x, 4)$ the symbol y is introduced. In this way the string α_i, erased by the third group of rules, is replaced by β_i. In case $\beta_i = \lambda$ the rule 20 is used instead of the fourth group. Accordingly, we have reconstructed the derivation from γ to γ' using the rule $\alpha_i \to \beta_i$. This procedure can be iterated by means of rules in the sixth group. If γ' does not contain any nonterminal, then by rules in the last group the string can be reduced to γ'.

So, every string in L can be generated by the universal grammar with the axiom $w(L)$.

For the converse relation we notice that the nonterminal A can be eliminated only in the case when between A and C there exists a terminal string. Every derivation has to begin with the introduction of the nonterminal T. Erasing T determines the introduction of a nonterminal $(x, 1)$, which, in turn, can be eliminated by the nonterminal $(y, 9)$. These operations are possible if and only if the string α_i has been removed from γ. One can erase $(y, 9)$ after the translation of β_i in the place occupied by α_i. The symbol R constructed in this way can be eliminated when the given string is reduced to the form (1.1). In this way we have constructed a derivation using the rule $\alpha_i \to \beta_i$. All derivations which are not of this from will be eventually be blocked. □

1.2.4 Silencing a Universal Computer

A universal machine, be it Turing, Chomsky or Chaitin, despite all the clever things it can do, is not *omniscient.*[17]

Let P be program whose intended behaviour is to input a string, and then output another string. The "meaning" of P may be regarded as a function from strings over an alphabet V into strings over the same alphabet, i.e. a function $f : V^* \to V^*$. Such a program P on input x

- may eventually stop, in which case it prints a string, or
- it may run forever, in which case we say, following [134], that the program has been "silenced" by x.

We shall prove that *every universal program can be silenced by some input.* Here is the argument.

List all valid programs, $P_1, P_2, \ldots, P_n, \ldots$.[18]

Now suppose, by absurdity, that we have a universal program that cannot be silenced, so, by universality, we have a universal programming language such that none of its programs can be silenced. Let g_n by the function computed by P_n and construct, by *diagonalization*, the function $f(x) = xg_{string(x)}(x)$, where $string(x)$ is the position of the string x in the quasi-lexicographical enumeration of all strings in V^*.

- The function f is computable by the following procedure: given the string x, first compute $string(x)$, then generate all programs until we reach the program $P_{string(x)}$, run $P_{string(x)}$ on the input x and then concatenate x with the result of the above computation.

[16] There are seven groups of rules separated empty lines.

[17] As Leibniz might have hoped.

[18] There are many ways to do it, for instance, by listing all valid programs in quasi-lexicographical order. Such an enumeration is referred to as a *gödelization* of programs, as Gödel used first this method in his famous proof of the Incompleteness Theorem [59, 60].

- In view of the universality it follows that f has to be computed by some program P_n. Now, take the string x such that $string(x) = n$ and the input x: $f(x) = g_n(x)$, which false.

This concludes our proof.

At a first glance the contradiction in the above argument can be avoided, and here are two possibilities. First add to the initial programming language the program computing f. This construction does not work as we can use again the diagonalization method to derive a contradiction for the new language. Another possibility would be to say that f is "pathological" and we don't really want it computable. The last option is perfectly legitimate;[19] however, if we want to preserve universality we have to admit non-terminating programs.

1.2.5 *Digression: A Simple Grammatical Model of Brain Behaviour

A universal grammar can be used in describing a model of human brain behaviour.[20] The model has four grammatical components:

- the "grammar generator",
- the "parser",
- the "semantic analyzer", and
- the universal grammar.

The "grammar generator" can be realized as a regular grammar, but the "parser" and the universal grammar are type-0 Chomsky grammars.[21] The model works as follows (see [26]): an external stimulus, a question, for example, comes to the brain under the form of a string x in some language. Then an "exciting relay" activates the "grammar generator" which produces strings of the form

$$x_1 \to y_1 / x_2 \to y_2 / \cdots / x_k \to y_k$$

corresponding to the rules of that specific grammar. In fact, we assume that the grammar generator is producing exactly a string w of the form (1.1) required by the universal grammar. The string w and the input string come to a "syntactic analyzer" which decides whether x belongs or not to the language identified by w. If the answer is affirmative, that is the right "competence" corresponding to x has been found, then x and w come in the universal grammar which is the core of the answering device. The universal grammar behaves now as the specific grammar and is used to "answer" the stimulus. The answering mechanism, using some information provided by a "semantic analyzer" and some memory files, generates the answer and sends it to the environment.

The full information capability of the brain can be accomplished by using only these "few" components of the model, without actually retaining all grammars corresponding to the competences of the human brain.

1.2.6 The Halting Problem

We have seen that each universal machine can be silenced by some input. Is it natural to ask the following question: Can a machine test whether an arbitrary input will silence the universal machine? The answer is again *negative*.

For the proof we will assume, without restricting generality, that all valid programs incorporate inputs—which are coded as natural numbers. So, a program may be silenced or may just eventually stop, in which case it prints a natural number. Assume further that there exists a **halting program** deciding whether the universal program is silenced by an arbitrary input, that is, by universality, whether an arbitrary program is going to be silenced or not. Construct the following program:

[19] The language of primitive recursive functions is such a language.

[20] We adopt here the *Computabilism Thesis for Brains* according to which the "brain functions basically like a digital computer". This thesis has been formulated, without elaboration, by Gödel in 1972, cf. [153]. We also distinguish, again with Gödel, the mind from the brain: *The mind is the user of the brain functioning as a computer.* For more details we refer to Section 1.2.8.

[21] An attempt to implement this model at the level of context-sensitive grammars is discussed in [28].

1. read a natural N;
2. generate all programs up to N bits in size;
3. use the **halting program** to check for each generated program whether it halts;
4. simulate the running of the above generated programs, and
5. output double the biggest value output by these programs.

The above program halts for every natural N. How long is it? It is about $\log N$ bits. Reason: to know N we need $\log N$ bits (in binary); the rest of the program is a constant, so our program is $\log N + \mathrm{O}(1)$[22] bits.

Now observe that *there is a big difference between the size—in bits—of our program and the size of the output produced by this program.* Indeed, for large enough N, our program will belong to the set of programs having less than N bits (because $\log N + \mathrm{O}(1) < N$). Accordingly, the program will be generated by itself—at some stage of the computation. In this case we have got a contradiction since our program will output a natural number two times bigger than the output produced by itself!

The following two questions:

Does the Diophantine equation $P = 0$ have a solution?

Does the Diophantine equation $P = 0$ have an infinity of solutions?

are algorithmically unsolvable, but, they have a *different degree of unsolvability*!

If one considers a Diophantine equation with a parameter n, and asks whether or not there is a solution for $n = 0, 1, 2, \ldots, N-1$, then the N answers to these N questions really constitute only $\log_2 N$ bits of information, as we can determine which equation has a solution if we know *how many* of them are solvable. These answers are not independent. On the other hand, if we ask the second question, then the answers can be independent, if the equation is constructed properly.[23] The first question never leads to a pure chaotic, random behaviour, while the second question may sometimes lead to randomness.

1.2.7 Church–Turing's Thesis

Church-Turing's Thesis, a prevailing paradigm in computation theory, states that no realizable computing device can be "globally" more powerful, that is, aside from relative speedups, than a universal Turing machine. It is a *thesis*, and not a theorem, as it relates an informal notion—a realizable computing device—to a mathematical notion. Re-phrasing, Church-Turing's Thesis states that the universal Turing machine is an *adequate* model for the discrete computation. Here are some reasons why Church-Turing's Thesis is universally accepted:

- *Philosophical argument*: Due to Turing's analysis it seems very difficult to imagine some other method which falls outside the scope of his description.
- *Mathematical evidence*: Every mathematical notion of computability which has been proposed was proven equivalent to Turing computability.
- *Sociological evidence*: No example of computing device which cannot be simulated by a Turing machine has been given, i.e. the thesis has not been disproved despite having proposed over 60 years ago.

Church-Turing's Thesis includes a syntactic as well a physical claim. In particular, it specifies which types of computations are physically realisable. According to Deutsch [52], p. 101:

[22] $f(n) = \mathrm{O}(g(n))$ means that there exists a constant c such that $|f(n)| \leq c|g(n)|$, for all n.

[23] Chaitin [37] has effectively constructed such an equation; the result is a 900 000–character 17 000–variable universal exponential Diophantine equation.

> The reason why we find it possible to construct, say, electronic calculators, and indeed why we can perform mental arithmetic, cannot be found in mathematics or logic. *The reason is that the laws of physics "happen" to permit the existence of physical models for the operations of arithmetic* such as addition, subtraction and multiplication. If they did not, these familiar operations would be non-computable functions. We might still know *of* them and invoke them in mathematical proofs (which would be presumably be called "non-constructive") but we could not perform them.

As physical statements may change in time, so may our concept of computation. Indeed, Church-Turing's Thesis has been recently re-questioned; for instance, [132] has proposed an alternative model of computation, which builds on a particular chaotic dynamical system [55] and surpasses the computational power of the universal Turing machine. See [146, 72, 73, 145, 14, 123, 81, 56, 140, 141, 113] for related ideas.

1.2.8 *Digression: Mind, Brain, and Computers

Thinking is an essential, if not the most essential, component of human life—it is a mark of "intelligence".[24] In the intervening years Church-Turing's Thesis has been used to approach formally the notion of "intelligent being". In simple terms, Church-Turing's Thesis was stated as follows: *What is human computable is computable by a universal Turing machine.* Thus, it equates information-processing capabilities of a human being with the "intellectual capacities" of a universal Turing machine.[25] This discussion leads directly to the traditional problem of mind and matter which exceeds the aim of this paper (see the discussion in [51, 58, 70, 126, 127, 120, 121, 130]); in what follows we shall superficially review this topic in connection with the related question: can computers think?

The responses to the mind-body problem are very diverse; however, there are two main trends, *monism* which claims that the distinction between mind and matter is only apparent, simply, the mind is identical with the brain and its function, and *dualism* which maintains the we have a real distinction.

The dualism can be traced to Descartes. There are many types of dualism, among them being:

- "categorical dualism" (the mind and the body are different logical entities);
- "substance dualism" illustrated by Popper or Gödel, and claiming that mind exists in a mental space outside space or time, and the brain is just a complex organ which "translates" thoughts into the corporeal movements of the body. Gödel rejected monism by saying (in Wang's words, [153], p. 164) that monism *is a prejudice of our time which will be disproved scientifically—perhaps by the fact that there aren't enough nerve cells to perform the observable operations of the mind.* This is a challenging *scientific conjecture*—indeed, the capacity of nerve cells is a scientific research topic for neuroscience and the observable operations of mind are also things subject to scientific analysis. According to same author ([153], p. 169) Gödel asserted that *the brain functions basically like a digital computer.* The user of the brain functioning as a computer is just the *mind.*
- "property dualism" maintaining that the mind and our experiences are "emergent" properties of the material brain;
- "epistemic dualism", illustrated by Kant, saying that from a "theoretical reason" the states of the mind are reducible to the states of the brain, but from a "practical reason" such a reduction is not possible.

Von Neumann remarked that in 1940s two outstanding problems were confronting science: weather prediction and the brain's operation. Today we have a fairly better understanding of the complexity of weather,[26] but the brain still remains a mystery. Perhaps the brain, and accordingly, the mind, are simply unsimulatable and the reason is what von Neumann remarked: *the simplest model of a neuron may be a neuron itself.* This property suggests the notion of *randomness*, which will be discussed in a separate section.

[24] Descartes placed the essence of being in thinking.

[25] This may create the false impression of "EOE policy": *all brains are equal.*

[26] Even if the weather equations behave chaotically, that is a small difference in the data can cause wildly different weather patters.

1.3 Computational Complexity Measures and Complexity Classes

1.3.1 Time and Space Complexities and Their Properties

In the early 1960s the border between algorithmically solvable problems and algorithmically unsolvable ones was already well–defined and understood and scientists knew methods powerful enough to decide whether a given computing problem is decidable (algorithmically solvable) or undecidable. Because of the growth of the computer use in many areas of every day life the interest of researchers has moved to the questions like "How to measure the effectivity of computer programs (algorithms)?", "How to compare the effectivity of two algorithms?", and "How to measure the computational difficulty of computing problems?". Dealing with these questions two fundamental complexity measures, time and space, have been introduced by Hartmanis at. al. [78, 77]. Both these complexity measures are considered as functions of the input. Informally, the time complexity of an algorithm working on an input is the number of "elementary" operations executed by the algorithm processing the given input. In another words, it is the amount of work done to come from the input to the corresponding output. The space complexity of an algorithm is the number of "elementary" cells of the memory used in the computing process. Obviously, what does "elementary" mean depends on the formal model of algorithms one chooses (machine models, axiomatic models, programming languages, etc.). Since the theory always tries to establish results (assertions) which are independent on the formalism used and have a general validity, this dependence of the measurement on the model does not seem welcome. Fortunately, all reasonable computing models used are equivalent, in the sense that the differences in the complexity measurement are negligible for the main concepts and statements of the complexity theory.

Here, we use Turing machine (TM) as the standard computing model of computation theory to define the complexity measures. From the several versions of Turing machine we consider the off–line multitape Turing machine (MTM) consisting of a finite state control, one two–way read–only input tape with one read–only head and a finite number of infinite working tapes, each with one read/write head. This is the standard model used for the definitions of time and space complexities because it clearly separates the input data (input tape) from the computer memory (working tapes) [78]. A **computing step** (shortly, step) of a MTM is considered to be the elementary operation of this computing model. In one step a MTM M reads one symbol from each of its tapes (exactly those symbols are read which are on the positions where the heads are adjusted) and depending on them and the current state of the machine M, M possibly changes its state, rewrites the symbols read from the working tapes and moves the heads at most one position to the left or to the right. A configuration of a MTM M is the complete description of the global state of the machine M including the state of the finite control, the current contents of all tapes, and the positions of all heads on the tapes. A computation of M is a sequence of configurations $C_1, C_2, \ldots, C_m$ such that $C_i \vdash C_{i+1}$ (C_{i+1} is reached in one step from C_i). A formal description of $MTMs$ can be found in all textbooks on this topic and we omit it here. Now, we are ready to define the complexity measures.

Definition 1.2 *Let M be a MTM recognizing a language $L(M)$ and let $x \in Z^*$, where Z is the input alphabet of M. If $C = C_1, C_2, \ldots, C_k$ is the finite computation of M on x, then the* **time complexity of the computation of M on x** *is*

$$T_M(x) = k - 1 \, .$$

If the computation of M on x is infinite, then

$$T_M(x) = \infty \, .$$

The **time complexity of** *M is the partial function from $\mathbb{N}$ to $\mathbb{N}$,*

$$T_M(n) = \max\{T_M(x) \mid x \in \Sigma^n \cap L(M)\}.$$

The space complexity of one configuration C of M, $S_M(C)$, is the length of the longest word over the working alphabet stored on the working tapes in C. The **space complexity of a computation** *$D = C_1, \ldots, C_m$ is*

$$S_M(D) = \max\{S_M(C_i) \mid i = 1, \ldots, m\}.$$

The **space complexity of** M **on a word** $x \in L(M)$ *is* $S_M(x) = S_M(D_x)$, *where* D_x *is the computation of* M *on* x.
The **space complexity of** M *is the partial function from* $\mathbb{N}$ *to* $\mathbb{N}$,

$$S_M(n) = \max\{S_M(x) \mid x \in \Sigma^n \cap L(M)\}.$$

One can observe that the function $T_M(x)(S_M(x))$, as a function from $L(M)$ to $\mathbb{N}$ is the most precise (complete) description of the complexity behaviour of the machine M. But this description is not useful for the comparison of complexities of different algorithms ($MTMs$). It is so complex that one can have trouble to find a reasonable description of it. Thus, we prefer to consider the complexity measures as functions of the input sizes rather than of specific inputs. The time and space complexities are so called **worst case** complexities because $T_M(n)(S_M(n))$ is maximum of the complexities over all words of length n from $L(M)$ (i.e. M can recognize every input from $L(M) \cap \Sigma^n$ in time $T_M(n)$ and there exists $x \in L(M) \cap \Sigma^n$ such that $T_M(n) = T_M(x)$). In abstract complexity theory we usually prefer this worst–case approach to define the complexity measures, but in the analysis of concrete algorithms we are often interested to learn the average behaviour of the complexity on inputs of a fixed length. In such case one has to make a probabilistic analysis of the behaviour of $T_M(x)$ according to the probability distribution of inputs of a fixed length.

Another interesting point in the definition above is that to define $T_M(n)$ and $S_M(n)$ we have considered $T_M(x)$ and $S_M(x)$ only for words from $L(M)$. This is because we allow infinite (or very complex) computations of M on words in $\Sigma^* - L(M)$. In what follows we show that if $T_M(S_M)$ is a "nice" function, then it does not matter for the complexity of the recognition of the language $L = L(M)$ whether one defines T_M as above or as $T_M(n) = \max\{T_M(x) \mid x \in \Sigma^n\}$. The idea is to construct a machine M' simulating M in such a way that if M' simulates more than $T_M(n)$ steps of M on an input y, then M' halts and rejects the input. Obviously, M' is able to do it in this way if M' is able to compute the number $T_M(|y|)$ in time $O(T_M(|y|))$. Before starting to describe this we show that it is sufficient to study the asymptotical behaviour of $T_M(n)$ and $S_M(n)$.

Definition 1.3 [78, 77] *Let* t *and* s *be two functions from* $\mathbb{N}$ *to* $\mathbb{N}$. *Then*

$$TIME(t(n)) = \{L \mid L = L(M) \text{ for a } MTM\ M \text{ with } T_M(n) \leq t(n)\},$$

and

$$SPACE(s(n)) = \{L \mid L = L(M) \text{ for a } MTM\ M \text{ with } S_M(n) \leq s(n)\}.$$

Theorem 1.4 [77] *Let* c *be a positive real number and let* $s : \mathbb{N} \to \mathbb{N}$ *be a function. Then*

$$SPACE(s(n)) = SPACE(c \cdot s(n)).$$

Idea of proof. Without loss of generality we assume $c < 1$. Then, $SPACE(c \cdot s(n)) \subseteq SPACE(s(n))$ is obvious. To show that $SPACE(s(n)) \subseteq SPACE(c \cdot s(n))$ we use a compression of the contents of the tapes. Let M be a MTM recognizing a language $L(M) \in SPACE(s(n))$ with $S_M(n) \leq s(n)$. Let $k = \lceil 1/c \rceil$ and let Γ be the working alphabet of M. Then one constructs a MTM M' with the working alphabet Γ^k. This enables to store in one cell of the tapes of M' the contents of k adjacent cells of tapes of M. The detailed construction of the rules of M' is left to the reader.

□

The same idea of the compression can be used to reach the following speed–up theorem.

Theorem 1.5 [78] *For every constant* $c > 0$ *and every function* $t : \mathbb{N} \to \mathbb{N}$ *such that* $\liminf_{n\to\infty} t(n)/n = \infty$

$$TIME(t(n)) = TIME(c \cdot t(n)).$$

Obviously, the compression of tapes in the theorems above is not completely fair because we save space and time according to the complexity definition by storing larger data in one memory cell and executing more complicated operations in a step of the simulating machine. If one fixes the working

alphabet to $\{0,1\}$ this effect will be impossible. So, the lack of difference between $c \cdot f(n)$ and $f(n)$ for complexity considerations on Turing machines is rather a property of the computing model than a complexity property of algorithms. On the other hand, in studying the complexity of concrete problems one primarily estimates the asymptotical behaviour (polynomial growth, for instance) of complexity functions. Thus, we are satisfied with the study of asymptotical behaviours of complexity functions and, accordingly, we can accept Turing machines as a computing model for the complexity measurement.

Definition 1.6 *A function* $s : \mathbb{N} \to \mathbb{N}$ *is called* **space–constructible** *if there exists a MTM* M *having the following two properties:*

(i) $S_M(n) \leq s(n)$ *, for all* $n \in \mathbb{N}$*, and*

(ii) for every $n \in \mathbb{N}$*,* M *starting on the input* 0^n *computes* $0^{s(n)}$ *on the first working tape and halts in a final state.*

A function $t : \mathbb{N} \to \mathbb{N}$ *is called* **time–constructible** *if there exists a MTM* A *having the following two properties:*

(iii) $T_M(n) = O(t(n))$ *, and*

(iv) for every $n \in \mathbb{N}$*,* M *starting on the input* 0^n *computes* $0^{t(n)}$ *on the first working tape and halts in a final state.*

Now, we can show that for constructible functions it does not matter whether $T_M(n)$ is defined as $\max\{T_M(x) \mid x \in L(M) \cap \Sigma^n\}$ or as $\max\{T_M(x) \mid x \in \Sigma^n\}$.

Lemma 1.7 *Let* $t : \mathbb{N} \to \mathbb{N}$ $(s : \mathbb{N} \to \mathbb{N})$ *be a time–constructible (space–constructible) function. Then for every* $L \in TIME(t(n))$ $(L \in SPACE(s(n)))$*,* $L \subseteq \Sigma^*$*, there exists a MTM* M *such that*

(i) $L(M) = L$*, and*

(ii) for every $x \in \Sigma^*, T_M(x) = O(t(|x|))$ $(S_M(x) \leq S(|x|))$.

Proof. We give the proof for the time complexity. Since $L \in TIME(t(n))$ we may assume there is a MTM A with $L(A) = L$ and $T_A(n) \leq t(n)$. If A has k working tapes, then we construct M with $k+1$ working tapes acting on the input x as follows:

1. M computes $0^{t(|x|)}$ on the first working tape in time $O(t(n))$.
2. M simulates $t(|x|)$ steps of the computation of A on x.
3. If A accepts x, then M accepts x too. If A halts and rejects x, then M rejects, too. If A does not halt after $t(|x|)$ steps, then M halts and rejects x.

Note that step 1 can be done because t is time–constructible. Since M uses k free working tapes to simulate A with k working tapes, M can simulate one step of A in one step. Rewriting 0 to 1 in each simulation step M one can check that $t(|x|)$ simulation steps were executed. Since each accepting computation of A on a word $x \in L(A)$ fulfills $T_A(x) \leq T_A(|x|)$ one sees that every computation longer than $T_A(|x|)$ cannot lead to acceptance of x (i.e. $x \notin L(A)$). Thus $L = L(A) = L(M)$ and $T_M(|x|) = O(t(|x|))$ for every $x \in \Sigma^*$.

□

As we have seen the complexity classes bounded by two functions f and g are the same if $f(n) = \Theta(g(n))$.[27] A natural and important question has arisen. Which increase of the growth of g in the comparison to the growth of f is sufficient and necessary to reach $TIME(f(n)) \subset TIME(g(n))$ or
$SPACE(f(n)) \subset SPACE(g(n))$? The answer to this question is provided by the following theorems

[27] If there are two constants c, N such that for all $n \geq N$, $f(n) \geq cg(n)$, then we say that $f(n) = \Omega(g(n))$. We say that $f(n) = \Theta(g(n))$ in case $f(n) = \mathrm{O}(g(n))$ and $f(n) = \Omega(g(n))$.

which have been established by generalizing the well-known diagonalization method from the computability theory.

Before starting to formulate them, we observe that every $MTM\ A$ can be simulated by a $MTM\ B$ with one working tape only in the same space [77]. This is because the i-th cell of the working tape of B can save the tuple containing all symbols on the i-th positions of all working tapes of A.

Theorem 1.8 [77] *Let* s_1 *and* s_2 *be two functions having the following properties:*

(i) $s_2(n) \geq s_1(n) \geq \log_2 n$ *for every* $n \in \mathbb{N}$,

(ii) s_2 *is space–constructible,*

(iii) $\liminf\limits_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$.

Then, $SPACE(s_1(n)) \subset SPACE(s_2(n))$.

Proof. The proof idea is based on the diagonalization technique. The simple version of this technique orders all Turing machines in an enumerable sequence $T_1, T_2, \ldots, T_i, \ldots$ and, for any $i \in \mathbb{N}$, it chooses a word $x_i \in \{0,1\}^*$ $(x_i \neq x_j \ for\ i \neq j)$. Then, the diagonal language L_d is defined as $\{x_i \mid x_i \notin L(T_i),\ i \in \mathbb{N}\}$. Since, for every $i \in \mathbb{N}$, $L(T_i)$ and L_d differs in the behaviour on x_i we call x_i the **candidate** for the difference between $L(T_i)$ and L_d in the process of the choice of x_i.

There are two reasons why this simple approach fails to work directly for our theorem. We cannot enumerate the sequence of MTMs working in space s_1 because it is not decidable whether a given MTM is $s_1(n)$–space bounded. This means we have to take the sequence of all MTMs, and for every MTM M of this sequence (even in case M is not $s_1(n)$–space bounded), we have to try to find a candidate for the difference between $L(M)$ and the diagonal language. Obviously, if a machine M is not $s_1(n)$–space bounded the candidate may fail because we do not need to have the difference between $L(M)$ and L_d. We must be only sure that our candidates will work for $s_1(n)$–space bounded machines.

A second, more serious problem is the following one. L_d has to be in $SPACE(s_2(n))$, i.e. there must exist a $MTM\ M_2$ such that $L(M_2) = L_d$ and $S_{M_2}(n) \leq s_2(n)$. Since M_2 has a working alphabet of a fixed size and we want to simulate the work of other $s_1(n)$–space bounded machines on the chosen candidates we need more than $s_1(n)$ space to do it if the working alphabet Γ of the simulated machine is larger than the working alphabet of M_2. More precisely, M_2 needs $\lceil \log_2 |\Gamma| \rceil \cdot s_1(n)$ space for this simulation which may be greater than $s_2(n)$ for finitely many small n's. Thus, for words of small lengths M_2 is not able to simulate the machine with large Γ. It means we cannot fix the candidates x_i's for T_i's without knowing that they are large enough to have $s_2(|x_i|) \geq (\log_2 |\Gamma_i|) \cdot s_1(|x_i|)$, where Γ_i is the working alphabet of T_i.

To overcome this difficulty we choose for every $MTM\ T_i$ an infinite set of candidates for the difference between $L(T_i)$ and $L(M_2)$. Then, we can be sure that one candidate y will be large enough for $s_2(|y|) \geq (\log_2 |\Gamma_i|) \cdot s_1(|y|)$ and so M_2 can simulate T_i on y and accepts (rejects) if T_i has rejected (accepted).

Now, we are prepared to give the formal proof. As we have already observed it is sufficient to consider MTMs with one working tape only. Let $T_1, T_2, \ldots$ be sequence of all MTMs with one working tape and the input alphabet $\{0,1\}$, and let $\tilde{T}_1, \tilde{T}_2, \ldots$ be their binary codes. The infinite set $X_i = \{x \in \{0,1\}^* \mid x = 0^n 1 \tilde{T}_i\}$ will be considered as the set of candidates for the difference between $L(T_i)$ and $L(M_2)$. Obviously $X_i \bigcap X_j = \emptyset$ for $i \neq j$.

We define the diagonal language as the language $L(M_2)$ accepted by M_2 acting on every input w as follows:

1. M_2 computes $0^{S_2(|w|)}$ on the first working tape.
2. M_2 considers w as $0^r 1 \tilde{T}$ for some $r \in \mathbb{N}$ and $\tilde{T} \in \{0,1\}^*$. If $|\tilde{T}| > S_2(|w|)$, then M_2 halts and rejects w, else M_2 continues with the step 3.
3. M_2 decides whether $\tilde{T}$ is a code of a $MTM\ T$ with one working tape and the input alphabet $\{0,1\}$. If not, M_2 halts and rejects w. If the answer is yes, then M_2 continues with the step 4.
4. M_2 writes the number $2^{S_2(|w|)}$ in binary on the second working tape. On the third working tape M_2 simulates step by step the computation of T on w. The working alphabet Γ of T is coded by the binary working alphabet of M_2.

(a) If M_2 needs more than $s_2(|w|)$ space on the third tape (i.e. if $s_2(|w|) < \lceil \log_2(|\Gamma|) \rceil \cdot s_1(|w|)$ or T is not s_1–space bounded), then M_2 halts and rejects w.

(b) If M_2 has successfully simulated $2^{s_2(|w|)}$ steps of T and T does not halt, then M halts and accepts w.

(c) If T halts on w in fewer than $2^{s_2(|w|)}$ steps and M_2 succeeds to simulate all these steps, then M_2 accepts w iff T rejects w.

We observe that M_2 always halts and that $S_{M_2}(n) \leq s_2(n)$ for every $n \in \mathbb{N}$.

Now, we have to show that $L(M_2) \notin SPACE(s_1(n))$. We assume that $L(M_2) \in SPACE(s_1(n))$. Then there exists a MTM M with one working tape such that $L(M) = L(M_2)$ and $S_M(n) \leq s_1(n)$. Let Q be the set of states of M, and let Γ be the working alphabet of M. Obviously, for a given input w, there are at most

$$r(|w|) = |Q| \cdot s_1(|w|) \cdot (|\,\Gamma\,| + 1)^{s_1(|w|)} \cdot |w|$$

different configurations of M. This means that any computation of M on w consisting of more than $r(|w|)$ steps is infinite because M is deterministic and some configuration has occurred twice in the computation. Since $\liminf_{n\to\infty} s_1(n)/s_2(n) = 0$, there exists a positive integer n_0 such that:

(i) $|\,\tilde{M}\,| \leq s_2(n_0)$,

(ii) $\lceil \log_2(|\,\Gamma\,|) \rceil \cdot s_1(n_0) \leq s_2(n_0)$, and

(iii) $|Q| \cdot s_1(n_0) \cdot (|\,\Gamma\,| + 1)^{s_1(n_0)} \cdot n_0 < 2^{s_2(n_0)}$.

Set $x = 0^j 1 \tilde{M}$ for a $j \in \mathbb{N}$ such that $|x| = n_0$. If $x \in L(M)$, then there exists an accepting computation of M on x of the length at most $r(|x|)$. Because (i), (ii), (iii) are fulfilled, M_2 succeeds to simulate the whole computation of M on x. Thus, M_2 rejects x according to 4(c).
If $x \notin L(M)$, then two different reasons may determine it:

1^0 M halts in at most $r(|x|)$ steps and rejects x. Then, following the step 4(c), M accepts x.

2^0 The computation of M on x is infinite. Then according to (i), (ii) and (iii), M_2 succeeds to simulate the first $2^{s_2(|x|)} = 2^{s_2(n_0)}$ steps of M on x. Because of the step 4(b) M_2 halts and accepts x.

Thus, we have showed "$x \in L(M)$ iff $x \notin L(M_2)$" which is a contradiction.

□

The assertion above shows that the power of Turing machines strongly grows with the asymptotical growth of the space complexity used. For space bounds growing slower than $\log_2 n$ the situation is a little bit different. A survey of such small space–bounded classes can be found in [67, 139]. We only mention that $SPACE(0(1))$ [constant space] is exactly the class of regular languages and that if $SPACE(f(n))$ contains at least one nonregular language, then $\limsup_{n\to\infty} f(n)/\log\log_2 n > 0$. Thus the space classes $SPACE(s(n))$ with $s(n) = o(\log\log_2 n)$ [28] contain only regular languages.

A similar hierarchy can be achieved for time complexity, but it is not so strong as the space hierarchy. The reason is that we do not have only the problem to simulate different working alphabets of arbitrarily large cardinalities by one working alphabet of the "diagonalizing" MTM, but we have to fix the number of working tapes for the diagonalizing MTM while the simulated MTMs may have any number of working tapes. Since, for any $k \in \mathbb{N}$, we do not know a linear time simulation of an arbitrary number of tapes by a k–working tapes we formulate the hierarchy result as follows.

Theorem 1.9 [78] *Let there exist a positive integer k, a function $f : \mathbb{N} \to \mathbb{N}$, and a simulation of an arbitrary MTM A by a MTM with k working tapes in $O(T_A(n) \cdot f(n))$ time. Let t_1, t_2 be two functions from $\mathbb{N}$ to $\mathbb{N}$ such that*

[28] $f(n) = o(g(n))$ in case $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

(i) $t_2(n) \cdot f(n) \geq t_1(n)$ *for every* $n \in \mathbb{N}$,

(ii) t_2 *is time–constructible, and*

(iii) $\liminf_{n\to\infty} \frac{t_1(n)\cdot f(n)}{t_2(n)} = 0$.

Then $TIME(t_1(n)) \subset TIME(t_2(n))$.

Due to the fact that every $t(n)$ time–bounded MTM can be simulated by a MTM with two working tapes in $O(t(n) \cdot \log_2(t(n)))$ [80], we deduce the relation

$$TIME(t_1(n)) \subset TIME(t_1(n) \cdot \log_2(t_1(n)) \cdot q(n)),$$

for any monotone, unbounded $q : \mathbb{N} \to \mathbb{N}$.

The hierarchy result above shows that there are languages of arbitrarily large complexity, i.e. we have an infinite number of levels for the classification of the computational difficulty of language recognition (computing problems). If $L \in TIME(t_1(n)) - TIME(t_2(n))$, then we say that $t_1(n)$ is the **upper bound** on the time complexity of L and $t_2(n)$ is the **lower bound** on the time complexity of L. So, the upper bound t_1 for L means that there is an algorithm (MTM) A recognizing L with $T_A(n) \leq t_1(n)$ and the lower bound $t_2(n)$ means that there is no algorithm (MTM) B recognizing L with $T_B(n) \leq t_2(n)$. At this point one may wish to define the time (space) complexity of a language (computing problem) as the complexity of the best (asymptotically optimal) algorithm (MTM) recognizing L. But this is impossible because there are languages with no best MTM. This fact is more precisely formulated for time complexity in the following version of Blum's Speed–up Theorem (this result works for an arbitrary Blum space, cf. [23]).

Theorem 1.10 [17]

There exists a recursive language L *such that for any* MTM M_1 *accepting* L, *there exists a* MTM M_2 *such that*

(i) $L(M_2) = L(M_1)$, *and*

(ii) $T_{M_2}(n) \leq \log_2(T_{M_1}(n))$ *for almost all* $n \in \mathbb{N}$.

Idea of proof. The assertion of Blum's Speed–up Theorem may seem curious and surprising in the first moment. But the following idea shows an explanation of this phenomenon. First, one can observe that for every language L' there are infinitely many MTMs accepting it. Then, one has to construct a language L which cannot be recognized efficiently by any MTM (program) of small size (small index), but as MTM (program) sizes increase faster and faster MTMs accepting L more and more efficiently exist. The formal construction of L can be done by diagonalization.

□

We see that the theorem above can be infinitely many times applied to L and so there is no optimal MTM accepting L. For every concrete MTM M there exists a larger MTM M' working more efficiently than M.[29]

This is the reason why we cannot generally define the complexity of a computing problem (language) as the complexity of the optimal algorithm for it. But, we can always speak about lower and upper bounds on the problem complexity which is sufficient for the classification of computing problems according to their computational difficulty.

We close this section studying the relation between time complexity and space complexity. We observe that $TIME(t(n)) \subseteq SPACE(t(n))$ for any function $t(n) \geq n$ because no MTM can use more cells of the working tapes than the number of executed steps of M is. On the other hand we have $SPACE(s(n)) \subseteq \bigcup_{c\in\mathbb{N}} TIME(c^{s(n)})$ for any $s(n) \geq \log_2 n$. This is because for every $s(n)$–space bounded MTM M there is a constant c such that the number of different configurations of M with a fixed content of the input tape is bounded by $c^{s(n)}$. These above two relations suggest that space may be much more powerful than time. The best result currently known in this direction is contained in the following assertion.

[29] The speed-up phenomenon is non-constructive; for instance, there is no recursive function of the initial index that gives a bound for the exceptional values in Blum speed-up, but that there is a recursive bounding function of the speed-up index, [22]. The class of speedable functions is large, [31].

Theorem 1.11 [82] *For any function* $t : \mathbb{N} \to \mathbb{N}$ *such that* $t(n)/\log(t(n)) = \Omega(n)$*:*

$$TIME(t(n)) \subseteq SPACE(t(n)/\log(t(n))).$$

1.3.2 Classification of Problems According to Computational Difficulty and Nondeterminism

The hierarchy theorems show that there are problems of arbitrarily large computational difficulty. The main practical interest is in the classification of concrete computing problems according to their computational difficulty. To see the importance of this let us consider the following classical example [79]. Let us have four algorithms (MTMs) A_1, A_2, A_3, and A_4. Let $T_{A_1}(n) = 5n, T_{A_2}(n) = n \cdot \log n, T_{A_3}(n) = n^2$, and $T_{A_4}(n) = 2^n$. Then for the realistic input size $n = 1000$ we have $T_{A_1}(1000) = 5000, T_{A_2}(1000) = 9966, T_{A_3}(1000) = 1000000$, and $T_{A_4}(1000)$ is a 302–digit number. For comparison, the number of protons in the known universe has 126 digits, and the number of microseconds since the "Big Bang" has 24 digits. Thus, A_4 is not useful for any practical purposes because already $T_{A_4}(100)$ is a 31–digit number. If A_4 is an optimal algorithm for some problem L, then we cannot algorithmically solve L in general. We can see this also from the opposite side. Every computer has a finite memory and we always have a bound on the time we can wait for a result. These two constants together with the complexity functions of the given algorithm A bound the size of inputs which can be processed by the algorithm A on a given computer. We observe that, for algorithms with exponential complexity, the bounds are very small because, for $f(n) = 2^n, f(n + 10) \geq 10^3 \cdot f(n)$. Considerations similar to those made above have lead to the classification of computing problems into **tractable** (feasible) problems admitting a polynomial–time solution [46] and **intractable** problems for which no polynomial algorithm exists. The basic class for the classification of problems (languages) is the class

$$P = \bigcup_{k \in \mathbb{N}} TIME(n^k).$$

An important observation is that the definition of P is invariant with respect to all "reasonable" computing models used, i.e. if one finds a polynomial algorithm for a problem in one formalism than we can be sure that there exist polynomial algorithms for this problem in all other formalisms, and if one proves $L \notin P$ in one of the computing model formalisms then this is true for all.
Some further fundamental complexity classes are:

$$\begin{aligned} DLOG &= SPACE(\log_2 n), \\ PSPACE &= \bigcup_{k \in \mathbb{N}} SPACE(n^k), \\ EXPTIME &= \bigcup_{k \in \mathbb{N}} TIME(2^{n^k}). \end{aligned}$$

They satisfy the following relations:

$$DLOG \subseteq P \subseteq PSPACE \subseteq EXPTIME.$$

As we have already mentioned above, the theory of computation provides successful methods for the classification of languages (problems) into recursive (decidable) and nonrecursive (undecidable) ones. Unfortunately, this is not true for the decision whether a language L is in P or not. Usually it is not very hard to prove that $L \in P$, because it is sufficient to find a polynomial MTM accepting L. So, if $L \in P$, then it is rarely a problem to find a polynomial algorithm for L. An exception is the problem of linear programming which was not known to be in P, nor to be not in P for a longer time. In 1979 an ingenious polynomial–time algorithm was found for it [98].

The main difficulty is in proving lower bounds on concrete computing problems. We do not have any mathematical method enabling to prove higher than quadratic lower bounds on the time of restricted computing models (for instance, one–tape TM) or superlinear (higher than linear) lower bounds on

the time of general computing models (register machines). Thus, we cannot really classify computing problems because we are unable to obtain tight lower bounds on their complexity.

To overcome the absence of absolute lower bounds (i.e. the evidence that some computing problems are difficult) Cook [47] has introduced a method enabling to prove so called "**relative**" lower bounds which are viewed as a strong implication of hardness. The idea of this method is based on the reduction between computing problems—a classical mathematical approach to problem solving. The novelty consists in connecting the reduction, of one problem to another one, with the complexity.

Definition 1.12 [47] *Let* $L_1 \subseteq \Sigma_1^*$ *and* $L_2 \subseteq \Sigma_2^*$ *be two languages. We say, that* L_1 *is* **polynomial–time reducible** *to* L_2 *(denoted* $L_1 \leq L_2$*) if there is a polynomial–time bounded* MTM *that for every word* $x \in \Sigma_1^*$ *writes* x_2 *on the first working tape (considered as an output tape here) such that* $x \in L_1 \Longleftrightarrow y \in L_2$*. We say that* L_1 *and* L_2 *are* **polynomial–time equivalent** *if* $L_1 \leq L_2$ *and* $L_2 \leq L_1$*.*

We observe that "L_1 is polynomial–time reducible to L_2" means that L_1 cannot be "much more difficult" than L_2, i.e. if $L_2 \in P$ then L_1 must be in P, too. Thus, if L_1 is **polynomial–time equivalent** to L_2, then $L_1 \in P$ iff $L_2 \in P$.

The first idea behind the relative lower bounds (a strong implication to be difficult) is that if one finds many computing problems polynomial–time equivalent (reducible) each to the other, and no polynomial–time algorithm is known for any of these problems, then we have a large experience (strong implication) that each of these problems is computationally difficult. This idea is still strengthen by considering nondeterminism as follows (we assume that the reader is familiar with the concept of nondeterminism and with nondeterministic MTMs [116]).

Definition 1.13 *Let* M *be a nondeterministic* MTM *with an input alphabet* Σ*. Let* $x \in \Sigma^*$ *and* C *be a computation of* M *on* x*. We denote by* $T_M(C)$ *the length of* C *minus 1. For every* $x \in L(M)$*,* **time complexity of M on x** *is*

$$T_M(x) = \min\{T_M(C) \mid C \quad \textit{is an accepting computation of} \quad M \textit{ on } x\}.$$

The **time complexity of M** *is a partial function* $T_M : \mathbb{N} \to \mathbb{N}$ *such that*

$$T_M(n) = \max\{T_M(x) \mid x \in L(M) \cap \Sigma^n\}.$$

Further, for any $f : \mathbb{N} \to \mathbb{N}, f(n) \geq n$*,*

$$NTIME(f(n)) = \{L \mid L = L(M) \quad \textit{for a nondeterministic} \quad MTM\ M$$
$$\textit{with} \quad T_M(n) \leq f(n)\}.$$

We note that one can represent all possible computations of a nondeterministic MTM M on a given word as a directed possibly infinite tree $Tr_M(w)$ whose nodes are labeled by configurations of M. The root of $Tr_M(w)$ is labeled by the initial configuration $C_0(w)$ of M on w. Each node labeled by C has indegree 1 and outdegree equal to the number of all possible nondeterministic actions from the configuration C. M accepts w if and only if $Tr_M(w)$ contains at least one accepting configuration. We observe that $T_M(x)$ is bounded by the depth of $Tr_M(x)$ because $T_M(x)$ is the minimum of the distances between the root of $Tr_M(x)$ and accepting configurations of $Tr_M(x)$. A deterministic simulation of the work of M on an input w is usually a search for an accepting configuration in $Tr_M(w)$. Since the size of $Tr_M(w)$ may be exponential in the depth of $Tr_M(w)$ (and so exponential in $T_M(w)$) this deterministic search for an accepting computation consists of exponentially many steps according to $T_M(w)$. All simulations of nondeterministic machines by deterministic ones cause an exponential increase of time complexity. This is one reason to believe that

$$P \subset NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k).$$

Another reason for that is that for some mathematical problems the deterministic time corresponds to the complexity of the search for a solution while the nondeterministic time corresponds to the complexity

of verifying whether a given candidate for the solution is a consistent solution. This is because a nondeterministic algorithm can guess the solution (for instance the values of variables over $\{0,1\}$ in a system of equations) in real time and then check whether this guess was correct. For many such problems we cannot find better deterministic algorithms than those searching for a solution by generating exponentially many candidates for it. Thus, we have enough experience to believe $P \subset NP$.

Now, we add these two ideas together by saying that a problem is relatively hard if it is one of the hardest in NP in the following sense.

Definition 1.14 [47] *A language L is called NP–complete if*

(i) $L \in NP$, and

(ii) $\forall L' \in NP$ L' is polynomially reducible to L.

Clearly,

(i) If $P \subset NP$ then every NP–complete language is in $NP - P$ (i.e. intractable).

(ii) If an NP–complete language $L \in P$, then $P = NP$.

A few thousands NP–complete problems are known, and for none of them we have a polynomial–time algorithm. Moreover, we do not know any deterministic time–effective simulation of nondeterministic computations and we do not believe that to find a solution is not much harder than to verify whether a given candidate for the solution is correct. All these facts together provide a large experience supporting the opinion that NP–complete problems do not have polynomial algorithms.

We omit examples and proofs of NP–completeness in this short survey. A nice overview on this topic can be found in [64]. We note only that we can also define complete problems in other classes like $P, PSPACE$ too. If one defines P-complete problems according to the $\log_2 n$–space reduction, then such P–complete problems are candidates for the membership in $P - DLOG$. This question has been found interesting from the very beginning of the complexity theory because each polynomial time computation contains at most a polynomial number of different configurations. To generate any polynomial number of configuration the space $\mathrm{O}(\log_2 n)$ is sufficient. Thus $P = DLOG$ would mean that the memory of every polynomial–time algorithm can be optimized within the $\log_2 n$–space bound. But we conjecture that there are problems in P (exactly the P–complete problems according to $\log_2 n$–space reduction) which require polynomial time as well as polynomial space to be solved.

We conclude this section by giving the fundamental relations among the basic complexity classes.

Definition 1.15 *Let M be a nondeterministic MTM with an input alphabet Σ. Let $x \in \Sigma^*$ and $C = C_1, \ldots, C_k$ be a computation of M on x. We denote by $S_M(C_i)$ the length of the longest content of the working tapes of M in the configuration C_i. $S_M(C) = \max\{S_M(C_i) \mid i = 1, \ldots, k\}$. For every $x \in L(M)$, the* **space complexity of M on x** *is*

$$S_M(x) = \min\{S_M(C) \mid C \quad \textit{is an accepting computation of } M \quad \textit{on} \quad x\}.$$

The **space complexity of M** *is a function $S_M : \mathbb{N} \to \mathbb{N}$ such that*

$$S_M(n) = \max\{S_M(x) \mid x \in L(M) \cap \Sigma^n\}.$$

Further, for any $f : \mathbb{N} \to \mathbb{N}, f(n) \geq n$,

$$NSPACE(f(n)) = \{L \mid L(M) \quad \textit{for a nondeterministic MTM } M \textit{ with}$$

$$S_M(n) \leq f(n)\}.$$

$$NLOG = NSPACE(\log_2 n),$$

$$NPSPACE = \bigcup_{k \in \mathbb{N}} NSPACE(n^k).$$

First, we observe that there exists a space–efficient simulation of nondeterministic computations.

Theorem 1.16 [129] *Let* $f(n) \geq \log_2 n$ *is space–constructible. Then*

$$NSPACE(s(n)) \subseteq SPACE((s(n))^2).$$

Idea of proof. Let $L = L(M)$, $S_M(n) \leq s(n)$. Then there exists a constant c such that the number of different configurations of the nondeterministic MTM M on an input of length n is bounded by $c^{s(n)}$, for every $n \in \mathbb{N}$. So, if M accepts a word w, then $Tr_M(w)$ contains an accepting configuration in the distance at most $c^{s(|w|)}$ from the root.

Let $C_0(w)$ be the initial configuration of M on w. Without loss of generality we can assume that there is a unique accepting configuration C'. We have to test whether C' is reachable from $C_0(w)$ in exactly $c^{s(|w|)}$ steps. We can do it by testing for every configuration C whether C is reachable from $C_0(w)$ in $c^{s(|w|)/2}$ steps and C' is reachable from C in $c^{s(|w|)/2}$ steps. By this approach the space needed to determine whether C_1 is reachable from C_2 in 2^i steps is equal to the space needed to record the configuration C plus the space needed to test if one can reach one configuration from another one in 2^{i-1} steps. Using this approach recursively until one has to test whether one configuration can be reached from another one in one step we have to store at most $\log_2(c^{s(n)}) = \mathrm{O}(s(n))$ configurations. Since each configuration is of size $s(n)$, the simulation uses $O((s(n))^2)$ space.

□

Corollary 1.17 $PSPACE = NPSPACE$

As we have already mentioned, the best known time–bounded simulation causes an exponential increase of time.

Theorem 1.18 *Let* t *be a time–constructible function. Then*

$$NTIME(t(n)) \subseteq \bigcup_{c \in \mathbb{N}} TIME(c^{t(n)}).$$

Idea of proof. Let $L \in NTIME(t(n))$, *i.e.* $L = L(M)$ for some $t(n)$–time bounded nondeterministic NTM. Obviously, M is $t(n)$–space bounded, too. As we already know, there exists a constant d such that the number of configurations of M on an input of the length n is bounded by $d^{t(n)}$. A deterministic MTM can generate all configurations of M reachable from the given initial configuration in time $c^{t(n)}$, for some suitable constant c.

□

The fundamental complexity hierarchy is as follows.

Theorem 1.19

$$DLOG \subseteq NLOG \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

Proof. The following inclusions $DLOG \subseteq NLOG$, $P \subseteq NP$, $PSPACE \subseteq EXPTIME$ are obvious. Since $NP \subseteq NPSPACE = PSPACE$ we obtain $NP \subseteq PSPACE$. $NLOG \subseteq P$, because every nondeterministic $\log_2 n$–space bounded MTM has at most a polynomial number of configurations in any computation tree for an input of length n.

□

All inclusions in Theorem 1.19 are believed to be proper, but nobody has proven it for any of them. To prove at least one proper inclusion or equality in this hierarchy is one of the central open problems of theoretical computer science. One supposes that this problem is very hard because it can be proven that the usual machine simulations, as a method for proving equality between two language (complexity) classes, and the diagonalization, as a method for proving non-equality among complexity classes, do not work for any of the inclusions of the fundamental complexity hierarchy. How can one prove that these methods do not work? We illustrate this approach on the most important problem $P =?NP$ (see, for instance, [64, 7, 8, 116]).

An **oracle machine** is a pair (M, L), denoted M^L too, where M is a MTM with one additional oracle tape and L is an oracle. M has special states $q_?, q_Y, q_N$, and if M is in the state $q_?$ then M without looking on the content of its tapes, enters the state $q_Y(q_N)$ if the content of the oracle tape is (not) in L. A move from $q_?$ to q_Y or q_N according to L is considered as one step of M^L. After this, the content of the oracle tape is erased. We denote $L(M^A)$ the language accepted by the oracle machine M^A and

$$P^A = \{L(M^A) \mid M^A \text{ is a polynomial–time bounded oracle machine}\},$$

$$NP^A = \{L(M^A) \mid M^A \text{ is a nondeterministic polynomial–time bounded oracle machine}\}.$$

Theorem 1.20 [6] *There exist two languages A, B such that*

(i) $P^A = NP^A$

(ii) $P^B \subset NP^B$

Idea of proof. To prove (i) it is sufficient to take a $PSPACE$–complete language A. For this choice of A one can easily observe that $P^A = PSPACE = NPSPACE = NP^A$. The proof of (ii) is much more involved. The idea is to find an oracle B and a language $L \in NP^B - P^B$ in such a way $L = \{0^i \mid B \text{ contains a word of the length } i\}$ and $|B \bigcap \{0,1\}^n| \leq 1$, for any $n \in \mathbb{N}$. A nondeterministic MTM M^B can accept L in linear time by guessing the word x from $B \bigcap \{0,1\}^n$, for the input 0^n, and by verifying its nondeterministic decisions by asking the oracle B whether $x \in B$. If the oracle B is cleverly constructed then no polynomial–time bounded MTM can find in polynomial time the candidate $x \in B \bigcap \{0,1\}^n$ among the 2^n words of the length n.

□

The above theorem shows that we cannot prove $P = NP$ by any usual simulation and $P \neq NP$ by any typical diagonalization. Suppose one could "simulate" nondeterministic polynomial–time bounded MTMs by polynomial–time bounded MTMs. All known simulations of a machine M_1 by a machine M_2 remain valid if one attaches the same oracle to both machines M_1 and M_2. This means that $P = NP$, proven by such simulation implies $P^C = NP^C$, for every oracle C. But (ii) of Theorem 1.20 contradicts this. The same is true for the diagonalization method. All usual diagonalization proofs for the non-equality $G \neq H$ between two language classes G and H would also work when oracles are attached ($G^L \neq H^L$, for every language L). But we have an oracle A such that $P^A = NP^A$ which means that usual diagonalizations cannot help to separate P from NP.

Finishing this section we note that we omit to present two large intersections of formal language theory and complexity theory here. One is devoted to the characterizations of basic complexity classes by different kind of automata (multihead automata, pushdown machines, etc.). A nice overview about this topic can be found by Lange [100]. Another topic omitted is devoted to the complexity of classical problems of formal language theory like string matching, parsing, etc. Each one of this research areas involves enough result for a monograph and according to the size restrictions of this chapter we do not try to give a survey of them.

1.3.3 Hard Problems and Probabilistic Computations

What to do if one has proven the evidence or a strong implication that a given computing problem is computationally difficult? One can stop any attempt to solve the problem by giving the mathematical arguments justifying why the problem cannot be solved on a computer. This may be good for pure theory but quite unsatisfactory if there is a large practical interest to have a program solving the problem. In this case one uses one of the following approaches.

1) **Deterministic Algorithms**

Let us assume that our problem has exponential time complexity. Then one can still try to find a MTM M (algorithm, program) which solves the problem in time complexity close to $T_M(n) = \frac{1}{100} \cdot 2^{n/50}$. It means, that in spite of the fact that M has an exponential complexity,

one can still effectively compute results for inputs of several hundreds bits. Thus, if the lengths of the input from practical applications are in the range where $T_M(n)$ is not too large, then one has an useful algorithmical solution for a hard problem.

2) **Problem Restrictions**

Usually, you do not need to solve the given computing problem in the generality of its original formulation. In several cases some additional restrictions essentially decrease the computational difficulty of the problem. The most typical case is to restrict the set of potential inputs only (for instance, the Post Correspondence Problem starts to be decidable if one considers inputs over one–letter alphabet only, the Satisfiability Problem is in P if one restricts the inputs to conjunctive normal formulas in which the length of the elementary disjunctions are bounded by two).

3) **Approximate Algorithms**

If our difficult problem is an optimization problem, one can make it easier searching for a solution which is not too far from the optimum, but it is not necessarily the optimum. There are NP–complete optimization problems for which good approximations of optimal solutions can be found in polynomial time. Since the practice is often satisfied with approximate solutions this is one of the most successful methods for the design of algorithms for hard problems.

4) **Probabilistic Algorithms**

Probabilistic algorithms are based on nondeterministic ones. Each nondeterministic step (branching) is interpreted as a random decision (tossing coin). While for nondeterministic computations an accepting configuration in the computation tree has been sufficient to accept the input, for probabilistic computations we require a "sufficiently large" probability to reach an accepting configuration (correct output). Thus, instead of surely correct outputs produced by deterministic algorithms one computes an output whose probability to be correct is large enough. Usually the probabilistic algorithms allow repeated runs on the same input due to which the probability to get the right answer tends to 1 with the size of the input.

5) **Probabilistic Approximate Algorithms**

A combination of the two previous methods leads to algorithms providing with high probability solutions of optimization problems which are very close to optimum.

6) **Heuristics**

Heuristic algorithms are similar to probabilistic ones in that they make random decision during their computations, too. The difference is in that we are unable to prove that heuristic algorithms provide good solutions (outputs) despite of the fact that they successfully work in some practical applications (at least for most of the inputs generated in practice). The ideas of the algorithm design are based on some analogies to efficient optimization processes running in the nature (biology, physics). Two of the most popular representants are genetic algorithms and simulated annealing. But theoretical as well as experimental results show that the methods previously described are much more reliable than heuristic ones and the use of heuristic methods is recommended only in cases in which one is not able to find an efficient solution by the previous five approaches.

Note that the above approaches may be also mixed. For instance, we have polynomial–time approximate algorithms for Traveling Salesman Problem in Euclidean space.

Out of the six approaches mentioned above we shall give more details about the probabilistic one only. There are two reasons for this: i) one can use the formalism of formal language theory to describe probabilistic algorithms, and ii) the power of randomized computations is one of the topics of main interest in the recent complexity theory.

We distinguish two types of probabilistic MTMs (algorithms) called Las Vegas MTMs and Monte Carlo MTMs. Las Vegas algorithms always give the right result and their complexity on an input x is measured as the "weighted average" over all computations on x. Monte Carlo algorithms give the right result with some probability greater than $1/2$ (i.e. some computations may lead to wrong results) and one usually considers that the time complexity is measured by the depth of the computation tree. In what follows we give more details about these two models of probabilistic computations in the formalism of formal language theory.

Definition 1.21 *A nondeterministic MTM M is called a* **Las Vegas MTM M** *if*

(i) the degree of nondeterminism is bounded by two (i.e. all computation trees are binary trees), and

(ii) for every $x \in L(M)$ $[x \notin L(M)]$ all leaves of $Tr_M(x)$ are accepting [rejecting] configurations.

Let $D \in Tr_M(x)$ denote the fact that D is a computation of M on x corresponding to a path in $Tr_M(x)$ leading from the root to a leaf of $Tr_M(x)$. The probability of executing $D \in Tr_M(x)$ is $Prob(D) = 2^k$, where k is the number of nondeterministic choices of D. Let $|D|$ denote the length of D. **Las Vegas time complexity of M on an input x** *is*

$$LVT_M(x) = \sum_{D \in Tr_M(x)} Prob(D) \cdot |D|.$$

Las Vegas time complexity of M *is*

$$LVT_M(n) = \max\{LTV_M(x) \mid |x| = n\}.$$

We observe that polynomial Las Vegas algorithms are very useful in practical applications because they are reliable (they compute the right outputs), and the outputs are computed in polynomial time with a high probability.

Definition 1.22 *A nondeterministic MTM M is called* **one–sided error Monte Carlo MTM** *if*

(i) the degree of nondeterminism is bounded by two,

(ii) there is a function $f : \mathbb{N} \to \mathbb{N}$ such that all computations on the inputs of the length n have length $f(n)$ and each one uses exactly $f(n) - 1$ nondeterministic guesses (i.e. all $2^{f(n)-1}$ computations have the same probability)

(iii) if $x \in L(M)$, then at least $2^{f(n)-2}$ computations on x finish in accepting states, and

(iv) if $x \notin L(M)$, then all computations on x finish in rejecting states.

A nondeterministic MTM M is called **two–sided error Monte Carlo MTM** *if it satisfies (i), (ii), and*

(v) there exists a constant ϵ such that if $x \in L(M)(x \notin L(M))$, then more than $(2^{f(n)-1}/2)(1+\epsilon)$ computations finish in accepting (rejecting) states.

We observe that Monte Carlo algorithms are not reliable as Las Vegas ones, but one–sided error Monte Carlo algorithms are still extremely useful in solving decision problems. The reason for this claim is that we can iterate the algorithm on the same input a certain number of times with independent random choices in order to make the error probability arbitrarily small. For instance, k runs of a one–sided error Monte Carlo algorithm on the same input produce k answers. If at least one is "yes", then the answer "yes" is certainly correct. If all answers are "no", then we can conclude that the answer is "no" with probability $1 - (\frac{1}{2})^k$. For sufficiently large k the error 2^{-k} is smaller than the probability that our hardware/software fails during the execution of the k runs of the algorithm.

A nice example of the Monte Carlo approach is the recognition of the language

$$Com = \{w \in 1\{0,1\}^* \mid w \text{ is the binary code of a composite number}\}.$$

Obviously, the simple approach based on trying to divide the given input number n by all numbers $n \in \{1, 2, \ldots, \lceil \sqrt{n} \rceil\}$ has exponential complexity in the input length $\lceil \log_2 n \rceil$. We do not know any polynomial–time deterministic algorithm for Com, but one can construct an $O((\log_2 n)^5)$ algorithm assuming the extended Riemann hypothesis [111]. In what follows we present a polynomial time one–sided error Monte Carlo algorithm for Com [136, 124]. It is based on the following two results of the number theory. Let $GCD(a, b)$ denote the greatest common divisor of the numbers a and b.

Lemma 1.23 *If a positive integer $n > 2$ is composite, then there exists an integer $a, 1 \leq a \leq n$, such that*

(i) $a^{n-1} \not\equiv 1 \pmod{n}$ *or*

(ii) there exists an integer i *such that* 2^i *divides* $n-1$ *and* $1 < GCD(a^{(n-1)/2^i} - 1, n) < n$.

Definition 1.24 *Let* n *be a positive integer. An iteger* $a, 1 \leq a \leq n$, *is called a* **compositeness witness for n** *if (i) or (ii) of Lemma 1.22 hold.*

Lemma 1.25 *If* $n \geq 3$ *is an odd integer, then at least* $(n-1)/2$ *distinct integers from* $\{1, 2, \ldots, n-1\}$ *are compositeness witnesses for* n.

So, if for a fixed composite number n one randomly chooses a number $a \in \{1, 2, \ldots, n-1\}$, then the probability that a is a compositeness witness for n is at least $1/2$. On the other hand, if n is a prime, then no $a \in \{1, 2, \ldots, n-1\}$ is a compositeness witness for n. This two facts yield a Monte Carlo MTM (algorithm) M working as follows:

1) If n is even, then M accepts.

2) If n is odd, then M randomly chooses a number $a \in \{1, 2, \ldots, n-1\}$. If a is a compositeness witness for n then M accepts. Otherwise, M rejects.

The fact whether a is a compositeness witness for n can be checked in polynomial time. Obviously, if M accepts, then the input is certainly composite. If M rejects, then one can state that n is prime with probability at least $1/2$. As we have already seen, if one wishes a smaller error probability, the algorithm has to be executed repeatedly with the same input n.

We note that there even exists a Las Vegas algorithm deciding whether a given number is prime or not [3], but it is too technical to be presented here.

Probabilistic algorithms are very efficient, but only "probably correct". However, many probabilistic algorithms—in particular, the above primality probabilistic algorithm—can be "theoretically" converted into rigorous, deterministic algorithms provided a sufficiently long random string[30] input is supplied [44, 30]; this possibility is only "theoretical" as the set of random strings is not recursively enumerable [24].

A recent quantum algorithm[31] (based on Fourier transformation) proposed by Shor [131] seems to indicate that primality can be checked in polynomial time on a quantum computer.

¿From a practical point of view one can consider the class of languages (problems) recognized (solved) in polynomial time by some probabilistic algorithms to be the class of practically (solvable) languages (problems). Because of this it is reasonable to consider the following language classes, introduced by Gill [68]:

$$\begin{aligned} ZPP &= \{L \mid L = L(M), \quad \text{for a Las Vegas } MTM\ M \text{ working in polynomial time}\}, \\ R &= \{L \mid L = L(M), \quad \text{for an one–sided error Monte Carlo } MTM \text{ working in polynomial time}\}, \\ BPP &= \{L \mid L = L(M), \quad \text{for a two–sided error Monte Carlo } MTM \text{ working in polynomial time}\}. \end{aligned}$$

The following relations can be easily proven.

Theorem 1.26

$$P \subseteq ZPP = R \cap coR \subseteq R \subseteq NP,$$

$$R \cup coR \subseteq BPP \subseteq PSPACE.$$

[30] See Section 1.4.3.

[31] One decisive feature of quantum computation is *parallelism*: during a computation cycle a quantum computer is processing all coherent paths *at once*. See more in [4, 9, 10, 32, 61, 62, 52, 53, 54, 15, 16, 12, 13, 142, 143].

Whether some of the inclusions above are proper or not is unknown. So, we do not know any NP–complete language recognizable by one–sided error Monte Carlo in polynomial time. But we have either examples of problems in P for which probabilistic algorithms are more effective than the best deterministic algorithms or we have polynomial–time probabilistic algorithms for problems whose membership in P is unknown. Thus, we do not conjecture that a NP–complete problem may have a polynomial–time probabilistic solution.

1.4 Program-Size Complexity

1.4.1 Dynamic vs Program-Size Complexities

Let us recall that a Chaitin computer C is a partial recursive function carrying strings (on A) into strings such that the domain of C is prefix-free. If C is a computer, then T_C denotes its time complexity, i.e. $T_C(x)$ is the running time of C on the entry x, if x is in the domain of C; $T_C(x)$ is undefined in the opposite case.

The property of universality discussed in Section 1.2.3 can be presented, in a stronger form, as follows:

Theorem 1.27 [Invariance Theorem] *There exists a universal Chaitin computer U with the property that for every computer C there exists a constant $sim(U,C)$—which depends upon U, C—such that in case $C(x) = y$, there exists*[32] *a string x' satisfying the following conditions:*

$$U(x') = y, \tag{1.2}$$

$$|x'| \leq |x| + sim(U,C). \tag{1.3}$$

Indeed, let (C_i) be a gödelization of all Chaitin computers and define $U(a_1^i a_2 u) = C_i(u)$, for all strings u (here a_1, a_2 are two distinct letters from the alphabet A).

The *program-size* or *Chaitin complexity* associated to the Chaitin computer C is the partial function H_C defined by $H_C(x) = \min\{|u| \in A^* \mid C(u) = x\}$ iff such a u does exist. The above result can be re-phrased as follows:

Theorem 1.28 *There exists a Chaitin computer U such that for every Chaitin computer C there effectively exists a constant c—depending upon U and C—such that for all strings x,*

$$H_U(x) \leq H_C(x) + c.$$

So, U not only simulates every Chaitin computer C, but the simulation is asymptotically optimal. Is it possible to prove a similar result for a dynamical complexity, e.g. time complexity? The answer is *negative* and here is a proof (cf. [25]).

Assume, for the sake of a contradiction, that U can simulate every other computer (1.2), in a shorter time. Formally, to equation (1.2) we add the constraint:

$$T_U(x') < T_C(x). \tag{1.4}$$

For every string x in the domain of U let

$$t(x) = \min\{T_U(z) \mid z \in A^*, U(z) = U(x)\}, \tag{1.5}$$

i.e. $t(x)$ is the minimal running time necessary for U to produce $U(x)$.[33]

Next define the *temporal canonical program (input)* associated with x to be the first (in quasi-lexicographical order) string $x^{\#}$ satisfying the equation (1.5):

$$x^{\#} = \min\{z \in dom(U) \mid U(z) = U(x), T_U(z) = t(x)\}.$$

So,

$$U(x^{\#}) = U(x), \text{ and } T_U(x^{\#}) = t(x).$$

[32] And can be effectively constructed.

[33] Actually, $t(x)$ is not computable.

As the universal computer U can simulate itself, it follows from (1.4) that there exists a string x' such that $U(x') = U(x^{\#}) = U(x)$, and $T_U(x') < T_U(x^{\#}) = t(x)$, which is false.

The reason for the above phenomenon can be illustrated by showing the existence of "small-sized" computers requiring "very large" running times. To this aim we use the information-theoretic version of the Busy Beaver function Σ. For every natural m let us denote by $string(m)$ the mth string in quasi-lexicographical order, and let $\Sigma(n)$ be the largest natural number whose algorithmic information content is less than or equal to n, i.e.

$$\Sigma(n) = \max\{m \in \mathbb{N} \mid H_U(string(m)) \leq n\}.$$

Chaitin ([39], 80-82, 189) has shown that Σ grows larger than any recursive function, i.e. for every recursive function f, there exists a natural number N, which depends upon f, such that $\Sigma(n) \geq f(n)$, for all $n \geq N$:[34] indeed, there is a constant q such that every program of length n either halts in time less than $\Sigma(n+q)$, or else it never halts.

As $H_U(string(\Sigma(n))) \leq n$, it follows that $U(y_n) = string(\Sigma(n))$, for some string y_n of length less than n. This program y_n takes, however, a huge amount of time to halt: there is a constant c such that for large enough n, $U(y_n)$ takes between $\Sigma(n-c)$ and $\Sigma(n+c)$ units of time to halt. To conclude, the equation (1.2) is compatible with (1.3), but incompatible with (1.4).[35]

1.4.2 The Halting Problem Revisited

Can the halting problem be solved if one could compute program-size complexity? The answer is **yes** and here is a proof (cf. [43]).

Fix a universal Chaitin computer U and denote H_U simply by H. We have seen that if an n–bit program p halts, then the time t it takes to halt satisfies $H(t) \leq n+q$. So if p has run for time T without halting, and if for all $t \geq T$ one has $H(t) > n+q$, then p will never halt.

Consider the recursively enumerable set of all true upper bounds on H, $Ch = \{(x,k) \in A^* \times \mathbb{N} \mid \{H(x) \leq k\}$. Imagine enumerating this set, and keep track of the running time. Assuming that H is computable, compute $H(x)$ for each n–bit string x. Then enumerate Ch until we get the best possible upper bound on $H(x)$ for all n–bit strings x. Let $\beta(n)$ be defined to be the time it takes to enumerate enough of the set of all true upper bounds on program-size complexity until one obtains the correct value of $H(x)$ for all n–bit strings x. If one is given n and a number greater than $\beta(n)$, one can determine an n–bit bit string x_{max} with maximum possible complexity

$$H(x_{max}) = n + H(string(n)) + O(1).$$

Thus any number $k \geq \beta(n)$ has

$$n + H(string(n)) - q' < H(x_{max}) \leq H(string(k)) + H(string(n)) + q''$$

and

$$H(k) > n - q' - q''.$$

Thus we can use $\beta(n)$, which is computable from H, to solve the halting problem as follows: an n–bit program p halts iff it halts before time $\beta(n+q+q'+q'')$.

As a bonus we derive the fact that the information-theoretic Busy Beaver function Σ is computable from H: the formula

$$\Sigma(n) = \max\{U(p) \mid |p| \leq n\},$$

proves that Σ is computable relative to the halting problem which, in turn, is computable from H.

[34] The difficulty might be also explained by the fact that Σ grows as fast as the least time necessary for all programs of length less than n that halt on U to stop, [34].

[35] Incidentally, the above discussion shows that, contrary to what Penrose has suggested (see [121], p. 560), there is no incompatibility between the strong determinism and computability: it is indeed impossible for a (universal) machine to "learn its own theory".

1.4.3 Random Strings

Consider the number

one million, one hundred one thousand, one hundred and one.

This number appears to be

the first number not nameable in under ten words.

However, the above expression has only **nine** words, pointing out a naming inconsistency: it is an instance of Berry's paradox.

It follows that the property of nameability[36] is inherently ambiguous and, consequently, too powerful to be freely used.

Of course, the above analysis is rather vague. We can make it more rigorous by using a universal Chaitin computer U. Some programs for U specify positive integers: when we run such a program on U the computation eventually halts and produces the number. In other words, a program for U "specifies" a positive integer in case after running a finite amount of time it prints the number. What we get is the statement:

THE FIRST POSITIVE INTEGER THAT CANNOT BE SPECIFIED BY A PROGRAM FOR U WITH LESS THAN N BITS.

However, **there is a program for U, of length $\log N + c$, for calculating the number that supposedly cannot be specified by any program of N bits!** And, of course, for large N, $\log N + c$ is much smaller than N.

Suppose that persons A and B give us a sequence of 32 bits each, saying that they were obtained from independent coin flips. If A gives the string $x = 01101000100110101101100110100101$ and B gives the string $y = 00000000000000000000000000000000$, then we would believe A and would not believe B: the string x *seems* to be random, but the string y does not. Why? The strings are extremely different from the point of view of *regularity*: the second string has a maximum regularity which allows us to express it in a very compact way, *only zeros*, while the first one appears to have no shorter definition at all.[37]

Classical probability theory is not sensitive to the above distinction, as strings are all equally probable. Laplace (1749-1827) was, in a sense, aware of the above paradox when he wrote (cf. [101], pp.16-17):

In the game of heads and tails, if head comes up a hundred times in a row then this appears to us extraordinary, because after dividing the nearly infinite number of combinations that can arise in a hundred throws into regular sequences, or those in which we observe a rule that is easy to grasp, and into irregular sequences, the latter are incomparably more numerous.

Non-random strings are strings possessing some kind of regularity, and since the number of all those strings (of a given length) is **small**, the occurrence of such a string is **extraordinary**. The overwhelming majority of strings have hardly any "computable" regularities—they are random. Randomness means the absence of any compression possibility; it corresponds to maximum information content (because after dropping any part of the string, there remains no possibility of recovering it). Borel (1909), Von Mises (1919), Ville (1939), and Church (1940) elaborated on this idea, but a formal model of irregularity was not found until the mid-1960s in the work of Kolmogorov [99] and Chaitin [33]. Currently, it appears that the model due to Chaitin [35], which will be briefly discussed in what follows, is the best adequate model (cf. [37, 41, 24]).

A string is random in case it has maximal program-size complexity when compared with the program-size complexity of all strings of the same length. As for every $n \in \mathbb{N}$, one has:

$$\max_{x \in A^n} H(x) = n + H(string(n)) + \mathrm{O}(1),$$

[36] Another famous example refers to the classification of numbers as *interesting* or *dull*. There can be no dull numbers: if they were, the first such number would be *interesting* on account of its dullness.

[37] The distinction between regular and irregular strings becomes sharper and sharper for longer and longer strings (e.g. it is easier to specify the number

$$10^{10^{10^{10}}}$$

than the first 100 digits of π).

and is led to the following definition: a string $x \in A^*$ is **(Chaitin)** m–**random** (m is a positive integer) if $H(x) \geq \Gamma(|x|) - m$; x is **(Chaitin) random** if it is 0–random. Here $\Gamma(n) = \max\{H(x) \mid x \in A^*, |x| = n\}$.

The above definition depends upon the fixed universal computer U; the generality of the approach comes from the Invariance Theorem. Obviously, for every length n and for every $m \geq 0$ there exists a m–random string x of length n.

It is worth noticing that randomness is an asymptotic property: the larger is the difference between $|x|$ and m, the more random is x. There is no sharp dividing line between randomness and pattern, but it was proven that all m–random strings x with $m \leq H(string(|x|))$ have a true random behaviour [24].

A random string cannot be algorithmically compressed. Incompressibility is a non-effective property: no individual string, except finitely many, can be proven random. Under these circumstances it is doubtful that we can exhibit an example of a random string, even though the vast majority of strings are random. However, we can describe a non-effective construction of random strings. We start by asking ourselves: How many strings x of length n have maximal complexity, i.e. $H(x) = \Gamma(|x|)$? Answer: There exists a natural constant $c > 0$ such that

$$\gamma(n) = \#\{x \in A^* \mid |x| = n, H(x) = \Gamma(|x|)\} > Q^{n-c},$$

for all natural n; here Q is the cardinality of the alphabet A.[38]

Now fix a natural base $Q \geq 2$, and write $\gamma(n)$ in base Q. The resulting string—over the alphabet containing the letters $0, 1, \ldots, Q-1$—is itself **random** (cf. [42]).[39]

1.4.4 From Random to Regular Languages

Recall that $\{string_Q(n) \mid n \geq 0\}$ is the enumeration of all strings over the alphabet A (having $Q \leq 2$ elements) in quasi-lexicographical order. A language $L \subset A^*$ is described by its binary characteristic sequence $\mathbf{l}, \mathbf{l}_i = 0$ iff $string_Q(i) \in L$. A language L is **random** if its characteristic function is random (as an infinite, binary sequence). Denote by **RAND** the set of random languages. In what follows we shall present different characterizations of random languages.

We start with a piece of notation. For every sequence $\mathbf{l} = l_1 l_2 \ldots l_n \ldots$ we denote by $\mathbf{l}(n) = l_1 l_2 \ldots l_n$, the prefix of length n of $\mathbf{l}$.

The unbiased discrete probability on $B = \{0, 1\}$ is defined by the function

$$h : 2^A \to [0, 1], h(X) = \frac{\#X}{2},$$

for all subsets $X \subset B$. This uniform measure induces the product measure μ on the set of binary infinite sequences B^ω: for all strings $x \in B^*$, $xB^\omega = \{\mathbf{y} \in B^\omega \mid \mathbf{y}(|x|) = x\}$, and

$$\mu(xB^\omega) = 2^{-|x|}.$$

If $x = x_1 x_2 \ldots x_n \in B^*$ is a string of length n, then $\mu(xB^\omega) = 2^{-n}$ and the expression $\mu(\ldots)$ can be interpreted as "the probability that a sequence $\mathbf{y} = y_1 y_2 \ldots y_n \ldots \in B^\omega$ has the first element $y_1 = x_1$, the second element $y_2 = x_2, \ldots$, the nth element $y_n = x_n$". Independence means that the probability of an event of the form $y_i = x_i$ does not depend upon the probability of the event $y_j = x_j$.

Every open set $G \subset B^\omega$, i.e. a set $G = \cup_{x \in X} xB^\omega$, for some prefix-free subset $X \subset B^*$, is μ measurable and

$$\mu(G) = \sum_{x \in X} 2^{-|x|}.$$

Finally, $S \subset B^\omega$ is a *null set* in case for every real $\varepsilon > 0$ there exists an open set G_ε which contains S and $\mu(G_\varepsilon) < \varepsilon$. For instance, every enumerable subset of B^ω is a null set.

[38] How large is c? Out of Q^n strings of length n, at most $Q + Q^2 + \cdots + Q^{n-m-1} = (Q^{n-m} - 1)/(Q - 1)$ can be described by programs of length less than $n - m$. The ratio between $(Q^{n-m} - 1)/(Q - 1)$ and Q^n is less than 10^{-i} as $Q^m \geq 10^i$, irrespective of the value of n. For instance, this happens in case $Q = 2$, $m = 20$, $i = 6$; it says that *less than one in a million among the binary strings of any given length is not* 20*–random.*

[39] This string is random because it represents a large number.

A property P of sequences $\mathbf{x} \in B^\omega$ is *true almost everywhere in the sense of* μ in case the set of sequences not having the property P is a null set. The main example of such a property was discovered by Borel and it is known as the *Law of Large Numbers*. For every sequence $\mathbf{x} = x_1 x_2 \dots x_m \dots \in B^\omega$ and natural number $n \geq 1$ put

$$S_n(\mathbf{x}) = x_1 + x_2 + \cdots + x_n.$$

Then,

> *The limit of* S_n/n, *when* $n \to \infty$, *exists almost everywhere in the sense of* μ *and has the value* $1/2$.

It is clear that a sequence satisfying a property false almost everywhere with respect to μ is very "particular". Accordingly, it is tempting to try to say that

> a sequence $\mathbf{x}$ is "random" iff it satisfies every property true almost everywhere with respect to μ.

Unfortunately, we may define for every sequence $\mathbf{x}$ the property $P_{\mathbf{x}}$ as following:

> $\mathbf{y}$ *satisfies* $P_{\mathbf{x}}$ *iff for every* $n \geq 1$ *there exists a natural* $m \geq n$ *such that* $x_m \neq y_m$.

Every $P_{\mathbf{x}}$ is an asymptotic property which is true almost everywhere with respect to μ and $\mathbf{x}$ does not have property $P_{\mathbf{x}}$. Accordingly, *no sequence can verify all properties true almost everywhere with respect to* μ. The above definition is vacuous!

However, there is a way—due to Martin-Löf (cf. [108])—to overcome the above difficulty: we consider not *all* asymptotic properties true almost everywhere with respect to μ, but only a *sequence* of such properties. In this context the important question becomes: *What sequences of properties should be considered?* Clearly, the "larger" the chosen sequence of properties is, the "more random" will be the sequences satisfying that sequence of properties. As a constructive selection criterion seems to be quite natural, we will impose the minimal computational restriction on objects: each set of strings will be recursively enumerable, and every convergent process will be regulated by a recursive function.

A **constructively open set** $G \subset B^\omega$ is an open set $G = XB^\omega$ for which $X \subset B^*$ is recursively enumerable. A **constructive sequence of constructively open sets**, for short, **c.s.c.o. sets** is a sequence $(G_m)_{m \geq 1}$ of constructively open sets $G_m = X_m B^\omega$ such that there exists a recursively enumerable set $X \subset B^* \times \mathbb{N}$ with

$$X_m = \{x \in B^* \mid (x, m) \in X\},$$

for all natural $m \geq 1$. A **constructively null set** $S \subset B^\omega$ is a set such that there exists a c.s.c.o. sets $(G_m)_{m \geq 1}$ for which

$$S \subset \bigcap_{m \geq 1} G_m,$$

and

$$\lim_{m \to \infty} \mu(G_m) = 0, \text{ constructively},$$

i.e. there exists an increasing, unbounded, recursive function $H : \mathbb{N} \to \mathbb{N}$ such that $\mu(G_m) < 2^{-k}$ whenever $m \geq H(k)$.

It is clear that $\mu(S) = 0$, for every constructive null set, but the converse is not true.

Here are some properties equivalent to randomness:

Theorem 1.29

- **(Martin-Löf)** *A sequence* $\mathbf{x} \in B^\omega$ *is random iff* $\mathbf{x}$ *is not contained in any constructive null set.*
- **(Chaitin-Schnorr)** *A sequence* $\mathbf{x} \in B^\omega$ *is random iff there exists a natural* $c > 0$ *such that*

  $$H(\mathbf{x}(n)) \geq n - c,$$

  for all natural $n \geq 1$.

- **(Chaitin)** *A sequence* $\mathbf{x} \in A^\omega$ *is random iff*

$$\lim_{n\to\infty}(H(\mathbf{x}(n)) - n) = \infty.$$

Random languages have remarkable properties:

Theorem 1.30

- [24] *No random language is recursively enumerable; in fact, every random language is immune in the sense that it contains no infinite recursively enumerable language.*
- [24] *Random languages are closed under finite variation.*
- [19, 18] *If* $L = \{string_Q(i) \mid \mathbf{l}_i = 1\}$ *is random and* $f : \mathbb{N} \to \mathbb{N}$ *is recursive and one-one, then the language* $\{string_Q(i) \mid \mathbf{l}_{f(i)} = 1\}$ *is also random.*
- [19, 18] *Let* $L \subset A^\omega$ *be a union of constructively closed sets that is closed under finite variation. Then*

$$\mu(L) = 1 \text{ iff } X \cap \mathbf{RAND} \neq \emptyset.$$

- [19, 18] *Let* L *be an intersection of constructively open sets that is closed under finite variation. Then*

$$\mu(L) = 1 \text{ iff } \mathbf{RAND} \subset L.$$

- [63, 24] *Every language is Turing reducible to a random language.*

To characterize recursive and regular languages by means of descriptional complexity we need to introduce the *blank-endmarker or Kolmogorov–Chaitin complexity* (see [99, 33]) associated to a universal Turing machine TM: $K(x) = \min\{|y| \mid TM(y) = x\}$.

Recursive languages can be characterized as follows

Theorem 1.31 ([40]) *A language* $L \subset A^*$ *is recursive iff one of the following two equivalent conditions holds true:*

- *there exists a constant* c *(depending upon* L*) such that* $K(l_1 l_2 \cdots l_n) < K(string_Q(n)) + c$*, for all positive integers* n*,*
- *there exists a constant* c' *(depending upon* L*) such that* $K(l_1 l_2 \cdots l_n) < \log_2 n + c'$*, for all positive integers* n*.*

It is worth noticing that recursive sequences cannot be characterized by the property "$H(l_1 l_2 \cdots l_n) < H(string_Q(n)) + \mathrm{O}(1)$", as shown in [135] (see also [36]).

The above result can be used to describe regular languages. For $L \subset A^*$ and $x \in A^*$ let $\alpha = \alpha_1\alpha_2 \cdots \alpha_n \cdots$ be the characteristic sequence of the language $L_x = \{y \in A^* \mid xy \in L\}$.

Theorem 1.32 [104] *A language* $L \subset A^*$ *is regular iff one of the following equivalent statements holds true:*

- *There is a constant* c *(depending upon* L*) such that for all* $x \in A^*$ *and all positive integers* n*,* $K(\alpha(n)) \leq K(string_Q(n) + c$*.*
- *There is a constant* c' *(depending upon* L*) such that for all* $x \in A^*$ *and all positive integers* n*,* $K(\alpha(n)) \leq \log_Q n + c'$*.*

Program-size pumping lemmas for regular and context-free languages have been proposed in [104]; however, they are currently superseded by pumping lemmas based on other tools (see for example [149]).

1.4.5 Trade-Offs

In this section we give some examples of trade-offs between program-size and computational complexities.

We start with an example, discussed in [125], of a language having a very low program-size complexity, but a fairly high computational complexity. Let $\alpha_i, i = 0, 1, \ldots$ be an ordering of all regular expressions over the alphabet A. A positive integer i is *saturated* iff the regular language denoted by α_i equals A^*. A real number

$$r = 0.a_0a_1\cdots$$

in base Q (the cardinality of A) is now defined by putting $a_i = 1$ iff i is saturated.

It is obvious that r is non-empty, as some indices i are saturated. The language L_r of all prefixes of the sequence $a_0a_1\cdots$ has a low program-size complexity; more precisely, there exists a constant k such that $H(a_0a_1\cdots a_n) \leq \log_Q n + k$. On the other hand, the computational complexity of the membership problem in L_r is very high: to decide the membership of a string of length i one has to prove that all the i regular languages involved have an empty complement. If instead of r we consider π, then the corresponding language L_π has the same (within an additive constant) program-size complexity, but the computational complexity of the membership in L_r is higher than the membership in L_π.

To an infinite sequence $\mathbf{x}$ we associate also the languages

$$S(\mathbf{x}) = \{u \in A^* \mid \mathbf{x} = vu\mathbf{y},\ v \in A^*,\ \mathbf{y} \in A^\omega\},$$

and

$$P(\mathbf{x}) = \{u \in A^* \mid \mathbf{x} = u\mathbf{y},\ \mathbf{y} \in A^\omega\},$$

that is, $S(\mathbf{x})$ is the set of all finite substrings of $\mathbf{x}$, and $P(\mathbf{x})$ is the set of all finite prefixes of $\mathbf{x}$.

A way to measure the complexity of an infinite sequence $\mathbf{x}$ is to evaluate the complexity of the language $P(\mathbf{x})$. If $\mathbf{x}$ is random, then $S(\mathbf{x}) = A^*$, but the converse relation is obviously false. Following [96], call a sequence $\mathbf{x}$ *disjunctive* if $S(\mathbf{x}) = A^*$.

Non-recursive disjunctive sequences have been constructed in [96]. Chaitin's Omega Number [39] is Borel normal in any base and, therefore, disjunctive in any base. More generally, every random sequence is Borel normal and, hence, disjunctive (cf. [24]).

Are there recursive disjunctive sequences? A direct application of Rabin's Theorem (see [23]) shows the existence of disjunctive sequences $\mathbf{x}$ such that $P(\mathbf{x})$ is recursive but arbitrarily complex. A more effective construction of a recursive disjunctive sequence can be obtained by concatenating, in some recursive order, all strings over A. This construction raises two questions: What is the "complexity" of such a sequence? Are there "simpler" ways to produce disjunctive sequences? If we measure the complexity of a sequence $\mathbf{x}$ by the complexity of the language $P(\mathbf{x})$, then one can prove (see [29]) that the language associated to the sequence consisting of all strings over the binary alphabet arranged in quasi-lexicographical order is context-sensitive and this complexity is the best possible we can obtain (in other terms, no language $P(\mathbf{x})$ can be regular when $\mathbf{x}$ is disjunctive).

1.4.6 More About $P =?NP$

We have seen (Section 1.3.2) that the extreme difficulty of the (in)famous problem $P =?NP$ can be explained in terms of oracles [6].

To complete the picture we quote the following two results:

Theorem 1.33

1) [75] *There exist two recursive sets A, B with $P(A) \neq NP(A)$ and $P(B) = NP(B)$, but neither result is provable within Zermelo–Fraenkel set theory augmented with the Axiom of Choice.*

2) [11] *If A is a random oracle, then $P(A) \neq NP(A)$, i.e. with probability one $P(A) \neq NP(A)$.*

Finally, consider the exponential complexity classes

$$E = DTIME\left(2^{\text{linear}}\right), \text{ and } E_2 = DTIME\left(2^{\text{polynomial}}\right).$$

There are several reasons for considering these classes ([106, 107]):

1) Both classes E, E_2 have rich internal structures.

2) E_2 is the smallest deterministic time complexity class known to contain NP and $PSPACE$.

3) $P \subset E \subset E_2, E \neq E_2$, and E contains many NP–complete problems.

4) Both classes E, E_2 have been proven to contain intractable problems.

In view of the property 2) there may be well a natural "notion of smallness" for subsets of E_2 such that P is a small subset of E_2, but NP is not. Similarly, it may be the case that P is a small subset of E, but that $NP \cap E$ is not! In the language of constructive measure theory smallness can be translated by "measure zero" (with respect to the induced spaces E or E_2). One can prove that indeed P has constructive measure zero in E and E_2, [106]. This motivates Lutz [107] to adopt the following quantitative hypothesis:

The set NP has not measure zero.

This is a strong hypothesis, as it implies $P \neq NP$. It is consistent with Zimand's topological analysis [150] (with respect to a natural, constructive topology, if $NP - P$ is non-empty, then it is a *second Baire category* set, while NP–complete sets form a *first category* class) and appears to have more explanatory power than traditional, qualitative hypotheses.

1.5 Parallelism

1.5.1 Parallel Computation Thesis and Alternation.

Achievements in the hardware technologies made possible the construction of parallel computers involving several thousands of processors capable to cooperate in solving one concrete computing task. While the time complexity has been measured as the number of elementary operations needed to compute the output in all sequential computing models this is not more true for the time complexity of parallel machines executing simultaneously lots of operations. The number of processors used by a parallel machine is a new complexity measure (computational resource) considered as a function of the input size. Since sequential time is the amount of work which has to be done we observe that

$$\text{(number of processors)} * \text{(parallel time)} \geq \text{sequential time},$$

for any parallel algorithm solving a problem. If the equality holds we say that the parallel algorithm exhibits an optimal speed up, because it does not need to execute more operations than the best deterministic algorithm. We learn from the non-equality one important fact: We cannot hope that parallelism helps to compute intractable computing problems. If the sequential time of a problem L is exponential, then each parallel algorithms for L has an exponential number of processors or it works in exponential parallel time. Both are unrealistic and we conclude that the main contribution of parallelism is to speed up sequential computations for the problems in P. This is of crucial importance for designing "real–time" parallel algorithms.

As in the sequential case, to study parallel complexity measures one needs a formal computing model. Unfortunately, we have a lot of parallel computing models and none is generally accepted. Moreover, the differences between there models are essential. This is because we have not only to arrange the cooperation between input, output, memory, and the operating unit as for sequential computing models, but we additionally have to arrange the cooperation (information exchange) between many processors working in parallel. The study of distinct models of parallel computations has lead to an invariant characterizing "reasonable" parallel computing models. This invariant is called **Parallel Computation Thesis** [69] and it says that, for any computing problem L,

"the parallel time of L is polynomially related to the sequential space of L".

Thus, each parallel computing model fulfilling the Parallel Computation Thesis is considered to be "suitable" for measuring of parallel computation resources.

In what follows we consider the alternating Turing machine as a parallel computing model and we show that it fulfills the Parallel Computation Thesis. Note that alternating machines cannot be really used to model parallel computations in the practice because they consider nondeterministic processors working in parallel. But they provide several nice new characterizations of sequential complexity classes obtained by typical approaches of formal language theory.

An **alternating TM M** is a natural extension of the nondeterministic TM introduced in [45]. The states of M are divided into 4 disjoint subsets of **accepting** states, **rejecting** states, **existential** states and **universal** states. A computation tree $Tr_M(x)$ of M on an input x is the same tree as by a nondeterministic TM. The difference is only in the definition of the acceptance. A computation of a nondeterministic TM corresponds to a path of the computation tree. A **computation of the alternating TM M on x** is any subtree T of $Tr_M(x)$ having the following properties:

(i) the root of T is the root of $Tr_M(x)$,

(ii) if an inner node v of T is labeled by a universal configuration (state), then T must contain all sons of v in $Tr_M(x)$,

(iii) if an inner node v of T is labeled by an existential configuration, then T contains exactly one of the sons of v in $Tr_M(x)$,

(iv) every leaf of T is labeled either by an accepting configuration or by a rejecting configuration.

An **accepting computation** of M on x is any computation T whose all leaves are labeled by accepting configurations. A **word x is accepted** by the alternating TM M ($x \in L(M)$) iff there exists an accepting computation of M on x.

We observe that existential states of an alternating TM has the same meaning as states of nondeterministic TMs. One has to choose one of the possible actions, and one accepts if at least one of these possibilities leads to acceptance. A step from a universal state (configuration) corresponds to a parallel branching of the machine into a number of copies continuing to work in parallel. Here, one requires that all branches lead to the acceptance.

Let, for any function $f : \mathbb{N} \to \mathbb{N}, ASPACE(f(n))$ and $ATIME(f(n))$ denote the alternating time and space complexity classes respectively. We define

$$ALOG = ASPACE(\log_2 n), AP = \bigcup_{k\in\mathbb{N}} ATIME(n^k),$$

and

$$APSPACE = \bigcup_{k\in\mathbb{N}} ASPACE(n^k).$$

Alternation has brought new characterizations of fundamental complexity classes. Some of the most important ones follow.

Theorem 1.34 [45] *For any space–constructible* $s : \mathbb{N} \to \mathbb{N}$, $s(n) \geq n$,

$$NSPACE(s(n)) \subseteq \bigcup_{c>0} ATIME(c(s(n))^2).$$

Idea of proof. The technique is similar to Savitch's Theorem. To check whether an $s(n)$ space–bounded nondeterministic TM M can achieve an accepting configuration C in $2^{s'(n)}$ steps for a suitable function $s'(n) = O(s(n))$ from an initial configuration C_0 on an input x, the alternating TM A guesses nondeterministically a configuration C' and universally branches into two copies continuing to work in parallel. One copy checks whether C' is reachable from C_0 in $2^{s'(n)-1}$ steps and the other one checks whether C is reachable from C' in $2^{s'(n)-1}$ steps.

□

We do not know whether $SPACE(s(n))$ is equal to $NSPACE(s(n))$ or not. Since all deterministic classes are closed under complementation people have hoped to prove $SPACE(s(n))$ is not equal to $NSPACE(s(n))$ by proving that $NSPACE(s(n))$ is not closed under complementation. But Immerman [94] and Szelepcsényi [138] have proved that this idea cannot work because all classes $NSPACE(s(n))$ for $s(n) \geq \log n$ are closed under complementation too.

Theorem 1.35 [45] *For any function* $t, t(n) \geq n$,

$$ATIME(t(n)) \subseteq DSPACE(t(n)).$$

Idea of proof. Let M be an alternating TM, and let $Tr_M(x)$ is the computation tree of M on x. To check whether $Tr_M(x)$ contains an accepting computation one assigns the value 1 (0) to the accepting (rejecting) leaves, the operator disjunction to the existential nodes, and the operator conjunction to universal nodes. Then one looks at $Tr_M(x)$ as a Boolean circuit with inputs on leaves and the output on the root. Obviously, M accepts x iff the output value of this circuit is 1. A deterministic TM B can traverse $Tr_M(x)$ and calculate the output of the circuit in post order. To do it, at any point of the computation B has to store the visiting configuration (node) and a string representing the position of the node in $Tr_M(x)$. Both can be done in $O(t(n))$ space.

□

Theorem 1.36 [45] *For any function* $s : \mathbb{N} \to \mathbb{N}, s(n) \geq \log_2 n$,

$$ASPACE(s(n)) = \bigcup_{c>0} DTIME(c^{s(n)}).$$

The above results show that alternation shifts by exactly one level the fundamental hierarchy of deterministic complexity classes because $ALOG = P, AP = PSPACE, APSPACE = EXPTIME$, etc. Another interesting result, proven in [97], is that the class of languages accepted by two–way alternating finite automata is exactly $PSPACE$.

One can observe that the copies of an alternating TM working in parallel in some computation do not have any possibility to communicate (exchange information). Since communication is one of the main ingredients of parallel computations, synchronized alternating TM ($SATM$) has been introduced in [85] as a generalization of alternating TMs. We give a brief informal description of this idea. (The formal definitions can be found in [90].) An $SATM$ M is an alternating TM with an additional finite synchronization alphabet. An internal state of M can be either an usual internal state or a pair (internal state, synchronizing symbol). The latter is called a **synchronizing state**. The synchronizing states are used in a computation as follows. Each time one of the machine copies, working in parallel in a computation, enters a synchronizing state it must wait until all other machines working in parallel enter an accepting state or a synchronizing state with the same synchronizing symbol. When this happens all machines are allowed to move from the synchronizing states (to continue to work). In what follows the usual notation $SATIME(f(n))$ and $SASPACE(f(n))$ is used for the synchronized alternating complexity classes. Analogously $SALOG$ and SAP denote synchronized alternating logspace and synchronized alternating polynomial time respectively. The main results proven in [90] are the following:

Theorem 1.37 [90] *For any space–constructible function* $s : \mathbb{N} \to \mathbb{N}$,

$$\bigcup_{c>0} NSPACE(nc^{s(n)}) = SASPACE(s(n)).$$

Corollary 1.38 [90, 133] *For any space–constructible function* $s(n) \geq \log_2 n$

$$SASPACE(s(n)) = \bigcup_{c>0} DSPACE(c^{s(n)})$$

$$= \bigcup_{c>0} ATIME(c^{s(n)}) = \bigcup_{c>0} SATIME(c^{s(n)}).$$

Corollary 1.39 [90, 133] $NSPACE(n)$ *is exactly the class of languages recognized by synchronized alternating finite automata.*

We observe that the equality

$$SASPACE(s(n)) = \bigcup_{c>0} SATIME(c^{s(n)})$$

implies that synchronized alternating machines are able to use the space in an "optimal way". It seems that deterministic, nondeterministic and alternating computing devices do not have this property because if they would have this property then some fundamental complexity hierarchies would collapse. More precisely, if alternating machines would have this property, then $P = NP = PSPACE$. If nondeterministic (deterministic) machines would have this property, then $NLOG = P = NP$ ($DLOG = NLOG = P$).

In [90] a new characterization of $PSPACE$ by $SALOG$ and by synchronized alternating multihead automata is given. This extends the well–known characterization of fundamental complexity classes $DLOG \subseteq NLOG \subseteq P$ by deterministic, nondeterministic, and alternating finite automata respectively. Since [93] shows that NP can be characterized by synchronized alternating multihead automata with polynomial number of synchronizations we get the characterization of the hierarchy $P \subseteq NP \subseteq PSPACE$ by synchronized alternating multihead automata without synchronization, with polynomial synchronization and with full (unbounded) synchronization, respectively.

1.5.2 Limits to Parallel Computation and P–Completeness

In the previous section we have explained that parallelism is used to speed up computations for problems in P, and not to address intractable problems. A very natural question arises: Are the parallel algorithms able to essentially speed up the time complexity of any problem in P? We do not believe that it is possible, but we are not able to prove it from the same reason why we are not able to prove $NP - P \neq \emptyset$. We conjecture that there exist feasible problems which are inherently sequential, i.e. which do not allow any high parallel execution because of hard sequential dependence of the operation order. To give more formal arguments we first define the class of **feasible highly parallel problems** as the class of problems allowing very high degree of parallelization.

Definition 1.40 [122] *For any positive integer i let*

$$NC^i = \{L \mid L \text{ can be accepted by uniform Boolean circuits with size } n^{O(1)} \text{ and depth } O((\log_2 n)^i)\}.$$

$$NC = \{L \mid L \text{ is decidable in parallel time } (\log_2 n)^{O(1)} \text{ using } n^{O(1)} \text{processors}\} = \bigcup_{i=1}^{\infty} NC^i.$$

Now, to support the strong conjecture that $P - NC \neq \emptyset$ we use the same approach as we have used to argue that P should be a proper subset of NP. Let ANC be a class of all parallel algorithms working in parallel time $(\log_2 n)^{O(1)}$ with $n^{O(1)}$ processors.

Definition 1.41 *We say that a language $L_1 \subseteq \Sigma_1^*$ is* **NC–reducible** *to a language $L_2 \subseteq \Sigma_2^*$ if there exists a parallel algorithm $A \in ANC$ which for any input $x \in \Sigma_1^*$ computes an $A(x) \in \Sigma_2^*$ such that*

$$x \in L_1 \Longleftrightarrow A(x) \in L_2.$$

A language L is P–complete under NC–reducibility if

(i) $L \in P$, and

(ii) every language in P is NC–reducible to L.

We note that if an NC–complete language would be in NC, then $NC = P$.

Again, as for NP–completeness, we have many P–complete problems, and for none of them we know a highly parallel solution. So, there is a large experience saying that P–complete problems according to NC–reducibility do not allow feasible highly parallel solutions (i.e. they are inherently sequential). Another reason to believe $NC \neq P$ is that we do not know any fast general simulation of sequential machines by parallel ones. The best known parallel simulations reduce sequential time T to parallel time $T/\log_2 T$ or $\sqrt{T}$, depending on the parallel model. Furthermore, an exponential number of processors is needed to achieve even these modest speed ups. Thus P–completeness is an instrument

helping to study the border between problems having a tractable highly parallel solution and problems which do not have efficient highly parallel solution. More about the classification of computing problems according to the class NC can be found in the excellent book [71] devoted to this topic only.

1.5.3 Communication in Parallel and Distributive Computing

In many parallel and distributive computations the main running time is devoted to the communication between processes. To built parallel architectures whose communication structure has high communication facilities is one of the central tasks of parallel computing. There are many studies in this direction dealing with the efficiency of the realization of basic communication tasks (like broadcast, gossip [91], routing [102]) in distinct communication structures as well as with the ability to effective simulate the communication facilities of several different interconnection networks (communication structures) on one network candidating for our parallel architecture [112]. Results and methods used in this area are primary connected with discrete mathematics and graph theory [102, 112] and so we do not want to give more details here. We omit to discuss typical parallel models of formal language theory like systolic arrays, Lindenmayer systems and other kinds of parallel rewriting too because they are not in any main research streams in the complexity of parallel computing.

We briefly discuss the **parallel communicating grammar systems (PCGSs)** introduced in [119]. Each PCGS can be considered as a directed graph whose nodes are simple regular grammars. If the graph (communication structure) contains a directed edge (G_1, G_2), then the grammar G_2 may ask the grammar G_1 for the submission of the word generated by G_2 (for more details and formal definitions consult [118, 89]). For this model it was shown that some classes of communication structures are absolutely more powerful than other graph classes [105, 88, 89, 118], i.e. that there are languages generated be one communication structure but not by any communication structure from any other graph class. These results are interesting for the theoretical study of communicational aspects in computing because no similar results were achieved for other parallel computing models. Note that it is not realistic to obtain an absolute comparison of interconnection networks from the following reasons. If nodes of networks are standard sequential computers, then already one node can compute anything computable. If every node is a simple processor (finite automaton) and communication structure is a finite graph, then the whole network can be considered as a finite automaton. If we have simple processors and we allow an unbounded growth of the communication structure during the computation, then one can simulate Turing machines with one–dimensional arrays which are the simplest communication structure at all.

Another really important research, partially influenced by the standard formal language methods, is the study of abstract communication complexity of languages. The communication complexity of a language [1, 148, 117] is the necessary and sufficient number of bits exchanged between two computers in order to decide about the acceptance of the input word whose input bits are distributed to the two computers in a balanced way. As program-size complexity, the communication complexity of computing problems has numerous applications for different computing models. It can be applied to get lower bounds on different complexity measures of Boolean circuits, VLSI circuits, interconnection networks and many other not primarily parallel computing models (Turing machines, for instance) [86]. Another important fact is that one can prove exponential gaps between determinismus, nondeterminismus and Monte Carlo probabilismus for communication complexity. Moreover, deterministic communication complexity and Las Vegas one are polynomially related. To prove similar results for time complexity is exactly one of the central problem of theoretical computer science.

Acknowledgement

We are deeply indebted to all authors of referenced works. Indeed, the material of this chapter represents little more than our belated understanding and organization of their original work. Cezar Câmpeanu, Greg Chaitin, Ivana Černá, Jeremy Gibbons, Dana Pardubská, Anna Slobodová, Karl Svozil have read and made valuable suggestions: we express to all our gratitude.

Bibliography

[1] H. Abelson, Lower bounds on information transfer in distributed computations. In: *Proc. 29th Annual IEEE FOCS,* IEEE 1978, 151–158.

[2] S. I. Adian, W. W. Boone, G. Higman (eds.), *Word problems II: The Oxford Book,* North–Holland, New York, 1980.

[3] L. M. Adleman, M. A. Huang, Recognizing primes in random polynomial time, *Technical Report,* Department of Computer Science, Washington, State University, 1988.

[4] D. Z. Albert, On Quantum-Mechanical Automata, *Phys. Lett.* 98A (1983), 249–252.

[5] K. Baker, *Minimalism: Art of Circumstance*, Abbeville Press, New York, 1988.

[6] T. Baker, J. Gill, R. Solovay, Relativizations of the problem $P =?NP$ question, *SIAM J. Comput.* 4(1975), 431–442.

[7] J. L. Balcazar, J. Diaz, J. Gabarro, *Structural Complexity I*, Springer–Verlag, New York, 1988.

[8] J. L. Balcazar, J. Diaz, J. Gabarro, *Structural Complexity II,* Springer–Verlag, New York, 1990.

[9] P. Benioff, Quantum mechanical models of Turing machines, *J. Stat. Phys.* 29 (1982), 515–546.

[10] P. Benioff, Quantum mechanical hamiltonian models of computers, s *Annals New York Academy of Sciences* 480 (1986), 475–486.

[11] C. H. Bennett, J. Gill, Relative to a random oracle A, $P^A \neq NP^A \neq co-NP^A$, with probability one, *SIAM J. Comput.* 10(1981), 96–113.

[12] C. H. Bennett, Certainty from uncertainty, *Nature* 371 (1994), 694–696.

[13] C. H. Bennett, E. Bernstein, G. Brassard, U. V. Vazirani, Strength and weaknesses of quantum computing, *Preprint,* 1995, 9p.

[14] P. Benacerraf, Tasks, super-tasks, and the modern eleatics, *The Journal of Philosophy* 59 (1962), 765–784.

[15] E. Bernstein and U. Vazirani, Quantum complexity theory. In: *Prpc. 25th ACM Symp. on Theory of Computation*, 1993, 11–20.

[16] A. Berthiaume and G. Brassard, The quantum challenge to structural complexity theory. In: *Proc. 7th IEEE Conf. on Structure in Complexity Theory*, 1992, 132–137.

[17] M. Blum, A machine–independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* 14 (1967), 322–336.

[18] R. V. Book, The complexity of languages reducible to algorithmically random languages, *SIAM J. Comput.* 23(1995), 1275–1282.

[19] R. V. Book, J. Lutz, K. Wagner, On complexity classes and algorithmically random languages, *Proc. STACS-92*, Lecture Notes Comput. Sci. 577, 1992, 319–328.

[20] D. P. Bovet, P. Crescenzi, *Introduction to the Theory of Complexity,* Prentice Hall, 1984.

[21] D. S. Bridges, *Computability—A Mathematical Sketchbook*, Springer–Verlag, Berlin, 1994.

[22] D. S. Bridges, C. Calude, On recursive bounds for the exceptional values in speed-up, *Theoret. Comput. Sci.* 132(1994), 387–394.

[23] C. Calude, *Theories of Computational Complexity*, North–Holland, Amsterdam, 1988.

[24] C. Calude, *Information and Randomness—An Algorithmic Perspective*, Springer–Verlag, Berlin, 1994.

[25] C. Calude, D. I. Campbell, K. Svozil, D. Ştefănescu, Strong determinism vs. computability. In: *Proceedings of the International Symposium "The Foundational Debate"*, Vienna Circle Institute Yearbook, 3, 1995, Dordrecht, Kluwer, 115–131.

[26] C. Calude, S. Marcus, Gh. Păun, The universal grammar as a hypothetical brain, *Rev. Roumaine Ling.* 27 (1979), 479–489.

[27] C. Calude, Gh. Păun, Global syntax and semantics for recursively enumerable languages, *Fund. Inform.* 4 (1981), 245–254.

[28] C. Calude, Gh. Păun, On the adequacy of a grammatical model of the brain, *Rev. Roumaine Ling.* 27 (1982), 343–351.

[29] C. Calude, S. Yu, Language-theoretic complexity of disjunctive sequences, *Technical Report No 119, 1995*, Department of Computer Science, The University of Auckland, New Zealand, 8 pp.

[30] C. Calude, M. Zimand, A relation between correctness and randomness in the computation of probabilistic algorithms, *Internat. J. Comput. Math.* 16(1984), 47–53.

[31] C. Calude, M. Zimand, Effective category and measure in abstract complexity theory—extended abstract, *Proceedings FCT'95*, Lectures Notes in Computer Science 965, Springer-Verlag, Berlin, 1995, 156–171.

[32] V. Černý, Quantum computers and intractable (NP-complete) computing problems, *Phys. Rev.* A 48 (1993), 116–119.

[33] G. J. Chaitin, On the length of programs for computing finite binary sequences, *J. Assoc. Comput. Mach.* 13(1966), 547–569.

[34] G. J. Chaitin, Information-theoretic limitations of formal systems, *J. Assoc. Comput. Mach.* 21(1974), 403–424.

[35] G. J. Chaitin, A theory of program size formally identical to information theory, *J. Assoc. Comput. Mach.* 22(1975), 329–340.

[36] G. J. Chaitin, Algorithmic information theory, *IBM J. Res. Develop.* 21(1977), 350–359,496.

[37] G. J. Chaitin, *Algorithmic Information Theory*, Cambridge University Press, Cambridge, 1987. (third printing 1990)

[38] G. J. Chaitin, Computing the Busy Beaver function. In: Cover, T. M. and Gopinath, B. (eds.), *Open Problems in Communication and Computation,* Springer-Verlag, 1987, 108–112.

[39] G. J. Chaitin, *Information, Randomness and Incompleteness, Papers on Algorithmic Information Theory*, World Scientific, Singapore, 1987. (2nd ed., 1990)

[40] G. J. Chaitin, Information-theoretic characterizations of recursive infinite strings, *Theoret. Comput. Sci.* 2(1976), 45–48.

[41] G. J. Chaitin, *Information-Theoretic Incompleteness*, World Scientific, Singapore, 1992.

[42] G. J. Chaitin, On the number of N-bit strings with maximum complexity, *Applied Mathematics and Computation* 59(1993), 97–100.

[43] G. J. Chaitin, A. Arslanov, C. Calude. Program-size complexity computes the halting problem, *Technical Report No 125, 1995*, Department of Computer Science, The University of Auckland, New Zealand, 3 pp.

[44] G. J. Chaitin, J. T. Schwartz, A note on Monte-Carlo primality tests and algorithmic information theory, *Comm. Pure appl. Math.* 31(1978), 521–527.

[45] A. K. Chandra, D. C. Kozen, L. J. Stockmeyer, Alternation, *J. Assoc. Comput. Mach.* 28(1981), 114–133.

[46] A. Cobham, The intrinsic computational difficulty of functions. In: *Proc. Congress for Logic, Mathematics, and Philosophy of Science* 1964, 24–30.

[47] S.A. Cook, The complexity of theorem proving procedure. In: *Proc. 3–rd Annual ACM STOC,* 1971, 151–158.

[48] S.A. Cook, An observation on time–storage trade off. In: *Proc. 5–th Annual ACM STOC,* 1973, 29–33.

[49] S. A. Cook, Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In: *Proc. ACM STOC'79*, 338–345.

[50] M. Davis, Unsolvable problems. In: Barwise, J., ed., *Handbook of Mathematical Logic*, North–Holland, Amsterdam, 1976, 568–594.

[51] P. Davies, *The Mind of God, Science and the Search for Ultimate Meaning*, Penguin Books, London, 1992.

[52] D. Deutsch, Quantum theory, the Church–Turing principle and the universal quantum computer, *Proceedings of the Royal Society of London* A400 (1985), 97–117.

[53] D. Deutsch, Quantum computational networks, *Proceedings of the Royal Society of London* A 425 (1989), 73–90.

[54] D. Deutsch and R. Jozsa, Rapid solution of problems by quantum computation, *Proceedings of the Royal Society of London* A 439 (1992), 553-558

[55] R. L. Devaney, *An Introduction to Chaotic Dynamical Systems*, Addison–Wessley, 1989.

[56] J. Earman and J. D. Norton, Forever is a day: supertasks in Pitowsky and Malament-Hogarth spacetimes, *Philosophy of Science* 60 (1993), 22–42.

[57] U. Eco, *L'Oeuvre ouverte*, Editions du Seuil, Paris, 1965.

[58] G. M. Edelman, *Bright Air, Brilliant Fire—On the Matter of Mind*, Basic Books, 1992.

[59] S. Feferman, J. Dawson, Jr., S. C. Kleene, G. H. Moore, R. M. Solovay, J. van Heijenoort (eds.), *Kurt Gödel Collected Works*, Volume I, Oxford University Press, New York, 1986.

[60] S. Feferman, J. Dawson, Jr., S. C. Kleene, G. H. Moore, R. M. Solovay, J. van Heijenoort (eds.), *Kurt Gödel Collected Works*, Volume II, Oxford University Press, New York, 1990.

[61] R. P. Feynman, Simulating physics with computers, *International Journal of Theoretical Physics* 21 (1982), 467–488.

[62] R. P. Feynman, Quantum Mechanical Computers, *Opt. News* 11 (1985), 11–20.

[63] P. Gács, Every sequence is reducible to a random one, *Inform. and Control* 70(1986), 186-192.

[64] M. Garey, D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[65] M. Gardner, *Logic Machines and Diagrams*, University of Chicago Press, Chicago, 1958, 144. (second printing, Harvester Press, 1983)

[66] M. Gardner, *Fractal Music, Hypercards, and More ...*, W. H. Freeman, New York, 1992, 133.

[67] V. Geffert, Bridging across the $\log(n)$ space frontier. In: *Proc. MFCS'95,* Lect. Notes Comp. Sci. 969, Springer–Verlag, 1995, 50–65.

[68] J. Gill, Computational complexity of probabilistic Turing machines, *SIAM J. Computing* 6 (1977), 675–695.

[69] L.M. Goldschlager, A universal interconnection pattern for parallel computers, *J. Assoc. Comput. Mach.* 29 (1982), 1073–1086.

[70] A. Goswami, R. E. Reed, M. Goswami, *The Self-Aware Universe—How Consciouness Creates the Material World*, G. P. Putnam's Sons, New York, 1993.

[71] R. Greenlaw, H. J. Hoover, W. L. Ruzzo, *Limits to Parallel Computations,* Oxford University Press, Oxford, 1995.

[72] A. Grünbaum, *Modern Science and Zeno's paradoxes,* Allen and Unwin, London, 1968 (Second edition).

[73] A. Grünbaum, *Philosophical Problems of Space of Time,* D. Reidel, Dordrecht, 1973. (Second, enlarged edition)

[74] B. Grümbaum, G. C. Shephard, *Tilings and Patterns*, W. H. Freeman, 1987.

[75] J. Hartmanis, J. E. Hopcroft, Independence results in computer science, *SIGACT News* 8(1976), 13-24.

[76] J. Hartmanis, J. E. Hopcroft, An overview of the theory of computational complexity, *J. Assoc. Comput. Mach.* 18(1971), 444–475.

[77] J. Hartmanis, P. M. Lewis II, R. E. Stearns, Hierarchies of memory limited computations. In: *Proc. 6-th Annual IEEE Symp. on Suitching Circuit Theory and Logical Design,* 1965, 179–190.

[78] J. Hartmanis, R. E. Stearns, On the computational complexity of algorithms, *Trans. Amer. Math. Soc.* 117 (1965), 285–306.

[79] D. Havel, *Algorithmics: The Spirit of Computing,* Addison–Wesley, 1987.

[80] F. C. Hennie, R. E. Stearns, Two–tape simulation of multitape Turing machines, *J. Assoc. Comput. Mach.* 13 (1966), 533–546.

[81] M. Hogarth, Non-Turing computers and non-turing computability, *PSA 1994* 1 (1994), 126-138.

[82] J. E. Hopcroft, J. W. Paul, L. Valiant, On time versus space, *J. Assoc. Comput. Mach.* 24 (1977), 332–337.

[83] J. E. Hopcroft, J. D. Ullman, Relations between time and tape complexities, *J. Assoc. Comput. Mach.* 15 (1968), 414–427.

[84] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, 1979.

[85] J. Hromkovič, How to organize communication among parallel processes in alternating computations, *Unpublished manuscript*, Comenius University, Bratislava, January 1986.

[86] J. Hromkovič, *Communication Complexity and Parallel Computing,* EATCS Monographs, in preparation.

[87] J. Hromkovič, O.H. Ibarra, N.Q. Trân, P, NP and $PSPACE$ characterizations by synchronized alternating finite automata with different communication protocols, *Unpublished manuscript*, 1994.

[88] J. Hromkovič, J. Kari, L. Kari, Some hierarchies for the communication complexity measures of cooperating grammar systems, *Theoretical Computer Science* 127 (1994), 123–147.

[89] J. Hromkovič, J. Kari, L. Kari, D. Pardubská, Two lower bounds on distributive generation of languages. In: *Proc. 19th MFCS'94,* Lect. Notes in Comp. Sci. 841, Springer–Verlag 1994, 423–432.

[90] J. Hromkovič, J. Karhumäki, B. Rovan, and A. Slobodová, On the power of synchronization in parallel computations, *Disc. Appl. Math.* 32 (1991), 156–182.

[91] J. Hromkovič, R. Klasing, B. Monien, R. Peine, Dissemination of information in interconnection networks (broadcasting & gossiping). In: Hsu, F., Du, D.-Z. eds., *Combinatorial Network Theory,* Science Press & AMS 1995, to appear.

[92] J. Hromkovič, B. Rovan, A. Slobodová, Deterministic versus nondeterministic space in terms of synchronized alternating machines, *Theoret. Comp. Sci.* 132 (1994), 319–336.

[93] O.H. Ibarra, N. Q. Trân, On communication–bounded synchronized alternating finite automata, *Acta Informatica* 31 (1994), 315–327.

[94] N. Immerman, Nondeterministic space is closed under complementation, *SIAM J. Comput.* 17 (1988), 935–938.

[95] D. S. Johnson, A catalog of complexity classes. In: van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*, Vol. A, Elsevier, Amsterdam, 1990, 67–161.

[96] H. Jürgensen, G. Thierrin, Some structural properties of ω-languages, *13th Nat. School with Internat. Participation "Applications of Mathematics in Technology"*, Sofia, 1988, 56–63.

[97] K.N. King, Alternating multihead finite automata. In: *Proc. 8-th ICALP'81,* Lect. Notes Comp. Sci. 115, Springer–Verlag, Berlin, 1981, 506–520.

[98] L.G. Khachiyan, A polynomial algorithm in linear programming, *Soviet Mathematics Doklad* 20 (1979), 191–194.

[99] A. N. Kolmogorov, Three approaches for defining the concept of "information quantity", *Problems Inform. Transmission* 1(1965), 3–11.

[100] K.-J. Lange, Complexity and Structure in Formal Language Theory. *Unpublished manuscript,* University of Tübingen, Germany.

[101] P. S. Laplace, *A Philosophical Essay on Probability Theories*, Dover, New York, 1951.

[102] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Array · Trees · Hypercubes,* Morgan Kaufmann Publishers, San Mateo, California 1992.

[103] L. A. Levin, Randomness conservation inequalities: information and independence in mathematical theories, *Problems Inform. Transmission* 10(1974), 206–210.

[104] M. Li, P. M. Vitányi, A new approach to formal language theory by Kolmogorov complexity, *SIAM J. Comput.* 24 (1995), 398–410.

[105] M. Lukáč, *About Two Communication Structures of PCGS,* Master Thesis, Dept. of Computer Science, Comenius University, Bratislava, 1992.

[106] J. H. Lutz, Almost everywhere high nonuniform complexity, *J. Comput. System Sci.* 44(1992), 220–258.

[107] J. H. Lutz, The quantitative structure of exponential time, *Proceedings of the Eighth Annual Structure in Complexity Theory Conference*, (San Diego, CA, May 18–21, 1993), IEEE Computer Society Press, 1993, 158–175.

[108] P. Martin-Löf, The definition of random sequences, *Inform. and Control* 9(1966), 602–619.

[109] Yu. V. Matiyasevich, *Hilbert's Tenth Problem*, MIT Press, Cambridge, Massachusetts, London, 1993.

[110] M. Mendès France, A. Hénaut, Art, therefore entropy, *Leonardo* 27(1994), 219–221.

[111] G. L. Miller, Riemann's hypothesis and tests for primality, *J. Comput. Syst. Sci.* 13 (1976), 300–317.

[112] B. Monien, I. H. Sudborough, Embedding one interconnection network in another, *Computing Suppl.* 7 (1990), 257–282.

[113] C. Moore, Real-Valued, continuous-time computers: a model of analog computation, Part I, *Manuscript*, 1995, 15 pp.

[114] P. Odifreddi, *Classical Recursion Theory*, North–Holland, Amsterdam, Vol.1, 1989.

[115] H. R. Pagels, *The Dreams of Reason*, Bantam Books, New York, 1989.

[116] C. H. Papadimitriou, *Computational Complexity*, Addison–Wesley, New York, 1994.

[117] Ch. Papadimitriou, M. Sipser, Communication complexity, *J. Comp. Syst. Sci.* 28 (1984), 260–269.

[118] D. Pardubská, *On the Power of Communication Structure for Distributive Generation of Languages,* Ph.D. Thesis, Comenius University, Bratislava 1994.

[119] Gh. Păun, L. Sântean, Parallel communicating grammar systems: The regular case, *Ann. Univ. Buc. Ser. Mat.–Inform.* 37 (1989), 55–63.

[120] R. Penrose, *The Emperor's New Mind. Concerning Computers, Minds, and the Laws of Physics*, Oxford University Press, Oxford, 1989.

[121] R. Penrose, *Shadows of the Mind. A Search for the Missing Science of Consciousness*, Oxford University Press, Oxford, 1994.

[122] N. Pipinger, On simultaneous resource bounds (preliminary version). In: *Proc. 20th IEEE Symp. FOCS,* IEEE, New York 1979, 307–311.

[123] I. Pitowsky, The physical Church-Turing thesis and physical complexity theory, *Iyyun, A Jerusalem Philosophical Quarterly* 39 (1990), 81–99.

[124] M.O. Rabin, Probabilistic algorithms. In: Traub, J., ed., *Algorithms and Complexity: New Directions and Recent Results,* Academic Press, New York, 1976, 21–40.

[125] G. Rozenberg, A. Salomaa, *Cornerstones of Undecidability*, Prentice–Hall, 1994.

[126] R. Rucker, *Infinity and the Mind*, Bantam Books, New York, 1983.

[127] R. Rucker, *Mind Tools*, Houghton Mifflin Comp., Boston, 1987.

[128] A. Salomaa, *Computation and Automata*, Cambridge University Press, Cambridge, 1985.

[129] W. J. Savitch, Relationships between nondeterministic and deterministic tape complexities, *J. Comp. Syst. Sciences* 4 (1970), 177–192.

[130] J. R. Searle, *The Rediscovery of the Mind*, MIT Press, Cambridge, Mass., third printing 1992.

[131] P. W. Shor, Algorithms for quantum computation: discrete logarithms and factoring. In: *Proc. 35th Annual IEEE FOCS*, IEEE, 1995. (in press)

[132] H. T. Siegelmann, Computation beyond the Turing limit, *Science* 268, 28 April (1995), 545–548.

[133] A. Slobodová, On the power of communication in alternating machines. In: *Proc. 13th MFCS'88,* Lect. Notes in Comp.Sci. 324, Springer–Verlag 1988, 518–528.

[134] R. M. Smullyan, *Diagonalization and Self–Reference*, Clarendon Press, Oxford, 1994.

[135] R. M. Solovay, *Draft of a paper (or series of papers) on Chaitin's work ... done for the most part during the period of Sept.– Dec. 1974*, unpublished manuscript, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, May 1975, 215 pp.

[136] R. Solovay, V. Strassen, A fast Monte Carlo test for primality, *SIAM J. Computing* 6 (1977), 84–85.

[137] E. Strickland, *Minimalism–Origins*, Indiana University Press, Bloomington, 1993.

[138] R. Szelepcsényi, The method of forced enumeration for nondeterministic automata, *Acta Informatica* 26 (1988), 279–284.

[139] A. Szepietowski, *Turing Machines with Sublogarithmic Space,* Lect. Notes Comp. Sci. 843, Springer–Verlag, Berlin, 1994.

[140] K. Svozil, *Randomness and Undecidability in Physics,* World Scientific, Singapore, 1993.

[141] K. Svozil, On the computational power of physical systems, undecidability, the consistency of phenomena and the practical uses of paradoxes. In: Greenberger, D. M., Zeilinger, A. (eds.) *Fundamental Problems in Quantum Theory: A Conference Held in Honor of Professor John A. Wheeler*, *Annals of the New York Academy of Sciences* 755 (1995), 834–842.

[142] K. Svozil, Quantum computation and complexity theory, I, *EATCS Bull.* 55 (1995), 170–207.

[143] K. Svozil, Quantum computation and complexity theory, II, *EATCS Bull.* 56 (1995), 116–136.

[144] A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc. Ser. 2* 42(1936), 230–265.

[145] J. F. Thomson, Tasks and super-tasks, *Analysis* 15 (1954), 1–13.

[146] H. Weyl, *Philosophy of Mathematics and Natural Science*, Princeton University Press, Princeton, 1949.

[147] I. Xenakis, Musique formelle, *La Revue Musicale* 253/4 (1963), 10.

[148] A.C. Yao, Some complexity questions related to distributed computing. In: *Proc. 11th Annual ACM STOC,* ACM 1981, 308–311.

[149] S. Yu, A pumping lemma for deterministic context-free languages, *Inform. Process. Lett.* 31(1989), 47–51.

[150] M. Zimand, If not empty, $NP - P$ is topologically large, *Theoret. Comput. Sci.* 119(1993), 293–310.

[151] P. van Emde Boas, Machine models and simulations. In: van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*, Vol. A, Elsevier, Amsterdam, 1990, 525–632.

[152] D. J. Velleman, *How to Prove It. A Structural Approach*, Cambridge University Press, Cambridge, 1994.

[153] H. Wang, On 'computabilism' and physicalism: some subproblems. In: Cornwell, J., ed., *Nature's Imagination*, Oxford University Press, Oxford, 1995, 161–189.