Computational Complexity:

A Conceptual Perspective

Oded Goldreich

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel.

October 28, 2006

to Dana

Preface

The strive for efficiency is ancient and universal, as time and other resources are always in shortage. Thus, the question of which tasks can be performed efficiently is central to the human experience.

A key step towards the systematic study of the aforementioned question is a rigorous definition of the notion of a task and of procedures for solving tasks. These definitions were provided by computability theory, which emerged in the 1930's. This theory focuses on computational tasks, and considers automated procedures (i.e., computing devices and algorithms) that may solve such tasks.

In focusing attention on computational tasks and algorithms, computability theory has set the stage for the study of the computational resources (like time) that are required by such algorithms. When this study focuses on the resources that are necessary for *any* algorithm that solves a particular task (or a task of a particular type), the study becomes part of the theory of Computational Complexity (also known as Complexity Theory).[1]

Complexity Theory is a central field of the theoretical foundations of Computer Science. It is concerned with the study of the *intrinsic complexity of computational tasks*. That is, a typical Complexity theoretic study looks at the computational resources required to solve a a computational task (or a class of such tasks), rather than at a specific algorithm or an algorithmic schema. Actually, research in Complexity Theory tends to *start with and focus on the computational resources themselves*, and addresses the effect of limiting these resources on the class of tasks that can be solved. Thus, Computational Complexity is the study of the what can be achieved within limited time (and/or other limited natural computational resources).

The (half-century) history of Complexity Theory has witnessed two main research efforts (or directions). The first direction is aimed towards actually establishing concrete lower bounds on the complexity of problems, via an analysis of the evolution of the process of computation. Thus, in a sense, the heart of this direction is a "low-level" analysis of computation. Most research in circuit com-

[1] In contrast, when the focus is on the design and analysis of specific algorithms (rather than on the intrinsic complexity of the task), the study becomes part of a related subfield that may be called Algorithmic Design and Analysis. Furthermore, Algorithmic Design and Analysis tends to be sub-divided according to the domain of mathematics, science and engineering in which the computational tasks arise. In contrast, Complexity Theory typically maintains a unity of the study of tasks solveable within certain resources (regardless of the origins of these tasks).

plexity and in proof complexity falls within this category. In contrast, a second research effort is aimed at exploring the connections among computational problems and notions, without being able to provide absolute statements regarding the individual problems or notions. This effort may be viewed as a "high-level" study of computation. The theory of NP-completeness as well as the studies of approximation, probabilistic proof systems, pseudorandomness and cryptography all fall within this category.

The current book focuses on the latter effort (or direction). We list several reasons for our decision to focus on the "high-level" direction. The first is the great *conceptual significance* of the known results; that is, many known results (as well as open problems) in this direction have an extremely appealing conceptual message, which can also be appreciated by non-experts. Furthermore, these conceptual aspects may be explained without entering excessive technical detail. Consequently, the "high-level" direction is more suitable for an exposition in a book of the current nature. Finally, there is a subjective reason: the "high-level" direction is within our own expertise, while this cannot be said about the "low-level" direction.

The last paragraph brings us to a discussion of the nature of the current book, which is captured by the subtitle (i.e., "a conceptual perspective"). Our main thesis is that *complexity theory is extremely rich in conceptual content*, and that *this contents should be explicitly communicated in expositions and courses on the subject*. The desire to provide a corresponding textbook is indeed the motivation for writing the current book and its main governing principle.

This book offers a conceptual perspective on complexity theory, and the presentation is designed to highlight this perspective. It is intended mainly for students that wish to learn complexity theory and for educators that intend to teach a course on complexity theory. The book is also intended to promote interest in complexity theory and make it accessible to general readers with adequate background (which is mainly being comfortable with abstract discussions, definitions, and proofs). We expect most readers to have a basic knowledge of algorithms, or at least be fairly comfortable with the notion of an algorithm.

The book focuses on several sub-areas of complexity theory (see the following organization and chapter summaries). In each case, the exposition starts from the intuitive questions addresses by the sub-area, as embodied in the concepts that it studies. The exposition discusses the fundamental *importance* of these questions, the *choices* made in the actual formulation of these questions and notions, the *approaches* that underly the answers, and the *ideas* that are embedded in these answers. Our view is that these ("non-technical") aspects are the core of the field, and the presentation attempts to reflect this view.

We note that being guided by the conceptual contents of the material leads, in some cases, to technical simplifications. Indeed, for many of the results presented in this book, the presentation of the proof is different (and arguably easier to understand) than the standard presentations.

Organization and Chapter Summaries

This book consists of ten chapters and seven appendices. The chapters constitute the core of this book and are written in a style adequate for a textbook, whereas the appendices provide additional perspective and are written in the style of a survey article. The relative length and ordering of the chapters (and appendices) does not reflect their relative importance, but rather an attempt at the best logical order (i.e., minimizing the number of forward pointers).

Following are brief summaries of the book's chapters and appendices. Theses summaries are more detailed than those provided in Section 1.1.3 but less detailed than the summaries provided at the beginning of each chapter.

Chapter 1: Introduction and Preliminaries. The introduction provides a high-level overview of some of the content of complexity theory as well as a discussion of some of the characteristic features of this field. The preliminaries provide the relevant background on *computability theory*, which is the setting in which complexity theoretic questions are being studied. Most importantly, central notions such as search and decision problems, algorithms that solve such problems, and their complexity are defined. In addition, this part presents the basic notions underlying non-uniform models of computation (like Boolean circuits).

Chapter 2: P, NP and NP-completeness. The P-vs-NP Question can be phrased as asking whether or not finding solutions is harder than checking the correctness of solutions. An alternative formulation in terms of decision problems asks whether or not discovering proofs is harder than verifying their correctness; that is, is proving harder than verifying. It is widely believed that the answer to the two equivalent formulation is that finding (resp., proving) is harder than checking (resp., verifying); that is, that P is different from NP. At present, when faced with a hard problem in NP, we can only hope to prove that it is not in P assuming that NP is different from P. This is where the theory of NP-completeness, which is based on the notion of a reduction, comes into the picture. In general, one computational problem is reducible to another problem if it is possible to efficiently solve the former when provided with an (efficient) algorithm for solving the latter. A problem (in NP) is NP-complete if any problem in NP is reducible to it. Amazingly enough, NP-complete problems exist, and furthermore hundreds of natural computational problems arising in many different areas of mathematics and science are NP-complete.

Chapter 3: Variations on P and NP. Non-uniform polynomial-time (P/poly) captures efficient computations that are carried out by devices that handle specific input lengths. The basic formalism ignores the complexity of constructing such devices (i.e., a uniformity condition), but a finer formalism (based on "machines that take advice") allows to quantify the amount of non-uniformity. The Polynomial-time Hierarchy (PH) generalizes NP by considering statements expressed by a quantified Boolean formula with a fixed number of alternations of existential and universal quantifiers. It is widely believed that each quantifier alternation adds expressive power to the class of such formulae. The two different classes are related by showing that if NP is contained in P/poly then the Polynomial-time Hierarchy collapses to its second level.

Chapter 4: More Resources, More Power? When using "nice" functions to determine the algorithm's resources, it is indeed the case that more resources allow for more tasks to be performed. However, when "ugly" functions are used for the same purpose, increasing the resources may have no effect. By nice functions we mean functions that can be computed without exceeding the amount of resources that they specify. Thus, we get results asserting, for example, that there are problems that are solvable in cubic-time but not in quadratic-time. In the case of non-uniform models of computation, the issue of "nicety" does not arise, and it is easy to establish separations results.

Chapter 5: Space Complexity. This chapter is devoted to the study of the space complexity of computations, while focusing on two rather extreme cases. The first case is that of algorithms having logarithmic space complexity, which seem a proper and natural subset of the set of polynomial-time algorithms. The second case is that of algorithms having polynomial space complexity, which in turn can solve almost all computational problems considered in this book. Among the results presented in this chapter are a log-space algorithm for exploring (undirected) graphs, a non-deterministic log-space procedure for recognizing directed graphs that are *not* strongly connected, and complete problems for $\mathcal{NL}$ and $\mathcal{PSPACE}$ (under log-space and polynomial-time reductions, respectively).

Chapter 6: Randomness and Counting. Various failure types of probabilistic polynomial-time algorithms give rise to complexity classes such as $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$. The results presented include the emulation of probabilistic choices by non-uniform advice (i.e., $\mathcal{BPP} \subset \mathcal{P}/\mathrm{poly}$) and the emulation of two-sided probabilistic error by an $\exists\forall$-sequence of quantifiers (i.e., $\mathcal{BPP} \subseteq \Sigma_2$). Turning to counting problems (i.e., counting the number of solutions for NP-type problems), we distinguish between exact counting and approximate counting (in the sense of relative approximation). While any problem in $\mathcal{PH}$ is reducible to the exact counting class $\#\mathcal{P}$, approximate counting (for $\#\mathcal{P}$) is (probabilisticly) reducible to $\mathcal{NP}$. Additional related topics include $\#\mathcal{P}$-completeness, the complexity of searching for unique solutions, and the relation between approximate counting and generating almost uniformly distributed solutions.

Chapter 7: The Bright Side of Hardness. It turns out that hard problem can be "put to work" to our benefit, most notably in cryptography. One key issue that arises in this context is bridging the gap between "occasional" hardness (e.g., worst-case hardness or mild average-case hardness) and "typical" hardness (i.e., strong average-case hardness). We consider two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems that are solvable in exponential-time but are not solvable by (non-uniform) families of small (say polynomial-size) circuits. We show that these types of worst-case conjectures can be transformed into average-case hardness results that yield non-trivial derandomizations of $\mathcal{BPP}$ (and even $\mathcal{BPP} = \mathcal{P}$). The second conjecture is that there are problems in NP for which it is easy to generate (solved) instances that are hard to solve for other people. This conjecture is captured in the notion of *one-way functions*, which are functions that are easy to evaluate but hard to invert (in an average-case sense). We show that functions that are hard to invert in a relatively mild average-case sense yield functions that are hard to invert almost everywhere, and that the latter yield predicates that are very hard to approximate (called *hard-core predicates*). The latter are useful for the construction of general-purpose pseudorandom generators as well as for a host of cryptographic applications.

Chapter 8: Pseudorandom Generators. A fresh view at the *question of randomness* was taken in the theory of computing: It has been postulated that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by any efficient procedure. The paradigm, originally associating efficient procedures with polynomial-time algorithms, has been applied also with respect to a variety of limited classes of such distinguishing procedures. The archetypical case of pseudorandom generators refers to efficient generators that fool any feasible procedure; that is, the potential distinguisher is any probabilistic polynomial-time algorithm, which may be more complex than the generator itself. These generators are called general-purpose, because their output can be safely used in any efficient application. In contrast, for purposes of derandomization, one may use pseudorandom generators that are somewhat more complex than the potential distinguisher (which represents the algorithm to be derandomized). Following this approach and using various hardness assumptions, one may obtain corresponding derandomizations of $\mathcal{BPP}$ (including a full derandomization; i.e., $\mathcal{BPP} = \mathcal{P}$). Other forms of pseudorandom generators include ones that fool space-bounded distinguishers, and even weaker ones that only exhibit some limited random behavior (e.g., outputting a pair-wise independent sequence).

Chapter 9: Probabilistic Proof Systems. Randomized and interactive verification procedures, giving rise to *interactive proof systems*, seem much more powerful than their deterministic counterparts. In particular, interactive proof systems exist for any set in $\mathcal{PSPACE} \supseteq \text{co}\mathcal{NP}$ (e.g., for the set of unsatisfied propositional formulae), whereas it is widely believed that some sets in $\text{co}\mathcal{NP}$ do *not* have NP-proof systems. Interactive proofs allow the meaningful conceptualization of *zero-knowledge proofs*, which are interactive proofs that yield nothing (to the verifier) beyond the fact that the assertion is indeed valid. Under reasonable complexity assumptions, every set in $\mathcal{NP}$ has a zero-knowledge proof system. (This result has many applications in cryptography.) A third type of probabilistic proof systems is the model of PCPs, standing for *probabilistically checkable proofs*. These are (redundant) NP-proofs that offers a trade-off between the number of locations (randomly) examined in the proof and the confidence in its validity. In particular, a small constant error probability can be obtained by reading a constant number of bits in the redundant NP-proof. The PCP Theorem asserts that NP-proofs can be efficiently transformed into PCPs. The study of PCPs is closely related to the study of the complexity of approximation problems.

Chapter 10: Relaxing the Requirement. In light of the apparent infeasibility of solving numerous useful computational problems, it is natural to seek relaxations of these problems that remain useful for the original applications and yet allow for feasible solving procedures. Two such types of relaxations are provided by adequate notions of approximation and a theory of average-case complexity. The notions of approximation refer to the computational problems themselves; that is, for each problem instance we extend the set of admissible solutions. In the context of search problems this means settling for solutions that have a value that is "sufficiently close" to the value of the optimal solution, whereas in the context of decision problems this means settling for procedures that distinguish yes-instances from instances that are "far" from any yes-instance. Turning to average-case complexity, we note that a systematic study of this notion requires the development of a non-trivial conceptual framework. A major aspect of this framework is limiting the class of distributions in a way that, on one hand, allows for various types of natural distributions and, on the other hand, prevents the collapse of average-case complexity to worst-case complexity.

Appendix A: Glossary of Complexity Classes. The glossary provides self-contained definitions of most complexity classes mentioned in the book. The glossary is partitioned into two parts, dealing separately with complexity classes that are defined in terms of algorithms and their resources (i.e., time and space complexity of Turing machines) and complexity classes defined in terms of non-uniform circuit (and referring to their size and depth). The following classes are defined: $\mathcal{P}$, $\mathcal{NP}$, $\text{co}\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, $\text{co}\mathcal{RP}$, $\mathcal{ZPP}$, $\#\mathcal{P}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$, $\mathcal{NEXP}$, $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$, $\mathcal{PSPACE}$, $\mathcal{P}/\text{poly}$, $\mathcal{NC}^k$, and $\mathcal{AC}^k$.

Appendix B: On the Quest for Lower Bounds. This appendix surveys some attempts at proving lower bounds on the complexity of natural computational problems. The first part, devoted to Circuit Complexity, reviews lower bounds for the *size* of (restricted) circuits that solve natural computational problems. This represents a program whose long-term goal is proving that $\mathcal{P} \neq \mathcal{NP}$. The second part, devoted to Proof Complexity, reviews lower bounds on the length of (restricted) propositional proofs of natural tautologies. This represents a program whose long-term goal is proving that $\mathcal{NP} \neq \text{co}\mathcal{NP}$.

Appendix C: On the Foundations of Modern Cryptography. This appendix surveys the foundations of cryptography, which are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural security concerns. It presents some of these conceptual tools as well as some of the fundamental results obtained using them. The appendix augments the partial treatment of one-way functions, pseudorandom generators, and zero-knowledge proofs (which is included in Chapters 7–9). Using these basic tools, the appendix provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

Appendix D: Probabilistic Preliminaries and Advanced Topics in Randomization. The probabilistic preliminaries include conventions regarding random variables and overviews of three useful inequalities (i.e., Markov Inequality, Chebyshev's Inequality, and Chernoff Bound). The advanced topics include constructions and lemmas regarding families of hashing functions, a study of the sample and randomness complexities of estimating the average value of an arbitrary function, and the problem of randomness extraction (i.e., procedures for extracting almost perfect randomness from sources of weak or defected randomness).

Appendix E: Explicit Constructions. Complexity theory provides a clear perspective on the intuitive notion of an explicit construction. This perspective is demonstrated with respect to error correcting codes and expander graphs. On the topic of codes, the appendix focuses on various computational aspects, containing a review of several popular constructions as well as a construction of a binary code of constant rate and constant relative distance. Also included are a brief review of the notions of locally testable and locally decodable codes, and a useful upper-bound on the number of codewords that are close to any single word. Turning to expander graphs, the appendix contains a review of two standard definitions of expanders, two levels of explicitness, two properties of expanders that are related to (single-step and multi-step) random walks on them, and two explicit constructions of expander graphs.

Appendix F: Some Omitted Proofs. This appendix contains some proofs that were not included in the main text (for a variety of reasons) and still are beneficial as alternatives to the original and/or standard presentations. Included are proofs that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via randomized Karp-reductions, and that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$, for any function f such that $f(n) \in \{2, ..., \mathrm{poly}(n)\}$.

Appendix G: Some Computational Problems. This appendix includes definitions of most of the specific computational problems that are referred to in the main text. In particular, it contains a brief introduction to graph algorithms, boolean formulae and finite fields.

Acknowledgments

My perspective on complexity theory was most influenced by Shimon Even and Leonid Levin. In fact, it was hard not to be influenced by these two remarkable and highly opinionated researchers (especially for somebody like me who was fortunate to spend a lot of time with them).[2]

Shimon Even viewed complexity theory as the study of the limitations of algorithms, a study concerned with natural computational resources and natural computational tasks. Complexity theory was there to guide the engineer and to address the deepest questions that bother an intellectually curious computer scientist. I believe that this book shares Shimon's view of complexity theory as evolving around such questions.

Leonid Levin emphasized the general principles that underly complexity theory, rejecting any "model-dependent effects" as well as the common coupling of complexity theory with the theory of automata and formal languages. In my opinion, this book is greatly influenced by these opinions of Levin.

I wish to acknowledge the influence of numerous other colleagues on my professional perspectives and attitudes. These include Shafi Goldwasser, Dick Karp, Silvio Micali, and Avi Wigderson. I also wish to thank many colleagues for their comments and advice regarding earlier versions of this text. A partial list includes Noam Livne, Omer Reingold, Dana Ron, Ronen Shaltiel, Amir Shpilka, Madhu Sudan, Salil Vadhan, and Avi Wigderson.

Lastly, I am grateful to Mohammad Mahmoody Ghidary and Or Meir for their careful reading of drafts of this manuscript and for the numerous corrections and suggestions they have provided.

Relation to previous texts of mine. Some of the text of this book has been adapted from previous texts of mine. In particular, Chapters 8 and 9 were written based on my surveys [87, Chap. 3] and [87, Chap. 2], respectively; but the exposition has been extensively revised to fit the significantly different aims of the current book. Similarly, Section 7.1 and Appendix C were written based on my survey [87, Chap. 1] and books [88, 89]; but, again, the previous texts are very different in many ways. In contrast, Appendix B was adapted with relatively little modifications from a section of an article by Avi Wigderson and myself [104].

[2] Shimon Even was my graduate studies adviser (at the Technion, 1980-83); whereas I had a lot of meetings with Leonid Levin during my post-doctoral period (at MIT, 1983-86).

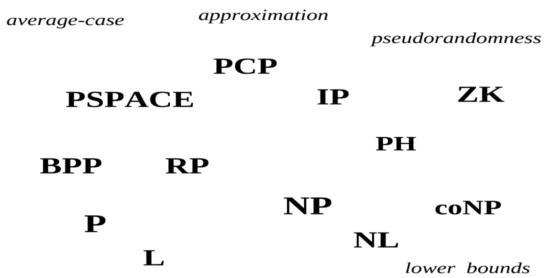
Chapter 1

Introduction and Preliminaries

You can start by putting the DO NOT DISTURB *sign.*

Cay, in *Desert Hearts* (1985).

The current chapter consists of two parts. The first part provides a high-level introduction to (computational) complexity theory. This introduction is much more detailed than the laconic statements made in the preface, but is quite sparse when compared to the richness of the field. In addition, the introduction contains several important comments regarding the contents, approach and style of the current book.



The second part of this chapter provides the necessary preliminaries to the rest of the book. It includes a discussion of computational tasks and computational models, as well as natural complexity measures associated with the latter. More specifically, this part recalls the basic notions and results of computability theory (including the definition of Turing machines, some undecidability results, the notion of universal machines, and the definition of oracle machines). In addition, this part presents the basic notions underlying non-uniform models of computation (like Boolean circuits).

1.1 Introduction

This section consists of two parts: the first part refers to the area itself, whereas the second part refers to the current book. The first part provides a brief overview of Complexity Theory (Section 1.1.1) as well as some reflections about its characteristics (Section 1.1.2). The second part describes the contents of this book (Section 1.1.3), the considerations underlying the choice of topics as well as the way they are presented (Section 1.1.4), and various notations and conventions (Section 1.1.5).

1.1.1 A brief overview of Complexity Theory

Out of the tough came forth sweetness[1]

Judges, 14:14

Complexity Theory is concerned with the study of the *intrinsic complexity* of computational tasks. Its "final" goals include the determination of the complexity of any well-defined task. Additional goals include obtaining an understanding of the relations between various computational phenomena (e.g., relating one fact regarding computational complexity to another). Indeed, we may say that the former type of goals is concerned with *absolute* answers regarding specific computational phenomena, whereas the latter type is concerned with questions regarding the *relation* between computational phenomena.

Interestingly, so far Complexity Theory has been more successful in coping with goals of the latter ("relative") type. In fact, the failure to resolve questions of the "absolute" type, led to the flourishing of methods for coping with questions of the "relative" type. Musing for a moment, let us say that, in general, the difficulty of obtaining absolute answers may naturally lead to seeking conditional answers, which may in turn reveal interesting relations between phenomena. Furthermore, the lack of absolute understanding of individual phenomena seems to facilitate the development of methods for relating different phenomena. Anyhow, this is what happened in Complexity Theory.

Putting aside for a moment the frustration caused by the failure of obtaining absolute answers, we must admit that there is something fascinating in the success to relate different phenomena: in some sense, relations between phenomena are more revealing than absolute statements about individual phenomena. Indeed, the first example that comes to mind is the theory of NP-completeness. Let us consider this theory, for a moment, from the perspective of these two types of goals.

Complexity theory has failed to determine the intrinsic complexity of tasks such as finding a satisfying assignment to a given (satisfiable) propositional formula or finding a 3-coloring of a given (3-colorable) graph. But it has established that these two seemingly different computational tasks are in some sense the same (or, more precisely, are computationally equivalent). We find this success amazing

[1]The quote is commonly used to mean that benefit arose out of misfortune.

and exciting, and hopes that the reader shares these feelings. The same feeling of wonder and excitement is generated by many of the other discoveries of Complexity theory. Indeed, the reader is invited to join a fast tour of some of the other questions and answers that make up the field of Complexity theory.

We will indeed start with the "P versus NP Question". Our daily experience is that it is harder to solve a problem than it is to check the correctness of a solution (e.g., think of either a puzzle or a research problem). Is this experience merely a coincidence or does it represent a fundamental fact of life (or a property of the world)? Could you imagine a world in which solving any problem is not significantly harder than checking a solution to it? Would the term "solving a problem" not lose its meaning in such a hypothetical (and impossible in our opinion) world? The denial of the plausibility of such a hypothetical world (in which "solving" is not harder than "checking") is what "P different from NP" actually means, where P represents tasks that are efficiently solvable and NP represents tasks for which solutions can be efficiently checked.

The mathematically (or theoretically) inclined reader may also consider the task of proving theorems versus the task of verifying the validity of proofs. Indeed, finding proofs is a special type of the aforementioned task of "solving a problem" (and verifying the validity of proofs is a corresponding case of checking correctness). Again, "P different from NP" means that there are theorems that are harder to prove than to be convinced of their correctness when presented with a proof. This means that the notion of a proof is meaningful (i.e., that proofs do help when trying to be convinced of the correctness of assertions). Here NP represents sets of assertions that can be efficiently verified with the help of adequate proofs, and P represents sets of assertions that can be efficiently verified from scratch (i.e., without proofs).

In light of the foregoing discussion it is clear that the P-versus-NP Question is a fundamental scientific question of far-reaching consequences. The fact that this question seems beyond our current reach led to the development of the theory of NP-completeness. Loosely speaking, this theory identifies a set of computational problems that are as hard as NP. That is, the fate of the P-versus-NP Question lies with each of these problems: if any of these problems is easy to solve then so are all problems in NP. Thus, showing that a problem is NP-complete provides evidence to its intractability (assuming, of course, "P different than NP"). Indeed, demonstrating NP-completeness of computational tasks is a central tool in indicating hardness of natural computational problems, and it has been used extensively both in computer science and in other disciplines. NP-completeness indicates not only the conjectured intractability of a problem but rather also its "richness" in the sense that the problem is rich enough to "encode" any other problem in NP. The use of the term "encoding" is justified by the exact meaning of NP-completeness, which in turn is based on establishing relations between different computational problems (without referring to their "absolute" complexity).

The foregoing discussion of the P-versus-NP Question also hints to *the importance of representation*, a phenomenon that is central to complexity theory. In general, complexity theory is concerned with problems the solutions of which are implicit in the problem's statement (or rather in the instance). That is, the problem (or rather its instance) contains all necessary information, and one merely needs to process this information in order to supply the answer.[2] Thus, complexity theory is concerned with manipulation of information, and its transformation from one representation (in which the information is given) to another representation (which is the one desired). Indeed, a solution to a computational problem is merely a different representation of the information given; that is, a representation in which the answer is explicit rather than implicit. For example, the answer to the question of whether or not a given Boolean formula is satisfiable is implicit in the formula itself (but the task is to make the answer explicit). Thus, complexity theory clarifies a central issue regarding representation; that is, the distinction between what is explicit and what is implicit in a representation. Furthermore, it even suggests a quantification of the level of non-explicitness.

In general, complexity theory provides new viewpoints on various phenomena that were considered also by past thinkers. Examples include the aforementioned concepts of proofs and representation as well as concepts like randomness, knowledge, interaction, secrecy and learning. We next discuss some of these concepts and the perspective offered by complexity theory.

The concept of *randomness* has puzzled thinkers for ages. Their perspective can be described as ontological: they asked "what is randomness" and wondered whether it exist at all (or is the world deterministic). The perspective of complexity theory is behavioristic: it is based on defining objects as equivalent if they cannot be told apart by any efficient procedure. That is, a coin toss is (defined to be) "random" (even if one believes that the universe is deterministic) if it is infeasible to predict the coin's outcome. Likewise, a string (or a distribution of strings) is "random" if it is infeasible to distinguish it from the uniform distribution (regardless of whether or not one can generate the latter). Interestingly, randomness (or rather pseudorandomness) defined this way is efficiently expandable; that is, under a reasonable complexity assumption (to be discussed next), short pseudorandom strings can be deterministically expanded into long pseudorandom strings. Indeed, it turns out that randomness is intimately related to intractability. Firstly, note that the very definition of pseudorandomness refers to intractability (i.e., the infeasibility of distinguishing a pseudorandomness object from a uniformly distributed object). Secondly, as stated, a complexity assumption, which refers to the existence of functions that are easy to evaluate but hard to invert (called *one-way functions*), implies the existence of deterministic programs (called *pseudorandom generators*) that stretch short random seeds into long pseudorandom sequences. In fact, it turns out that the existence of pseudorandom generators is equivalent to the existence of one-way functions.

Complexity theory offers its own perspective on the concept of *knowledge* (and distinguishes it from information). Specifically, complexity theory views knowledge as the result of a hard computation. Thus, whatever can be efficiently done by any-

[2]In contrast, in other disciplines, solving a problem may require gathering information that is not available in the problem's statement. This information may either be available from auxiliary (past) records or be obtained by conducting new experiments.

one is not considered knowledge. In particular, the result of an easy computation applied to publicly available information is not considered knowledge. In contrast, the value of a hard to compute function applied to publicly available information is knowledge, and if somebody provides you with such a value then it has provided you with knowledge. This discussion is related to the notion of *zero-knowledge* interactions, which are interactions in which no knowledge is gained. Such interactions may still be useful, because they may convince a party of the *correctness* of specific data that was provided beforehand.

The foregoing paragraph has explicitly referred to *interaction*. It has pointed one possible motivation for interaction: gaining knowledge. It turns out that interaction may help in a variety of other contexts. For example, it may be easier to verify an assertion when allowed to interact with a prover rather than when reading a proof. Put differently, interaction with a good teacher may be more beneficial than reading any book. We comment that the added power of such *interactive proofs* is rooted in their being randomized (i.e., the verification procedure is randomized), because if the verifier's questions can be determined beforehand then the prover may just provide the transcript of the interaction as a traditional written proof.

Another concept related to knowledge is that of *secrecy*: knowledge is something that one party has while another party does not have (and cannot feasibly obtain by itself) – thus, in some sense knowledge is a secret. In general, complexity theory is related to *Cryptography*, where the latter is broadly defined as the study of systems that are easy to use but hard to abuse. Typically, such systems involve secrets, randomness and interaction as well as a complexity gap between the ease of proper usage and the infeasibility of causing the system to deviate from its prescribed behavior. Thus, much of Cryptography is based on complexity theoretic assumptions and its results are typically transformations of relatively simple computational primitives (e.g., one-way functions) into more complex cryptographic applications (e.g., secure encryption schemes).

We have already mentioned the concept of *learning* when referring to learning from a teacher versus learning from a book. Recall that complexity theory provides evidence to the advantage of the former. This is in the context of gaining knowledge about publicly available information. In contrast, computational learning theory is concerned with learning objects that are only partially available to the learner (i.e., learning a function based on its value at a few random locations or even at locations chosen by the learner). Complexity theory sheds light on the intrinsic limitations of learning (in this sense).

Complexity theory deals with a variety of computational tasks. We have already mentioned two fundamental types of tasks: *searching for solutions* (or rather "finding solutions") and *making decisions* (e.g., regarding the validity of assertion). We have also hinted that in some cases these two types of tasks can be related. Now we consider two additional types of tasks: *counting the number of solutions* and *generating random solutions*. Clearly, both the latter tasks are at least as hard as finding arbitrary solutions to the corresponding problem, but it turns out that for some natural problems they are not significantly harder. Specifically, under some natural conditions on the problem, approximately counting the number of solutions and generating an approximately random solution is not significantly harder than finding an arbitrary solution.

Having mentioned the notion of *approximation*, we note that the study of the complexity of finding approximate solutions has also received a lot of attention. One type of approximation problems refers to an objective function defined on the set of potential solutions. Rather than finding a solution that attains the optimal value, the approximation task consists of finding a solution that attains an "almost optimal" value, where the notion of "almost optimal" may be understood in different ways giving rise to different levels of approximation. Interestingly, in many cases, even a very relaxed level of approximation is as difficult to obtain as solving the original (exact) search problem (i.e., finding an approximate solution is as hard as finding an optimal solution). Surprisingly, these hardness of approximation results are related to the study of *probabilistically checkable proofs*, which are proofs that allow for ultra-fast probabilistic verification. Amazingly, every proof can be efficiently transformed into one that allows for probabilistic verification based on probing a *constant* number of bits (in the alleged proof). Turning back to approximation problems, we note that in other cases a reasonable level of approximation is easier to achieve than solving the original (exact) search problem.

Approximation is a natural relaxation of various computational problems. Another natural relaxation is the study of *average-case complexity*, where the "average" is taken over some "simple" distributions (representing a model of the problem's instances that may occur in practice). We stress that, although it was not stated explicitly, the entire discussion so far has referred to "worst-case" analysis of algorithms. We mention that worst-case complexity is a more robust notion than average-case complexity. For starters, one avoids the controversial question of what are the instances that are "important in practice" and correspondingly the selection of the class of distributions for which average-case analysis is to be conducted. Nevertheless, a relatively robust theory of average-case complexity has been suggested, albeit it is less developed than the theory of worst-case complexity.

In view of the central role of randomness in complexity theory (as evident, say, in the study of pseudorandomness, probabilistic proof systems, and cryptography), one may wonder as to whether the randomness needed for the various applications can be obtained in real-life. One specific question, which received a lot of attention, is the possibility of "purifying" randomness (or "extracting good randomness from bad sources"). That is, can we use "defected" sources of randomness in order to implement almost perfect sources of randomness. The answer depends, of course, on the model of such defected sources. This study turned out to be related to complexity theory, where the most tight connection is between some type of *randomness extractors* and some type of pseudorandom generators.

So far we have focused on the time complexity of computational tasks, while relying on the natural association of efficiency with time. However, time is not the only resource one should care about. Another important resource is *space*: the amount of (temporary) memory consumed by the computation. The study of space complexity has uncovered several fascinating phenomena, which seem to indicate a fundamental difference between space complexity and time complexity. For example, in the context of space complexity, verifying proofs of validity of assertions (of any specific type) has the same complexity as verifying proofs of invalidity for the same type of assertions.

In case the reader feels dizzy, it is no wonder. We took an ultra-fast air-tour of some mountain tops, and dizziness is to be expected. Needless to say, the rest of the book offers a totally different touring experience. We will climb some of these mountains by foot, step by step, and will often stop to look around and reflect.

Absolute Results (a.k.a. Lower-Bounds). As stated up-front, absolute results are not known for many of the "big questions" of complexity theory (most notably the P-versus-NP Question). However, several highly non-trivial absolute results have been proved. For example, it was shown that using negation can speed-up the computation of monotone functions (which do not require negation for their mere computation). In addition, many promising techniques were introduced and employed with the aim of providing a low-level analysis of the progress of computation. However, as stated in the preface, the focus of this book is elsewhere.

1.1.2 Characteristics of Complexity Theory

We are successful because we use the right level of abstraction

Avi Wigderson (1996)

Using the "right level of abstraction" seems to be a main characteristic of the Theory of Computation at large. The right level of abstraction means abstracting away second-order details, which tend to be context-dependent, while using definitions that reflect the main issues (rather than abstracting them away too). Indeed, using the right level of abstraction calls for an extensive exercising of good judgment, and one indication for having chosen the right abstractions is the result of their study.

One major choice of the theory of computation, which is currently taken for granted, is the *choice of a model of computation and corresponding complexity measures and classes*. Two extreme choices that were avoided are a too realistic model and a too abstract model. On the one hand, the main model of computation used in complexity theory does not try to reflect (or mirror) the specific operation of real-life computers used at a specific historical time. Such a choice would have made it very hard to develop complexity theory as we know it and to uncover the fundamental relations discussed in this book: the mass of details would have obscured the view. On the other hand, avoiding any reference to any concrete model (like in the case of recursive function theory) does not encourage the introduction and study of natural measures of complexity. Indeed, as we shall see in Section 1.2.3, the choice was (and is) to use a simple model of computation (which does not mirror real-life computers), while avoiding any effects that are specific to that model (by keeping a eye on a host of variants and alternative models). The freedom from the specifics of the basic model is obtained by considering complexity classes that are invariant under a change of model (as long as the alternative model is "reasonable").

Another major choice is the use of *asymptotic analysis*. Specifically, we consider the complexity of an algorithm as a function of its input length, and study the asymptotic behavior of this function. It turns out that structure that is hidden by concrete quantities appears at the limit. Furthermore, depending on the case, we classify functions according to different criteria. For example, in case of time complexity we consider classes of functions that are closed under multiplication, whereas in case of space complexity we consider closure under addition. In each case, the choice is governed by the nature of the complexity measure being considered. Indeed, one could have developed a theory without using these conventions, but this would have resulted in a far more cumbersome theory. For example, rather than saying that finding a satisfying assignment for a given formula is polynomial-time reducible to deciding the satisfiability of some other formulae, one could have stated the exact functional dependence of the complexity of the search problem on the complexity of the decision problem.

Both the aforementioned choices are common to other branches of the theory of computation. One aspect that makes complexity theory unique is its perspective on the most basic question of the theory of computation; that is, the way it studies the question of *what can be efficiently computed*. The perspective of complexity theory is general in nature. This is reflected in its primary focus on the relevant *notion of efficiency* (captured by corresponding resource bounds) rather than on specific computational problems. In most cases, complexity theoretic studies do not refer to any specific computational problems or refer to such problems merely as an illustration. Furthermore, even when specific computational problems are studied, this study is (explicitly or at least implicitly) aimed at understanding the computational limitations of certain resource bounds.

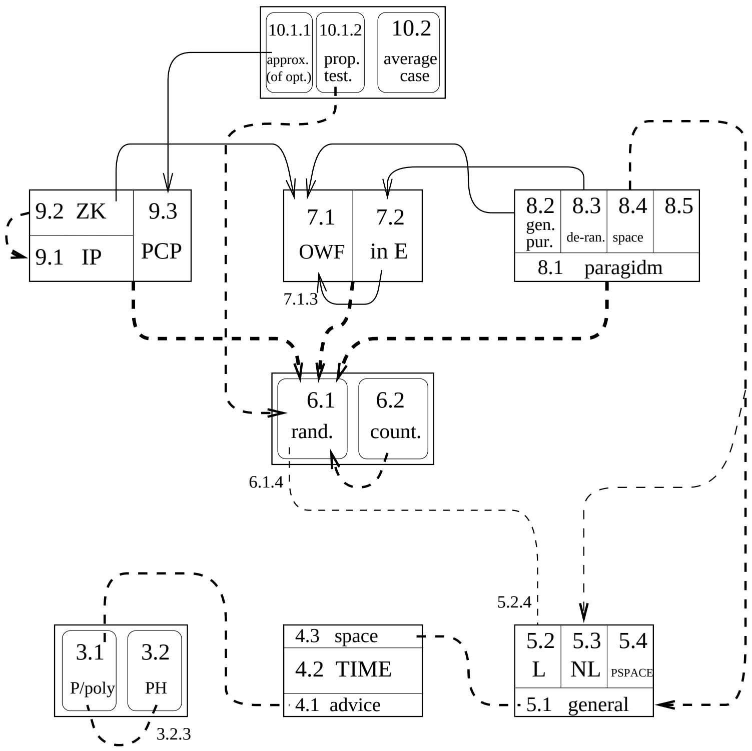
The aforementioned general perspective seems linked to the significant role of conceptual considerations in the field: The rigorous study of an intuitive notion of efficiency must be initiated with an adequate choice of definitions. Since this study refers to any possible (relevant) computation, the definitions cannot be derived by abstracting some concrete reality. Indeed, the definitions attempt to capture any possible reality, which means that the choice of definitions is governed by conceptual principles and not merely by empirical observations.

1.1.3 Contents of this book

This book consists of ten chapters and seven appendices. The chapters constitute the core of this book and are written in a style adequate for a textbook, whereas the appendices provide additional perspective and are written in the style of a survey article.

Section 1.2 and Chapter 2 are a prerequisite to the rest of the book. Technically, it is the case that notions and results that appear in these parts are extensively used in the rest of the book. More importantly, the former parts are the conceptual framework that shapes the field and provides a good perspective on the questions and answers provided. Indeed, Section 1.2 and Chapter 2 provide the very basic material that must be understood by anybody having an interest in complexity theory.

In contrast, the rest of the book covers more advanced material, which means that none of it can be claimed to be absolutely necessary for a basic understanding of complexity theory. Indeed, although some advanced chapters refer to material in other advanced chapters, the relation between these chapters is not a fundamental one. Thus, one may choose to read and/or teach an arbitrary subset of the advanced chapters and do it in an arbitrary order, provided one is willing to follow the relevant references to some parts of other chapters (see Figure 1.1). Needless to say, we recommend reading and/or teaching all the advanced chapters, and doing so by following the order presented in this book.



Solid arrows indicate the use of specific results that are stated in the section to which the arrow points. Dashed lines (and arrows) indicate an important conceptual connection; the wider the line, the tighter the connection. When relations are only between subsections, their index is indicated.

Figure 1.1: Dependencies among the advanced chapters.

The rest of this section provides a brief summary of the contents of the various chapters and appendices. This summary is intended for the teacher and/or the expert, whereas the student is referred to the more reader-friendly summaries that appear in the book's prefix.

Section 1.2: Preliminaries. This section provides the relevant background on computability theory, which is the basis for the rest of this book (as well as for complexity theory at large). Most importantly, it contains a discussion of central notions such as search and decision problems, algorithms that solve such problems, and their complexity. In addition, this section presents non-uniform models of computation (e.g., Boolean circuits).

Chapter 2: P, NP and NP-completeness. This chapter presents the P-vs-NP Question both in terms of search problems and in terms of decision problems. The second main topic of this chapter is the theory of NP-completeness. The chapter also provides a treatment of the general notion of a (polynomial-time) reduction, with special emphasis on self-reducibility. Additional topics include the existence of problems in NP that are neither NP-complete nor in P, optimal search algorithms, the class coNP, and promise problems.

Chapter 3: Variations on P and NP. This chapter provides a treatment of non-uniform polynomial-time (P/poly) and of the Polynomial-time Hierarchy (PH). Each of the two classes is defined in two equivalent ways (e.g., P/poly is defined both in terms of circuits and in terms of "machines that take advice"). In addition, it is shown that if NP is contained in P/poly then PH collapses to its second level (i.e., Σ_2).

Chapter 4: More Resources, More Power? The focus of this chapter is on Hierarchy Theorems, which assert that typically more resources allow for solving more problems. These results depend on using bounding functions that can be computed without exceeding the amount of resources that they specify, and otherwise Gap Theorems may apply.

Chapter 5: Space Complexity. Among the results presented in this chapter are a log-space algorithm for testing connectivity of (undirected) graphs, a proof that $\mathcal{NL} = \text{co}\mathcal{NL}$, and complete problems for $\mathcal{NL}$ and $\mathcal{PSPACE}$ (under log-space and poly-time reductions, respectively).

Chapter 6: Randomness and Counting. This chapter focuses on various randomized complexity classes (i.e., $\mathcal{BPP}$, $\mathcal{RP}$, and $\mathcal{ZPP}$) and the counting class $\#\mathcal{P}$. The results presented in this chapter include $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$ and $\mathcal{BPP} \subseteq \Sigma_2$, the $\#\mathcal{P}$-completeness of the `Permanent`, the connection between approximate counting and uniform generation of solutions, and the randomized reductions of approximate counting to $\mathcal{NP}$ and of $\mathcal{NP}$ to solving problems with unique solutions.

Chapter 7: The Bright Side of Hardness. This chapter deals with two conjectures that are related to $\mathcal{P} \neq \mathcal{NP}$. The first conjecture is that there are problems in $\mathcal{E}$ that are not solvable by (non-uniform) families of small (say polynomial-size) circuits, whereas the second conjecture is equivalent to the notion of *one-way functions*. Most of this chapter is devoted to "hardness amplification" results that convert these conjectures into tools that can be used for non-trivial derandomizations of $\mathcal{BPP}$ (resp., for a host of cryptographic applications).

Chapter 8: Pseudorandom Generators. The pivot of this chapter is the notion of *computational indistinguishability* and corresponding notions of pseudorandomness. The definition of general-purpose pseudorandom generators (running in polynomial-time and withstanding any polynomial-time distinguisher) is presented as a special case of a general paradigm. The chapter also contains a presentation of other instantiations of the latter paradigm, including generators aimed at derandomizing complexity classes such as $\mathcal{BPP}$, generators withstanding space-bounded distinguishers, and some special-purpose generators.

Chapter 9: Probabilistic Proof Systems. This chapter provides a treatment of three types of probabilistic proof systems: *interactive proofs*, *zero-knowledge proofs*, and *probabilistic checkable proofs*. The results presented include $\mathcal{IP} = \mathcal{PSPACE}$, zero-knowledge proofs for any NP-set, and the PCP Theorem. For the latter, only overviews of the two different known proofs are provided.

Chapter 10: Relaxing the Requirement. This chapter provides a treatment of two types of approximation problems and a theory of average-case (or rather typical-case) complexity. The traditional type of approximation problems refers to search problems and consists of a relaxation of standard optimization problems. The second type is known as "property testing" and consists of a relaxation of standard decision problems. The theory of average-case complexity involves several non-trivial definitional choices (e.g., an adequate choice of the class of distributions).

Appendix A: Glossary of Complexity Classes. The glossary provides self-contained definitions of most complexity classes mentioned in the book.

Appendix B: On the Quest for Lower Bounds. The first part, devoted to Circuit Complexity, reviews lower bounds for the *size* of (restricted) circuits that solve natural computational problems. The second part, devoted to Proof Complexity, reviews lower bounds on the length of (restricted) propositional proofs of natural tautologies.

Appendix C: On the Foundations of Modern Cryptography. The first part of this appendix augments the partial treatment of one-way functions, pseudorandom generators, and zero-knowledge proofs (which is included in Chapters

7–9). Using these basic tools, the second part provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

Appendix D: Probabilistic Preliminaries and Advanced Topics in Randomization. The probabilistic preliminaries include conventions regarding random variables and overviews of three useful inequalities (i.e., Markov Inequality, Chebyshev's Inequality, and Chernoff Bound). The advanced topics include constructions of *hashing* functions and variants of the Leftover Hashing Lemma, and overviews of *samplers* and *extractors* (i.e., the problem of randomness extraction).

Appendix E: Explicit Constructions. This appendix focuses on various computational aspects of error correcting codes and expander graphs. On the topic of codes, the appendix contains a review of the Hadamard code, Reed-Solomon codes, Reed-Muller codes, and a construction of a binary code of constant rate and constant relative distance. Also included are a brief review of the notions of locally testable and locally decodable codes, and a list-decoding bound. On the topic of expander graphs, the appendix contains a review of the standard definitions and properties as well as a presentation of the Margulis-Gabber-Galil and the Zig-Zag constructions.

Appendix F: Some Omitted Proofs. This appendix contains some proofs that are beneficial as alternatives to the original and/or standard presentations. Included are proofs that $\mathcal{PH}$ is reducible to $\#\mathcal{P}$ via randomized Karp-reductions, and that $\mathcal{IP}(f) \subseteq \mathcal{AM}(O(f)) \subseteq \mathcal{AM}(f)$.

Appendix G: Some Computational Problems. This appendix contains a brief introduction to graph algorithms, Boolean formulae, and finite fields.

Bibliography. As stated in Section 1.1.4, we tried to keep the bibliographic list as short as possible (and still reached a couple of hundreds of entries). As a result many relevant references were omitted. In general, our choice of references was biased in favor of textbooks and survey articles. We tried, however, not to omit references to key papers in an area.

Absent from this book. As stated in the preface, the current book does not provide a uniform cover of the various areas of complexity theory. Notable omissions include the areas of *circuit complexity* (cf. [43, 223]) and *proof complexity* (cf. [25]), which are briefly reviewed in Appendix B. Additional topics that are commonly covered in complexity theory courses but omitted here include the study of *branching programs* and *decision trees* (cf. [224]), *parallel computation* [133], and *communication complexity* [141]. We mention that the recent textbook of Arora and Barak [13] contains a treatment of all these topics. Finally, we mention a two areas that we consider related to complexity theory, although this view is not very common. These areas are *distributed computing* [16] and *computational learning theory* [135].

1.1.4 Approach and style of this book

According to a common opinion, the most important aspect of a scientific work is the technical result that it achieves and the rest is redundancy introduced for the sake of "error correction" and/or comfort. It is further believed that, like in a work of art, the interpretation of the work should be left with the reader (or viewer or listener).

The author disagrees with the aforementioned opinions, and argues that there is a fundamental difference between art and science, and that this difference refers exactly to the meaning of a piece of work. Science is concerned with meaning (and not with form), and in its quest for truth and/or understanding it follows philosophy (and not art). The author holds the opinion that the most important aspects of a scientific work are the intuitive question that it addresses, the reason that it addresses this question, the way it phrases the question, the approach that underlies its answer, and the ideas that are embedded in the answer. Following this view, it is important to communicate these aspects of the work, and the current book is written accordingly.

The foregoing issues are even more acute when it comes to complexity theory, firstly because conceptual considerations seems to play an even more central role in complexity theory (as opposed to other fields; cf., Section 1.1.2). Furthermore (or maybe consequently), complexity theory is extremely rich in conceptual content. Unfortunately, this content is rarely communicated (explicitly) in books and/or surveys of the area.[3] The annoying (and quite amazing) consequences are students that have only a vague understanding of the *meaning* and general relevance of the fundamental notions and results that they were taught. The author's view is that these consequences are easy to avoid by taking the time to explicitly discuss the *meaning* of definitions and results. A related issue is using the "right" definitions (i.e., those that reflect better the fundamental nature of the notion being defined) and teaching things in the (conceptually) "right" order.

1.1.4.1 The general principle

In accordance with the foregoing, the focus of this book is on the conceptual aspects of the technical material. Whenever presenting a subject, the starting point is the intuitive questions being addressed. The presentation explains the *importance* of these questions, the specific ways that they are phrased (i.e., the *choices* made in the actual formulation), the *approaches* that underly the answers, and the *ideas* that are embedded in these answers. Thus, a significant portion of the text is

[3]It is tempting to speculate on the reasons for this phenomenon. One speculation is that communicating the conceptual content of complexity theory involves making bold philosophical assertions that are technically straightforward, whereas this combination does not fit the character of most researchers in complexity theory.

devoted to motivating discussions that refer to the concepts and ideas that underly the actual definitions and results.

The material is organized around conceptual themes, which reflect fundamental notions and/or general questions. Specific computational problems are rarely referred to, with exceptions that are used either for sake of clarity or because the specific problem happens to capture a general conceptual phenomenon. For example, in this book, complete problems are always secondary to the class for which they are complete.[4]

1.1.4.2 On a few specific choices

Our technical presentation often differs from the standard one. In many cases this is due to conceptual considerations. At times, this leads to some technical simplifications. In this section we only discuss general themes and/or choices that have a global impact on much of the presentation.

Avoiding non-deterministic machines. We try to avoid non-deterministic machines as much as possible. As argued in several places (e.g., Section 2.1.4), we believe that these fictitious "machines" have a negative effect both from a conceptual and technical point of view. The conceptual damage caused by using non-deterministic machines is that it is unclear why one should care about what such machines can do. Needless to say, the reason to care is clear when noting that these fictitious "machines" offer a (convenient or rather slothful) way of phrasing fundamental issues. The technical damage caused by using non-deterministic machines is that they tend to confuse the students. Furthermore, they do not offer the best way to handle more advanced issues (e.g., counting classes).

In contrast, we use search problems as the basis for much of the presentation. Specifically, the class $\mathcal{PC}$ (see Definition 2.3), which consists of search problems having efficiently checkable solutions, plays a central role in our presentation. Indeed, defining this class is slightly more complicated than the standard definition of $\mathcal{NP}$ (based on non-deterministic machines), but the technical benefits start accumulating as we proceed. Needless to say, the class $\mathcal{PC}$ is a fundamental class of computational problems and this fact is the main motivation to its presentation. (Indeed, the most conceptually appealing phrasing of the P-vs-NP Question consists of asking whether every search problem in $\mathcal{PC}$ can be solved efficiently.)

Avoiding model-dependent effects. Our focus is on the notion of efficient computation. A rigorous definition of this notion seems to require reference to some concrete model of computation; however, all questions and answers considered

[4] We admit that a very natural computational problem can give rise to a class of problems that are computationally equivalent to it, and that in such a case the class may be less interesting than the original problem. This is not the case for any of the complexity classes presented in this book. Still, in some cases (e.g., $\mathcal{NP}$ and $\#\mathcal{P}$), the historical evolution actually went from a specific computational problem to a class of problems that are computationally equivalent to it. However, in all cases presented in this book, a retrospective evaluation suggests that the class is actually more important than the original problem.

in this book are invariant under the choice of such a concrete model, provided of course that the model is "reasonable" (which, needless to say, is a matter of intuition). Indeed, the foregoing text reflects the tension between the need to make rigorous definitions and the desire to be independent of technical choices, which are unavoidable when making such definitions. Furthermore, in contrast to common beliefs, the foregoing comments refer not only to time-complexity but also to space-complexity. However, in both cases, the claim of invariance may not hold for marginally small resources (e.g., linear-time or sub-logarithmic space).

In contrast to the foregoing paragraph, in some cases we choose to be specific. The most notorious case is the association of efficiency with polynomial-time (see §1.2.3.4). Indeed, all the questions and answers regarding efficient computation can be phrased without referring to polynomial-time (i.e., by stating explicit functional relations between the complexities of the problems involved), but such a generalized treatment will be painful to follow.

1.1.4.3 On the presentation of technical material

In general, the more complex the technical material is, the more levels of expositions we employ (starting from the most high-level exposition, and when necessary providing more than one level of details). In particular, whenever a proof is not very simple, we try to present the key ideas first, and postpone implementation details to later. We also try to clearly indicate the passage from a high-level presentation to its implementation details (e.g., by using phrases such as "details follow"). In some cases, especially in the case of advanced results, only proof sketches are provided and the implication is that the reader should be able to fill-up the missing details.

Few results are stated without a proof. In some of these cases the proof idea or a proof overview is provided, but the reader is *not* expected to be able to fill-up the highly non-trivial details. (In these cases, the text clearly indicates this state of affairs.) One notable example is the proof of the PCP Theorem (Theorem 9.16).

We tried to avoid the presentation of material that, in our opinion, is neither the "last word" on the subject nor represents the "right" way of approaching the subject. Thus, we do not always present the "best" known result.

1.1.4.4 Organizational principles

Each of the main chapters starts with a high-level summary and ends with chapter notes and exercises. The latter are not aimed at testing or inspiring creativity, but are rather designed to help and verify the basic understanding of the main text.

As stated in the preface, this book focuses on the high-level approach to complexity theory and the low-level approach (i.e., lower bounds) is briefly reviewed in Appendix B. Other appendices contain material that is closely related to complexity theory but is not an integral part of it (e.g., the Foundations of Cryptography).[5]

[5] As further articulated in Section 7.1, we recommend not including a basic treatment of cryptography within a course on complexity theory. Indeed, cryptography may be claimed to be the most appealing application of complexity theory, but a superficial treatment of cryptography

In an attempt to keep the bibliographic list from becoming longer than an average chapter, we omitted many relevant references. One trick used towards this end is referring to lists of references in other texts, especially when these texts are cited anyhow. Indeed, our choices of references were biased in favor of textbooks and survey articles, because we believe that they provide the best way to further learn about a research direction and/or approach. We tried, however, not to omit references to key papers in an area. In some cases, when we needed a reference for a result of interest and could not resort to the aforementioned trick, we cited also less central work.

As a matter of policy, we tried to avoid credits in the main text. The few exceptions are either pointers to texts that provide details that we chose to omit or usage of terms (bearing researchers' names) that are too popular to avoid.

> **Teaching note:** The text also includes some teaching notes, which are typeset as this one. Some of these notes express quite opinionated recommendations and/or justify various expositional choices made in the text.

1.1.4.5 Additional notes

The author's guess is that the text will be criticized for lengthy discussions of technically trivial issues. Indeed, most researchers dismiss various conceptual clarifications as being trivial and devote all their attention to the technically challenging parts of the material. The consequence is students that master the technical material but are confused about its meaning. In contrast, the author recommends not being embarrassed of devoting time to conceptual clarifications, even if some students may view them as obvious.

The motivational discussions presented in the text do not necessarily represent the original motivation of the researchers that pioneered a specific study and/or contributed greatly to it. Instead, these discussions provide what the author considers to be a good motivation and/or perspective on the corresponding concepts.

1.1.5 Standard notations and other conventions

Following are some notations and conventions that are freely used in this book.

Standard asymptotic notation: When referring to integral functions, we use the standard asymptotic notation; that is, for $f, g : \mathbb{N} \to \mathbb{N}$, we write $f = O(g)$ (resp., $f = \Omega(g)$) if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ (resp., $f(n) \geq c \cdot g(n)$) holds for all $n \in \mathbb{N}$. We usually denote by "poly" an unspecified polynomial, and write $f(n) = \text{poly}(n)$ instead of "there exists a polynomial p such that $f(n) \leq p(n)$ for all $n \in \mathbb{N}$." We also use the notation $f = \widetilde{O}(g)$ that mean $f(n) = \text{poly}(\log n) \cdot g(n)$, and $f = o(g)$ (resp., $f = \omega(g)$) that mean $f(n) < c \cdot g(n)$ (resp., $f(n) > c \cdot g(n)$) for every constant $c > 0$ and all sufficiently large n.

(from this perspective) is likely to be misleading and cause more harm than good.

Integrality issues: Typically, we ignore integrality issues. This means that we may assume that $\log_2 n$ is an integer rather than using a more cumbersome form as $\lfloor \log_2 n \rfloor$. Likewise, we may assume that various equalities are satisfied by integers (e.g., $2^n = m^m$), even when this cannot possibly be the case (e.g., $2^n = 3^m$). In all these cases, one should consider integers that approximately satisfy the relevant equations (and deal with the problems that emerge by such approximations, which will be ignored in the current text).

Standard combinatorial and graph theory terms and notation: For any set S, we denote by 2^S the set of all subsets of S (i.e., $2^S = \{S' : S' \subseteq S\}$). For a natural number $n \in \mathbb{N}$, we denote $[n] \stackrel{\text{def}}{=} \{1, ..., n\}$. Many of the computational problems refer to finite (undirected) graphs. Such a graph, denoted $G = (V, E)$, consists of a set of vertices, denoted V, and a set of edges, denoted E, which are unordered pairs of vertices. By default, graphs are undirected, whereas directed graphs consists of vertices and directed edges, where a directed edge is an order pair of vertices. We also refer to other graph theoretic terms such as connectivity, being acyclic (i.e., having no simple cycles), being a tree (i.e., being connected and acyclic), k-colorability, etc. For further background on graphs and computational problems regarding graphs, the reader is referred to Appendix G.1.

Typographic conventions: We denote formally defined complexity classes by calligraphic letters (e.g., $\mathcal{NP}$), but we do so only after defining these classes. Furthermore, when we wish to maintain some ambiguity regarding the specific formulation of a class of problems we use Roman font (e.g., NP may denote either a class of search problems or a class of decision problems). Likewise, we denote formally defined computational problems by typewriter font (e.g., `SAT`). In contrast, generic problems and algorithms will be denoted by italic font.

1.2 Computational Tasks and Models

We start by introducing the general framework for our discussion of computational tasks (or problems) This framework refers to the representation of instances and to two types of tasks (i.e., searching for solutions and making decisions). Once the stage is set, we consider two types of models of computation: uniform models that correspond to the intuitive notion of an algorithm, and non-uniform models (e.g., Boolean circuits) that facilitates a closer look at the way computation progresses.

Contents of Section 1.2. The contents of Sections 1.2.1–1.2.3 corresponds to a traditional *Computability course*, except that it includes a keen interest in universal machines (see §1.2.3.3), a discussion of the association of efficient computation with polynomial-time algorithm (§1.2.3.4), and a definition of oracle machines (§1.2.3.5). This material (with the exception of Kolmogorov Complexity) is taken for granted in the rest of the current book. (We also call the reader's attention to the discussion of generic complexity classes in Section 1.2.5.) In contrast, Section 1.2.4

presents basic preliminaries regarding non-uniform models of computation (i.e., various types of Boolean circuits), and these are only used lightly in the rest of the book. Thus, whereas Sections 1.2.1–1.2.3 (and 1.2.5) are absolute prerequisites for the rest of this book, Section 1.2.4 is not.

Teaching note: The author believes that there is no real need for a semester-long course in Computability (i.e., a course that focuses on what can be computed rather than on what can be computed efficiently). Instead, undergraduates should take a course in Computational Complexity, which should contain the computability aspects that serve as a basis for the rest of the course. Specifically, the former aspects should occupy at most 25% of the course, and the focus should be on basic complexity issues (captured by P, NP, and NP-completeness) augmented by a selection of some more advanced material. Indeed, such a course can be based on Chapters 1 and 2 of the current book (augmented by a selection of some topics from other chapters).

1.2.1 Representation

In Mathematics and related sciences, it is customary to discuss objects without specifying their representation. This is not possible in the theory of computation, where the representation of objects plays a central role. In a sense, a computation merely transforms one representation of an object to another representation of the same object. In particular, a computation designed to solve some problem merely transforms the problem instance to its solution, where the latter can be though of as a (possibly partial) representation of the instance. Indeed, the answer to any fully specified question is implicit in the question itself.

Computation refers to objects that are represented in some canonical way, where such canonical representation provides an "explicit" and "full" (but not "overly redundant") description of the corresponding object. We will consider only *finite* objects like sets, graphs, numbers, and functions (and keep distinguishing these types of objects although, actually, they are all equivalent). (For example, see Appendix G.1 for a discussion of the representation of graphs.)

Strings. We consider finite objects, each represented by a finite binary sequence, called a string. For a natural number n, we denote by $\{0,1\}^n$ the set of all strings of length n, hereafter referred to as n-bit strings. The set of all strings is denoted $\{0,1\}^*$; that is, $\{0,1\}^* = \cup_{n\in\mathbb{N}}\{0,1\}^n$. For $x\in\{0,1\}^*$, we denote by $|x|$ the length of x (i.e., $x\in\{0,1\}^{|x|}$), and often denote by x_i the i^{th} bit of x (i.e., $x = x_1x_2\cdots x_{|x|}$). For $x,y\in\{0,1\}^*$, we denote by xy the string resulting from concatenation of the strings x and y.

At times, we associate $\{0,1\}^*\times\{0,1\}^*$ with $\{0,1\}^*$; the reader should merely consider an adequate encoding (e.g., the pair $(x_1\cdots x_m, y_1\cdots y_n)\in\{0,1\}^*\times\{0,1\}^*$ may be encoded by the string $x_1x_1\cdots x_mx_m01y_1\cdots y_n \in \{0,1\}^*$). Likewise, we may represent sequences of strings (of fixed or varying length) as single strings. When we wish to emphasize that such a sequence (or some other object) is to be considered as a single object we use the notation $\langle\cdot\rangle$ (e.g., "the pair (x,y) is encoded as the string $\langle x,y\rangle$").

Numbers. Unless stated differently, natural numbers will be encoded by their binary expansion; that is, the string $b_{n-1} \cdots b_1 b_0 \in \{0,1\}^n$ encodes the number $\sum_{i=0}^{n-1} b_i \cdot 2^i$, where typically we assume that this representation has no leading zeros (i.e., $b_{n-1} = 1$). Rational numbers will be represented as pairs of natural numbers. In the rare cases in which one considers real numbers as part of the input to a computational problem, one actually mean rational approximations of these real numbers.

Special symbols. We denote the empty string by λ (i.e., $\lambda \in \{0,1\}^*$ and $|\lambda| = 0$), and the empty set by $\emptyset$. It will be convenient to use some special symbols that are not in $\{0,1\}^*$. One such symbol is $\perp$, which typically denotes an indication by some algorithm that something is wrong.

1.2.2 Computational Tasks

Two fundamental types of computational tasks are so-called search problems and decision problems. In both cases, the key notions are the problem's *instances* and the problem's specification.

1.2.2.1 Search problems

A search problem consists of a specification of a set of valid solutions (possibly an empty one) for each possible instance. That is, given an instance, one is required to find a corresponding solution (or to determine that no such solution exists). For example, consider the problem in which one is given a system of equations and is asked to find a valid solution. Needless to say, much of computer science is concerned with solving various search problems (e.g., finding shortest paths in a graph, sorting a list of numbers, finding an occurrence of a given pattern in a given string, etc). Furthermore, search problems correspond to the daily notion of "solving a problem" (e.g., finding one's way between two locations), and thus a discussion of the possibility and complexity of solving search problems corresponds to the natural concerns of most people.

In the following definition of solving search problems, the potential solver is a function (which may be thought of as a solving strategy), and the sets of possible solutions associated with each of the various instances are "packed" into a single binary relation.

Definition 1.1 (solving a search problem): *Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ and $R(x) \stackrel{\rm def}{=} \{y : (x,y) \in R\}$ denote the set of solutions for the instance x. A function $f : \{0,1\}^* \to \{0,1\}^* \cup \{\perp\}$* solves the search problem of *R if for every x the following holds: if $R(x) \neq \emptyset$ then $f(x) \in R(x)$ and otherwise $f(x) = \perp$.*

Indeed, $R = \{(x,y) : y \in R(x)\}$, and the solver f is required to find a solution (i.e., given x output $y \in R(x)$) whenever one exists (i.e., the set $R(x)$ is not empty). It is also required that the solver f never outputs a wrong solution (i.e., if $R(x) \neq \emptyset$

then $f(x) \in R(x)$ and if $R(x) = \emptyset$ then $f(x) = \perp$), which in turn means that f indicates whether x has any solution.

A special case of interest is the case of search problems having a unique solution (for each possible instance); that is, the case that $|R(x)| = 1$ for every x. In this case, R is essentially a (total) function, and solving the search problem of R means computing (or evaluating) the function R (or rather the function R' defined by $R'(x) \stackrel{\rm def}{=} y$ where $R(x) = \{y\}$). Popular examples include sorting a sequence of numbers, multiplying integers, finding the prime factorization of a composite number, etc.

1.2.2.2 Decision problems

A decision problem consists of a specification of a subset of the possible instances. Given an instance, one is required to determine whether the instance is in the specified set (e.g., the set of prime numbers, the set of connected graphs, or the set of sorted sequences). For example, consider the problem where one is given a natural number, and is asked to determine whether or not the number is a prime. One important case, which corresponds to the aforementioned search problems, is the case of the set of instances having a solution; that is, for any binary relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ we consider the set $\{x : R(x) \neq \emptyset\}$. Indeed, being able to determine whether or not a solution exists is a prerequisite to being able to solve the corresponding search problem (as per Definition 1.1). In general, decision problems refer to the natural task of making binary decision, a task that is not uncommon in daily life (e.g., determining whether a traffic light is red). In any case, in the following definition of solving decision problems, the potential solver is again a function (i.e., in this case it is a Boolean function that is supposed to indicate membership in the said set).

Definition 1.2 (solving a decision problem): *Let $S \subseteq \{0,1\}^*$. A function $f : \{0,1\}^* \to \{0,1\}$ solves the decision problem of S (or decides membership in S) if for every x it holds that $f(x) = 1$ if and only if $x \in S$.*

We often identify the decision problem of S with S itself, and identify S with its characteristic function (i.e., with $\chi_S : \{0,1\}^* \to \{0,1\}$ defined such that $\chi_S(x) = 1$ if and only if $x \in S$). Note that if f solves the search problem of R then the Boolean function $f' : \{0,1\}^* \to \{0,1\}$ defined by $f'(x) \stackrel{\rm def}{=} 1$ if and only if $f(x) \neq \perp$ solves the decision problem of $\{x : R(x) \neq \emptyset\}$.

Most people would consider search problems to be more natural than decision problems: typically, people seeks solutions more than they stop to wonder whether or not solutions exist. Definitely, search problems are not less important than decision problems, it is merely that their study tends to require more cumbersome formulations. This is the main reason that most expositions choose to focus on decision problems. The current book attempts to devote at least a significant amount of attention also to search problems.

1.2.2.3 Promise problems (an advanced comment)

Many natural search and decision problems are captured more naturally by the terminology of promise problems, where the domain of possible instances is a subset of $\{0,1\}^*$ rather than $\{0,1\}^*$ itself. In particular, note that the natural formulation of many search and decision problems refers to instances of a certain types (e.g., a system of equations, a pair of numbers, a graph), whereas the natural representation of these objects uses only a strict subset of $\{0,1\}^*$. For the time being, we ignore this issue, but we shall re-visit it in Section 2.4.1. Here we just note that, in typical cases, the issue can be ignored by postulating that every string represents some legitimate object (i.e., each string that is not used in the natural representation of these objects is postulated as a representation of some fixed object).

1.2.3 Uniform Models (Algorithms)

We are all familiar with computers and with the ability of computer programs to manipulate data. This familiarity seems to be rooted in the positive side of computing; that is, we have some experience regarding some things that computers can do. In contrast, complexity theory is focused at what computers cannot do, or rather with drawing the line between what can be done and what cannot be done. Drawing such a line requires a precise formulation of *all* possible computational processes; that is, we should have a clear model of *all* possible computational processes (rather than some familiarity with some computational processes).

Before being formal, let we offer a general and abstract description, which is aimed at capturing any artificial as well as natural process. Indeed, artificial processes will be associated with computers, whereas by natural processes we mean (attempts to model) the "mechanical" aspects the natural reality (be it physical, biological, or even social).

A computation is a process that modifies an environment via repeated applications of a predetermined rule. The key restriction is that this rule is *simple*: in each application it depends and affects only a (small) portion of the environment, called the active zone. We contrast the *a-priori bounded* size of the active zone (and of the modification rule) with the *a-priori unbounded* size of the entire environment. We note that, although each application of the rule has a very limited effect, the effect of many applications of the rule may be very complex. Put in other words, a computation may modify the relevant environment in a very complex way, although it is merely a process of repeatedly applying a simple rule.

As hinted, the notion of computation can be used to model the "mechanical" aspects of the natural reality; that is, the rules that determine the evolution of the reality (rather than the specifics of reality itself). In this case, the evolution process that takes place in the natural reality is the starting point of the study, and the goal of the study is finding the (computation) rule that underlies this natural process. In a sense, the goal of Science at large can be phrased as finding (simple) rules that govern various aspects of reality (or rather one's abstraction of these aspects of reality).

Our focus, however, is on artificial computation rules designed by humans in order to achieve specific desired effects on a corresponding artificial environment. Thus, our starting point is a desired functionality, and our aim is to design computation rules that effect it. Such a computation rule is referred to as an algorithm. Loosely speaking, an algorithm corresponds to a computer program written in a high-level (abstract) programming language. Let us elaborate.

We are interested in the transformation of the environment affected by the computational process (or the algorithm). Throughout (most of) this book, we will assume that, *when invoked on any finite initial environment, the computation halts after a finite number of steps.* Typically, the initial environment to which the computation is applied encodes an input string, and the end environment (i.e., at termination of the computation) encodes an output string. We consider the mapping from inputs to outputs induced by the computation; that is, for each possible input x, we consider the output y obtained at the end of a computation initiated with input x, and say that the computation maps input x to output y. Thus, a computation rule (or an algorithm) determines a function (computed by it): this function is exactly the aforementioned mapping of inputs to outputs.

In the rest of this book (i.e., outside the current chapter), we will also consider the number of steps (i.e., applications of the rule) taken by the computation on each possible input. The latter function is called the time complexity of the computational process (or algorithm). While time complexity is defined per input, we will often considers it per input length, taking the maximum over all inputs of the same length.

In order to define computation (and computation time) rigorously, one needs to specify some model of computation; that is, provide a concrete definition of environments and a class of rules that may be applied to them. Such a model corresponds to an abstraction of a real computer (be it a PC, mainframe or network of computers). One simple abstract model that is commonly used is that of *Turing machines* (see, §1.2.3.1). Thus, specific algorithms are typically formalized by corresponding Turing machines (and their time complexity is represented by the time complexity of the corresponding Turing machines). We stress, however, that most results in the Theory of Computation hold regardless of the specific computational model used, as long as it is "reasonable" (i.e., satisfies the aforementioned simplicity condition and can perform some obviously simple computations).

What is being computed? The forgoing discussion has implicitly referred to algorithms (i.e., computational processes) as means of computing functions. Specifically, an algorithm A computes the function $f_A : \{0,1\}^* \to \{0,1\}^*$ defined by $f_A(x)=y$ if, when invoked on input x, algorithm A halts with output y. However, algorithms can also serve as means of "solving search problems" or "making decisions" (as in Definitions 1.1 and 1.2). Specifically, we will say that algorithm A solves the search problem of R (resp., decides membership in S) if f_A solves the search problem of R (resp., decides membership in S). In the rest of this exposition we associate the algorithm A with the function f_A computed by it; that is, we write $A(x)$ instead of $f_A(x)$. For sake of future reference, we summarize the foregoing discussion.

Definition 1.3 (algorithms as problem-solvers): *We denote by $A(x)$ the output of algorithm A on input x. Algorithm A* solves the search problem R (resp., the decision problem S) *if A, viewed as a function, solves R* (resp., S).

Organization of the rest of Section 1.2.3. In §1.2.3.1 we provide a sketchy description of the model of Turing machines. This is done merely for sake of providing a concrete model that supports the study of computation and its complexity, whereas most of the material in this book will not depend on the specifics of this model. In §1.2.3.2 and §1.2.3.2 we discuss two fundamental properties of any reasonable model of computation: the existence of uncomputable functions and the existence of universal computations. The time (and space) complexity of computation is defined in §1.2.3.4. We also discuss oracle machines and restricted models of computation (in §1.2.3.5 and §1.2.3.6, respectively).

1.2.3.1 Turing machines

The model of Turing machines offer a relatively simple formulation of the notion of an algorithm. The fact that the model is very simple complicates the design of machines that solve problems of interest, but makes the analysis of such machines simpler. Since the focus of complexity theory is on the analysis of machines and not on their design, the trade-off offers by this model is suitable for our purposes. We stress again that the model is merely used as a concrete formulation of the intuitive notion of an algorithm, whereas we actually care about the intuitive notion and not about its formulation. In particular, all results mentioned in this book hold for any other "reasonable" formulation of the notion of an algorithm.

The model of Turing machines is not meant to provide an accurate (or "tight") model of real-life computers, but rather to capture their inherent limitations and abilities (i.e., a computational task can be solved by a real-life computer if and only if it can be solved by a Turing machine). In comparison to real-life computers, the model of Turing machines is extremely over-simplified and abstract away many issues that are of great concern to computer practice. However, these issues are irrelevant to the higher-level questions addressed by complexity theory. Indeed, as usual, good practice requires more refined understanding than the one provided by a good theory, but one should first provide the latter.

Historically, the model of Turing machines was invented before modern computers were even built, and was meant to provide a concrete model of computation and a definition of computable functions.[6] Indeed, this concrete model clarified fundamental properties of computable functions and plays a key role in defining the complexity of computable functions.

The model of Turing machines was envisioned as an abstraction of the process of an algebraic computation carried out by a human using a sheet of paper. In such a process, at each time, the human looks at some location on the paper, and depending on what he/she sees and what he/she has in mind (which is little...),

[6]In contrast, the abstract definition of "recursive functions" yields a class of "computable" functions defined recursively in terms of the composition of such functions.

he/she modifies the contents of this location and shifts his/her look to an adjacent location.

The actual model. Following is a high-level description of the model of Turing machines; the interested reader is referred to standard textbooks (e.g., [198]) for further details. Recall that we need to specify the set of possible environments, the set of machines (or computation rules), and the effect of applying such a rule on an environment.

- The main component in the environment of a Turing machine is an infinite sequence of cells, each capable of holding a single symbol (i.e., member of a finite set $\Sigma \supset \{0,1\}$). In addition, the environment contains the current location of the machine on this sequence, and the internal state of the machine (which is a member of a finite set Q). The aforementioned sequence of cells is called the tape, and its contents combined with the machine's location and its internal state is called the instantaneous configuration of the machine.
- The Turing machine itself consists of a finite rule (i.e., a finite function), called the transition function, which is defined over the set of all possible symbol-state pairs. Specifically, the transition function is a mapping from $\Sigma \times Q$ to $\Sigma \times Q \times \{-1,0,+1\}$, where $\{-1,+1,0\}$ correspond to a movement instruction (which is either "left" or "right" or "stay", respectively). In addition, the machine's description specifies an initial state and a halting state, and the computation of the machine halts when the machine enters its halting state.[7]

  We stress that, in contrast to the finite description of the machine, the tape has an a priori unbounded length (and is considered, for simplicity, as being infinite).
- A single computation step of such a Turing machine depends on its current location on the tape, on the contents of the corresponding cell and on the internal state of the machine. Based on the latter two elements, the transition function determines a new symbol-state pair as well as a movement instruction (i.e., "left" or "right" or "stay"). The machine modifies the contents of the said cell and its internal state accordingly, and moves as directed. That is, suppose that the machine is in state q and resides in a cell containing the symbol σ, and suppose that the transition function maps (σ, q) to (σ', q', D). Then, the machine modifies the contents of the said cell to σ', modifies its internal state to q', and moves one cell in direction D. Figure 1.2 shows a single step of a Turing machine that, when in state 'b' and seeing a binary symbol σ, replaces σ with the symbol $\sigma + 2$, maintains its internal state, and moves one position to the right.[8]

[7]Envisioning the tape as extending from left to right, we also use the convention by which if the machine tries to move left of the end of the tape then it is considered to have halted.

[8]Figure 1.2 corresponds to a machine that, when in the initial state (i.e., 'a'), replaces the symbol σ by $\sigma + 4$, modifies its internal state to 'b', and moves one position to the right. Indeed, "marking" the leftmost cell (in order to allow for recognizing it in the future), is a common practice in the design of Turing machines.

Formally, we define the successive configuration function that maps each instantaneous configuration to the one resulting by letting the machine take a single step. This function modifies its argument in a very minor manner, as described in the foregoing; that is, the contents of at most one cell (i.e., at which the machine currently resides) is changed, and in addition the internal state of the machine and its location may change too.

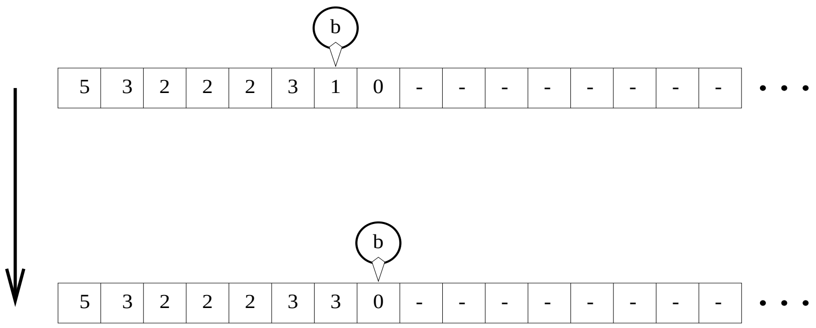


Figure 1.2: A single step by a Turing machine.

The initial environment (or configuration) of a Turing machine consists of the machine residing in the first (i.e., left-most) cell and being in its initial state. Typically, one also mandates that, in the initial configuration, a prefix of the tape's cells hold bit values, which concatenated together are considered the input, and the rest of the tape's cells hold a special symbol (which in Figure 1.2 is denoted by '-'). Once the machine halts, the output is defined as the contents of the cells that are to the left of its location (at termination time).[9] Thus, each machine defines a function mapping inputs to outputs, called the function computed by the machine.

Multi-tape Turing machines. We comment that in most expositions, one refers to the location of the "head of the machine" on the tape (rather than to the "location of the machine on the tape"). The standard terminology is more intuitive when extending the basic model, which refers to a single tape, to a model that supports a constant number of tapes. In the model of multi-tape machines, each step of the machine depends and effects the cells that are at the head location of the machine on each tape. As we shall see in Chapter 5 (and in §1.2.3.4), the extension of the model to multi-tape Turing machines is crucial to the definition of space complexity. A less fundamental advantage of the model of multi-tape Turing machines is that it facilitates the design of machines that compute functions of interest.

[9]By an alternative convention, the machine halts while residing in the left-most cell, and the output is defined as the maximal prefix of the tape contents that contains only bit values.

Teaching note: We strongly recommend avoiding the standard practice of teaching the student to program with Turing machines. These exercises seem very painful and pointless. Instead, one should prove that a function can be computed by a Turing machine if and only if it is computable by a model that is closer to a real-life computer (see the following "sanity check"). For starters, one should prove that a function can be computed by a single-tape Turing machine if and only if it is computable by a multi-tape (e.g., two-tape) Turing machine.

The Church-Turing Thesis: The entire point of the model of Turing machines is its simplicity. That is, in comparison to more "realistic" models of computation, it is simpler to formulate the model of Turing machines and to analyze machines in this model. The Church-Turing Thesis asserts that nothing is lost by considering the Turing machine model: *A function can be computed by some Turing machine if and only if it can be computed by some machine of any other* "reasonable and general" *model of computation.*

This is a thesis, rather than a theorem, because it refers to an intuitive notion that is left undefined on purpose (i.e., the notion of a *reasonable and general model of computation*). The model should be reasonable in the sense that it should refer to computation rules that are "simple" in some intuitive sense. On the other hand, the model should allow to compute functions that intuitively seem computable. At the very least the model should allow to emulate Turing machines (i.e., compute the function that given a description of a Turing machine and an instantaneous configuration returns the successive configuration).

A philosophical comment. The fact that a thesis is used to link an intuitive concept to a formal definition is common practice in any science (or, more broadly, in any attempt to reason rigorously about intuitive concepts). The moment an intuition is rigorously defined, it stops being an intuition and becomes a definition, and the question of the correspondence between the original intuition and the derived definition arises. This question can never be rigorously treated, because it relates to two objects, where one of them is undefined. Thus, the question of correspondence between the intuition and the definition always transcends a rigorous treatment (i.e., it always belongs to the domain of the intuition).

A sanity check: Turing machines can emulate an abstract RAM. To gain confidence in the Church-Turing Thesis, one may attempt to define an abstract Random-Access Machine (RAM), and verify that it can be emulated by a Turing machine. An abstract RAM consists of an infinite number of memory cells, each capable of holding an integer, a finite number of similar registers, one designated as program counter, and a program consisting of instructions selected from a finite set. The set of possible instructions includes the following instructions:

- `reset`(r), where r is an index of a register, results in setting the value of register r to zero.
- `inc`(r), where r is an index of a register, results in incrementing the content of register r. Similarly `dec`(r) causes a decrement.

- load(r_1,r_2), where r_1 and r_2 are indices of registers, results in loading to register r_1 the contents of the memory location m, where m is the current contents of register r_2.
- store(r_1,r_2), stores the contents of register r_1 in the memory, analogously to load.
- cond-goto(r,ℓ), where r is an index of a register and ℓ does not exceed the program length, results in setting the program counter to $\ell - 1$ if the content of register r is non-negative.

The program counter is incremented after the execution of each instruction, and the next instruction to be executed by the machine is the one to which the program counter points (and the machine halts if the program counter exceeds the program's length). The input to the machine may be defined as the contents of the first n memory cells, where n is placed in a special input register. We note that the RAM model satisfies the Church-Turing Thesis, but in order to make it closer to real-life computers we may augment the model with additional instructions that are available on such computers (e.g., the instruction add(r_1,r_2) (resp., mult(r_1,r_2)) that results in adding (resp., multiplying) the contents of registers r_1 and r_2 and placing the result in register r_1). We suggest proving that this abstract RAM can be emulated by a Turing machine.[10] (Hint: note that during the emulation, we only need to hold the input, the contents of all registers, and the contents of the memory cells that were accessed during the computation.)[11]

Observe that the abstract RAM model is significantly more cumbersome than the Turing machine model. Furthermore, seeking a sound choice of the instruction set (i.e., the instructions to be allowed in the model) creates a vicious cycle (because the sound guideline would have been to allow only instructions that correspond to "simple" operations, whereas the latter correspond to easily computable functions...). This vicious cycle was avoided by trusting the reader to consider only instructions that are available in some real-life computer. (We comment that this empirical consideration is justifiable in the current context, because our current goal is merely linking the Turing machine model with the reader's experience of real-life computers.)

1.2.3.2 Uncomputable functions

Strictly speaking, the current subsection is not necessary for the rest of this book, but we feel that it provides a useful perspective.

[10]We emphasize this direction of the equivalence of the two models, because the RAM model is introduced in order to convince the reader that Turing machines are not too weak (as a model of general computation). The fact that they are not too strong seems self-evident. Thus, it seems pointless to prove that the RAM model can emulate Turing machines. Still, note that this is indeed the case, by using the RAM's memory cells to store the contents of the cells of the Turing machine's tape.

[11]Thus, at each time, the Turning machine's tape contains a list of the RAM's memory cells that were accessed so far as well as their current contents. When we emulate a RAM instruction, we first check whether the relevant RAM cell appears on this list, and augment the list by a corresponding entry or modify this entry as needed.

In contrast to what every layman would think, we know that not all functions are computable. Indeed, an important message to be communicated to the world is that *not every well-defined task can be solved* by applying a "reasonable" procedure (i.e., a procedure that has a simple description that can be applied to any instance of the problem at hand). Furthermore, not only is it the case that there exist uncomputable functions, but it is rather the case that most functions are uncomputable. In fact, only relatively few functions are computable.

Theorem 1.4 (on the scarcity of computable functions): *The set of computable functions is countable, whereas the set of all functions* (from strings to string) *has cardinality* $\aleph$.

We stress that the theorem holds for any reasonable model of computation. In fact, it only relies on the postulate that each machine in the model has a finite description (i.e., can be described by a string).

Proof: Since each computable function is computable by a machine that has a finite description, there is a 1-1 correspondence between the set of computable functions and the set of strings (which in turn is in 1-1 correspondence to the natural numbers). On the other hand, there is a 1-1 correspondence between the set of Boolean functions (i.e., functions from strings to a bit) and the set of real number in $[0,1)$. This correspondence associates each real $r \in [0,1)$ to the function $f : \mathbb{N} \to \{0,1\}$ such that $f(i)$ is the i^{th} bit in the binary expansion of r. ∎

The Halting Problem: In contrast to the preliminary discussion, at this point we consider also machines that may not halt on some inputs. (The functions computed by such machines are partial functions that are defined only on inputs on which the machine halts.) Again, we rely on the postulate that each machine in the model has a finite description, and denote the description of machine M by $\langle M\rangle \in \{0,1\}^*$. The halting function, $\mathtt{h} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, is defined such that $\mathtt{h}(\langle M\rangle, x) \stackrel{\text{def}}{=} 1$ if and only if M halts on input x. The following result goes beyond Theorem 1.4 by pointing to an explicit function (of natural interest) that is not computable.

Theorem 1.5 (undecidability of the halting problem): *The halting function is not computable.*

The term undecidability means that the corresponding decision problem cannot be solved by an algorithm. That is, Theorem 1.5 asserts that the decision problem associated with the set $\mathtt{h}^{-1}(1) = \{(\langle M\rangle, x) : \mathtt{h}(\langle M\rangle, x) = 1\}$ is not solvable by an algorithm (i.e., there exists no algorithm that, given a pair $(\langle M\rangle, x)$, decides whether or not M halts on input x). Actually, the following proof shows that there exists no algorithm that, given $\langle M\rangle$, decides whether or not M halts on input $\langle M\rangle$.

Proof: We will show that even the restriction of $\mathtt{h}$ to its "diagonal" (i.e., the function $\mathtt{d}(\langle M\rangle) \stackrel{\text{def}}{=} \mathtt{h}(\langle M\rangle, \langle M\rangle)$) is not computable. Note that the value of $\mathtt{d}(\langle M\rangle)$ refers to the question of what happens when we feed M with its own description, which is indeed a "nasty" (but legitimate) thing to do. We will actually do worse: towards the contradiction, we will consider the value of $\mathtt{d}$ when evaluated at a (machine that is related to a) machine that supposedly computes $\mathtt{d}$.

We start by considering a related function, $\mathtt{d}'$, and showing that this function is uncomputable. This function is defined on purpose so to foil any attempt to compute it; that is, for every machine M, the value $\mathtt{d}'(\langle M\rangle)$ is defined to differ from $M(\langle M\rangle)$. Specifically, the function $\mathtt{d}' : \{0,1\}^* \to \{0,1\}$ is defined such that $\mathtt{d}'(\langle M\rangle) \stackrel{\rm def}{=} 1$ *if and only if M halts on input $\langle M\rangle$ with output* 0. (That is, $\mathtt{d}'(\langle M\rangle) = 0$ if either M does not halt on input $\langle M\rangle$ or its output does not equal the value 0.) Now, suppose, towards the contradiction, that $\mathtt{d}'$ is computable by some machine, denoted $M_{\mathtt{d}'}$. Note that machine $M_{\mathtt{d}'}$ is supposed to halt on every input, and so $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$. But, by definition of $\mathtt{d}'$, it holds that $\mathtt{d}'(\langle M_{\mathtt{d}'}\rangle) = 1$ if and only if $M_{\mathtt{d}'}$ halts on input $\langle M_{\mathtt{d}'}\rangle$ with output 0 (i.e., if and only if $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) = 0$). Thus, $M_{\mathtt{d}'}(\langle M_{\mathtt{d}'}\rangle) \neq \mathtt{d}'(\langle M_{\mathtt{d}'}\rangle)$ in contradiction to the hypothesis that $M_{\mathtt{d}'}$ computes $\mathtt{d}'$.

We next prove that $\mathtt{d}$ is uncomputable, and thus $\mathtt{h}$ is uncomputable (because $\mathtt{d}(z) = \mathtt{h}(z,z)$ for every z). To prove that $\mathtt{d}$ is uncomputable, we show that if $\mathtt{d}$ is computable then so is $\mathtt{d}'$ (which we already know not to be the case). Indeed, let A be an algorithm for computing $\mathtt{d}$ (i.e., $A(\langle M\rangle) = d(\langle M\rangle)$ for every machine M). Then we construct an algorithm for computing $\mathtt{d}'$, which given $\langle M'\rangle$, invokes A on $\langle M''\rangle$, where M'' is defined to operate as follows:

1. On input x, machine M'' emulates M' on input x.
2. If M' halts on input x with output 0 then M'' halts.
3. If M' halts on input x with an output different from 0 then M'' enters an infinite loop (and thus does not halt).
4. Otherwise (i.e., M' does not halt on input x), then machine M'' does not halt (because it just stays stuck in Step 1 forever).

Note that the mapping from $\langle M'\rangle$ to $\langle M''\rangle$ is easily computable (by augmenting M' with instructions to test its output and enter an infinite loop if necessary), and that $\mathtt{d}(\langle M''\rangle) = \mathtt{d}'(\langle M'\rangle)$, because M'' halts on x if and only if M'' halts on x with output 0. We thus derived an algorithm for computing $\mathtt{d}'$ (i.e., transform the input $\langle M'\rangle$ into $\langle M''\rangle$ and output $A(\langle M''\rangle)$), which contradicts the already established fact by which $\mathtt{d}'$ is uncomputable. ■

Turing-reductions. The core of the second part of the proof of Theorem 1.5 is an algorithm that solves one problem (i.e., computes $\mathtt{d}'$) by using as a subroutine an algorithm that solves another problem (i.e., computes $\mathtt{d}$ (or $\mathtt{h}$)). In fact, the first algorithm is actually an algorithmic scheme that refers to a "functionally specified" subroutine rather than to an actual (implementation of such a) subroutine, which may not exist. Such an algorithmic scheme is called a Turing-reduction (see formulation in §1.2.3.5). Hence, we have Turing-reduced the computation of $\mathtt{d}'$ to the computation of $\mathtt{d}$, which in turn Turing-reduces to $\mathtt{h}$. The "natural" ("positive") meaning of a Turing-reduction of f' to f is that when given an algorithm for computing f we obtain an algorithm for computing f'. In contrast, the proof of Theorem 1.5 uses the "unnatural" ("negative") counter-positive: if (as we know) there exists no algorithm for computing $f' = \mathtt{d}'$ then there exists no algorithm for computing $f = \mathtt{d}$ (which is what we wanted to prove). Jumping ahead, we mention that resource-bounded Turing-reductions (e.g., polynomial-time reductions) play a central role in complexity theory itself, and again they are used mostly in a "negative" way. We will define such reductions and extensively use them in subsequent chapters.

Rice's Theorem. The undecidability of the halting problem (or rather the fact that the function $\mathtt{d}$ is uncomputable) is a special case of a more general phenomenon: Every non-trivial decision problem *regarding the function computed by a given Turing machine* has no algorithmic solution. We state this fact next, clarifying what is the aforementioned class of problems. (Again, we refer to Turing machines that may not halt on all inputs.)

Theorem 1.6 (Rice's Theorem): *Let $\mathcal{F}$ be a non-trivial subset*[12] *of the set of all computable partial functions, and let $S_{\mathcal{F}}$ be the set of strings that describe machines that compute functions in $\mathcal{F}$. Then deciding membership in $S_{\mathcal{F}}$ cannot be solved by an algorithm.*

Theorem 1.6 can be proved by a Turing-reduction from $\mathtt{d}$. We do not provide a proof because this is too remote from the main subject matter of the book. We stress that Theorems 1.5 and 1.6 hold for any reasonable model of computation (referring both to the potential solvers and to the machines the description of which is given as input to these solvers). Thus, Theorem 1.6 means that *no algorithm can determine any non-trivial property of the function computed by a given computer program* (written in any programming language). For example, *no algorithm can determine whether or not a given computer program halts on each possible input.* The relevance of this assertion to the project of program verification is obvious.

The Post Correspondence Problem. We mention that undecidability arises also outside of the domain of questions regarding computing devices (given as input). Specifically, we consider the Post Correspondence Problem in which the input consists of two sequences of strings, $(\alpha_1, ..., \alpha_k)$ and $(\beta_1, ..., \beta_k)$, and the question is whether or not there exists a sequence of indices $i_1, ..., i_\ell \in \{1, ..., k\}$ such that $\alpha_{i_1} \cdots \alpha_{i_\ell} = \beta_{i_1} \cdots \beta_{i_\ell}$. (We stress that the length of this sequence is not bounded.)[13]

Theorem 1.7 *The Post Correspondence Problem is undecidable.*

[12] The set S is called a non-trivial subset of U if both S and $U \setminus S$ are non-empty. Clearly, if $\mathcal{F}$ is a trivial set of computable functions then the corresponding decision problem can be solved by a "trivial" algorithm that outputs the corresponding constant bit.

[13] In contrast, the existence of an adequate sequence of a specified length can be determined in time that is exponential in this length.

Again, the omitted proof is by a Turing-reduction from $\mathtt{d}$ (or $\mathtt{h}$).[14]

1.2.3.3 Universal algorithms

So far we have used the postulate that, in any reasonable model of computation, each machine (or computation rule) has a finite description. Furthermore, we also used the fact that such model should allow for the easy modification of such descriptions such that the resulting machine computes an easily related function (see the proof of Theorem 1.5). Here we go one step further and postulate that the description of machines (in this model) is "effective" in the following natural sense: there exists an algorithm that, given a description of a machine (resp., computation rule) and a corresponding environment, determines the environment that results from performing a single step of this machine on this environment (resp. the effect of a single application of the computation rule). This algorithm can, in turn, be implemented in the said model of computation (assuming this model is general; see the Church-Turing Thesis). Successive applications of this algorithm leads to the notion of a universal machine, which (for concreteness) is formulated next in terms of Turing machines.

Definition 1.8 (universal machines): *A* universal Turing machine *is a Turing machine that on input a description of a machine M and an input x returns the value of $M(x)$ if M halts on x and otherwise does not halt.*

That is, a universal Turing machine computes the partial function $\mathtt{u}$ that is defined over pairs $(\langle M\rangle, x)$ such that M halts on input x, in which case it holds that $\mathtt{u}(\langle M\rangle, x) = M(x)$. We note that if M halts on all possible inputs then $\mathtt{u}(\langle M\rangle, x)$ is defined for every x. We stress that the mere fact that we have defined something does not mean that it exists. Yet, as hinted in the foregoing discussion and obvious to anyone who has written a computer program (and thought about what he/she was doing), universal Turing machines do exist.

Theorem 1.9 *There exists a universal Turing machine.*

Theorem 1.9 asserts that the partial function $\mathtt{u}$ is computable. In contrast, it can be shown that any extension of $\mathtt{u}$ to a total function is uncomputable. That is, for any total function $\hat{\mathtt{u}}$ that agrees with the partial function $\mathtt{u}$ on all the inputs on which the latter is defined, it holds that $\hat{\mathtt{u}}$ is uncomputable.[15]

Proof: Given a pair $(\langle M\rangle, x)$, we just emulate the computation of machine M on input x. This emulation is straightforward, because (by the effectiveness of the

[14] We mention that the reduction maps an instance $(\langle M\rangle, x)$ of $\mathtt{h}$ to a pair of sequences such that only the first string in each sequence depends on x, whereas the other strings as well as their number depend only on M.

[15] The claim is easy to prove for the total function $\hat{\mathtt{u}}$ that extends $\mathtt{u}$ and assigns the special symbol $\perp$ to inputs on which $\mathtt{u}$ is undefined (i.e., $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \perp$ if $\mathtt{u}$ is not defined on $(\langle M\rangle, x)$ and $\hat{\mathtt{u}}(\langle M\rangle, x) \stackrel{\text{def}}{=} \mathtt{u}(\langle M\rangle, x)$ otherwise). In this case $\mathtt{h}(\langle M\rangle, x) = 1$ if and only if $\hat{\mathtt{u}}(\langle M\rangle, x) \neq \perp$, and so the halting function $\mathtt{h}$ is Turing-reducible to $\hat{\mathtt{u}}$. In the general case, we may adapt the proof of Theorem 1.5 by using the fact that, for a machine M that halts on every input, it holds that $\hat{\mathtt{u}}(\langle M\rangle, x) = \mathtt{u}(\langle M\rangle, x)$ for every x (and in particular for $x = \langle M\rangle$).

description of M) we can iteratively determine the next instantaneous configuration of the computation of M on input x. If the said computation halts then we will obtain its output and can output it (and so, on input $(\langle M\rangle, x)$, our algorithm returns $M(x)$). Otherwise, we turn out emulating an infinite computation, which means that our algorithm does not halt on input $(\langle M\rangle, x)$. Thus, the foregoing emulation procedure constitutes a universal machine (i.e., yields an algorithm for computing $\mathtt{u}$). ■

As hinted already, the existence of universal machines is the fundamental fact underlying the paradigm of general-purpose computers. Indeed, a specific Turing machine (or algorithm) is a device that solves a specific problem. A priori, solving each problem would have required building a new physical device that allows for this problem to be solved in the physical world (rather than as a thought experiment). The existence of a universal machine asserts that it is enough to build one physical device; that is, a general purpose computer. Any specific problem can then be solved by writing a corresponding program to be executed (or emulated) by the general purpose computer. Thus, universal machines correspond to general purpose computers, and provide the basis for separating hardware from software. In other words, the existence of universal machines says that software can be viewed as (part of the) input.

In addition to their practical importance, the existence of universal machines (and their variants) has important consequences in the theories of computability and computational complexity. Here we merely note that Theorem 1.9 implies that many questions about the behavior of a universal machine on certain input types are undecidable. For example, it follows that, for some fixed machines (i.e., universal ones), there is no algorithm that determines whether or not the (fixed) machine halts on a given input. Revisiting the proof of Theorem 1.7 (see Footnote 14), it follows that the Post Correspondence Problem remains undecidable even if the input sequences are restricted to have a specific length (i.e., k is fixed). A more important application of universal machines to the theory of computability follows.

A detour: Kolmogorov Complexity. The existence of universal machines, which may be viewed as universal languages for writing effective and succinct descriptions of objects, plays a central role in Kolmogorov Complexity. Loosely speaking, the latter theory is concerned with the length of (effective) descriptions of objects, and views the minimum such length as the inherent "complexity" of the object; that is, "simple" objects (or phenomena) are those having short description (resp., short explanation), whereas "complex" objects have no short description. Needless to say, these (effective) descriptions have to refer to some fixed "language" (i.e., to a fixed machine that, given a succinct description of an object, produces its explicit description). Fixing any machine M, a string x is called a description of s with respect to M if $M(x) = s$. The complexity of s with respect to M, denoted $K_M(s)$, is the length of the shortest description of s with respect to M. Certainly, we want to fix M such that every string has a description with respect to M, and furthermore such that this description is not "significantly" longer than the description with respect to a different machine M'. The following theorem make it natural to use a universal machine as the "point of reference" (i.e., as the aforementioned M).

Theorem 1.10 (complexity w.r.t a universal machine): *Let U be a universal machine. Then, for every machine M', there exists a constant c such that $K_U(s) \leq K_{M'}(s) + c$ for every string s.*

The theorem follows by (setting $c = O(|\langle M' \rangle|)$ and) observing that if x is a description of s with respect to M' then $(\langle M' \rangle, x)$ is a description of s with respect to U. Here it is important to use an adequate encoding of pairs of strings (e.g., the pair $(\sigma_1 \cdots \sigma_k, \tau_1 \cdots \tau_\ell)$ is encoded by the string $\sigma_1\sigma_1 \cdots \sigma_k\sigma_k 01\tau_1 \cdots \tau_\ell$). Fixing any universal machine U, we define the Kolmogorov Complexity of a string s as $K(s) \stackrel{\text{def}}{=} K_U(s)$. The reader may easily verify the following facts:

1. $K(s) \leq |s| + O(1)$, for every s.

   (Hint: apply Theorem 1.10 to a machine that computes the identity mapping.)

2. There exist infinitely many strings s such that $K(s) \ll |s|$.

   (Hint: consider $s = 1^n$. Alternatively, consider any machine M such that $|M(x)| \gg |x|$ for every x.)

3. Some strings of length n have complexity at least n. Furthermore, for every n and i,
   $$|\{s \in \{0,1\}^n : K(s) \leq n - i\}| < 2^{n-i+1}$$
   (Hint: different strings must have different descriptions with respect to U.)

It can be shown that *the function K is uncomputable*. The proof is related to the paradox captured by the following "description" of a natural number: `the largest natural number that can be described by an English sentence of up-to a thousand letters`. (The paradox amounts to observing that if the above number is well-defined then so is `the integer-successor of the largest natural number that can be described by an English sentence of up-to a thousand letters`.) Needless to say, the foregoing sentences presuppose that any English sentence is a legitimate description in some adequate sense (e.g., in the sense captured by Kolmogorov Complexity). Specifically, the foregoing sentences presuppose that we can determine the Kolmogorov Complexity of each natural number, and furthermore that we can effectively produce the largest number that has Kolmogorov Complexity not exceeding some threshold. Indeed, the paradox provides a proof to the fact that the latter task cannot be performed; that is, there exists no algorithm that given t produces the lexicographically last string s such that $K(s) \leq t$, because if such an algorithm A would have existed then $K(s) \leq O(|\langle A \rangle|) + \log t$ and $K(s0) < K(s) + O(1) < t$ in contradiction to the definition of s.

1.2.3.4 Time and space complexity

Fixing a model of computation (e.g., Turing machines) and focusing on algorithms that halt on each input, we consider the number of steps (i.e., applications of the computation rule) taken by the algorithm on each possible input. The latter function is called the time complexity of the algorithm (or machine); that is, $t_A : \{0,1\}^* \to \mathbb{N}$ is called the time complexity of algorithm A if, for every x, on input x algorithm A halts after exactly $t_A(x)$ steps.

We will be mostly interested in the dependence of the time complexity on the input length, when taking the maximum over all inputs of the relevant length. That is, for t_A as above, we will consider $T_A : \mathbb{N} \to \mathbb{N}$ defined by $T_A(n) \stackrel{\text{def}}{=} \max_{x\in\{0,1\}^n}\{t_A(x)\}$. Abusing terminology, we sometimes refer to T_A as the time complexity of A.

The time complexity of a problem. As stated in the preface and in the introduction, typically is complexity theory not concerned with the (time) complexity of a specific algorithm. It is rather concerned with the (time) complexity of a problem, assuming that this problem is solvable at all (by some algorithm). Intuitively, the time complexity of such a problem is defined as the time complexity of the fastest algorithm that solves this problem (assuming that the latter term is well-defined).[16] Actually, we shall be interested in upper and lower bounds on the (time) complexity of algorithms that solve the problem. However, the complexity of a problem may depend on the specific model of computation in which algorithms that solve it are implemented. The following Cobham-Edmonds Thesis asserts that the variation (in the time complexity) is not too big, and in particular is irrelevant to much of the current focus of complexity theory (e.g., for the P-vs-NP Question).

The Cobham-Edmonds Thesis. As just stated, the time complexity of a problem may depend on the model of computation. For example, deciding membership in the set $\{xx : x \in \{0,1\}^*\}$ can be done in linear-time on a two-tape Turing machine, but requires quadratic-time on a single-tape Turing machine.[17] On the other hand, any problem that has time complexity t in the model of multi-tape Turing machines, has complexity $O(t^2)$ in the model of single-tape Turing machines. The Cobham-Edmonds Thesis asserts that the time complexities in any two "reasonable and general" models of computation are polynomially related. That is, *a problem*

[16] **Advanced comment:** As we shall see in Section 4.2.2 (cf. Theorem 4.8), the naive assumption that a "fastest algorithm" for solving a problem exists is not always justified. On the other hand, the assumption is essentially justified in some important cases (see, e.g., Theorem 2.31). But even in these case the said algorithm is "fastest" (or "optimal") only up to a constant factor.

[17] Proving the latter fact is quite non-trivial. One proof is by a "reduction" from a communication complexity problem [141, Sec. 12.2]. Intuitively, a single-tape Turing machine that decides membership in the aforementioned set can be viewed as a channel of communication between the two parts of the input. Focusing our attention on inputs of the form $y0^nz0^n$, for $y, z \in \{0,1\}^n$, each time the machine passes from the first part to the second part it carries $O(1)$ bits of information (in its internal state) while making at least n steps. The proof is completed by invoking the linear lower bound on the communication complexity of the (two-argument) identity function (i.e, $\mathtt{id}(y,z) = 1$ if $y = z$ and $\mathtt{id}(y,z) = 0$ otherwise, cf. [141, Chap. 1]).

has time complexity t in some "reasonable and general" *model of computation if and only if it has time complexity* $\text{poly}(t)$ *in the model of* (single-tape) *Turing machines.*

Indeed, the Cobham-Edmonds Thesis strengthens the Church-Turing Thesis. It asserts not only that the class of solvable problems is invariant as far as "reasonable and general" models of computation are concerned, but also that the time complexity (of the solvable problems) in such models is polynomially related.

Efficient algorithms. As hinted in the foregoing discussions, much of complexity theory is concerned with efficient algorithms. The latter are defined as polynomial-time algorithms (i.e., algorithms that have a time complexity that is bounded by a polynomial in the length of the input). By the Cobham-Edmonds Thesis, the choice of a "reasonable and general" model of computation is irrelevant to the definition of this class. The association of efficient algorithms with polynomial-time computation is grounded in the following two considerations:

- *Philosophical consideration*: Intuitively, efficient algorithms are those that can be implemented within a number of steps that is a moderately growing function of the input length. To allow for reading the entire input, at least linear time complexity should be allowed, whereas exponential time (as in "exhaustive search") must be avoided. Furthermore, a good definition of the class of efficient algorithms should be closed under natural composition of algorithms (as well as be robust with respect to reasonable models of computation and with respect to simple changes in the encoding of problems' instances).

  Selecting polynomials as the set of time-bounds for efficient algorithms satisfy all the foregoing requirements: polynomials constitute a "closed" set of moderately growing functions, where "closure" means closure under addition, multiplication and functional composition. These closure properties guarantee the closure of the class of efficient algorithm under natural composition of algorithms (as well as its robustness with respect to any reasonable and general model of computation). Furthermore, polynomial-time algorithms can conduct computations that are intuitively simple (although not necessarily trivial), and on the other hand they do not include algorithms that are intuitively inefficient (like exhaustive search).

- *Empirical consideration*: It is clear that algorithms that are considered efficient in practice have running-time that is bounded by a small polynomial (at least on the inputs that occur in practice). The question is whether any polynomial-time algorithm can be considered efficient in an intuitive sense. The belief, which is supported by past experience, is that every natural problem that can be solved in polynomial-time also has "reasonably efficient" algorithms.

We stress that the association of efficient algorithms with polynomial-time computation is not essential to most of the notions, results and questions of complexity theory. Any other class of algorithms that supports the aforementioned closure properties and allows to conduct some simple computations but not overly complex ones gives rise to a similar theory, albeit the formulation of such a theory may be much more complicated. Specifically, all results and questions treated in this book are concerned with the relation among the complexities of different computational tasks (rather than with providing absolute assertions about the complexity of some computational tasks). These relations can be stated explicitly, by stating how any upper-bound on the time complexity of one task gets translated to an upper-bound on the time complexity of another task.[18] Such cumbersome statements will maintain the contents of the standard statements; they will merely be much more complicated. Thus, we follow the tradition of focusing on polynomial-time computations, while stressing that this focus is both natural and provides the simplest way of addressing the fundamental issues underlying the nature of efficient computation.

Universal machines, revisited. The notion of time complexity gives rise to a time-bounded version of the universal function $\mathtt{u}$ (presented in §1.2.3.3). Specifically, we define $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\rm def}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\rm def}{=} \perp$ if on input x machine M makes more than t steps. Unlike $\mathtt{u}$, the function $\mathtt{u}'$ is a total function. Furthermore, unlike any extension of $\mathtt{u}$ to a total function the function $\mathtt{u}'$ is computable. Moreover, $\mathtt{u}'$ is computable by a machine U' that on input $X = (\langle M\rangle, x, t)$ halts after $\mathrm{poly}(t)$ steps. Indeed, machine U' is a variant of a universal machine (i.e., on input X, machine U' merely emulates M for t steps rather than emulating M till it halts (and potentially indefinitely)). Note that the number of steps taken by U' depends on the specific model of computation (and that some overhead is unavoidable because emulating each step of M requires reading the relevant portion of the description of M).

Space complexity. Another natural measure of the "complexity" of an algorithm (or a task) is the amount of memory consumed by the computation. We refer to the memory used for storing some intermediate results of the computation. Since much of our focus will be on using memory that is sub-linear in the input length, it is important to use a model in which one can differentiate memory used for computation from memory used for storing the initial input or the final output. In the context of Turing machines, this is done by considering multi-tape Turing machines such that the input is presented on a special read-only tape (called the input tape), the output is written on a special write-only tape (called the output tape), and intermediate results are stored on a work-tape. Thus, the input and output tapes cannot be used for storing intermediate results. The space complexity

[18]For example, the NP-completeness of SAT (cf. Theorem 2.21) implies that any algorithm solving SAT in time T yields an algorithm that factors composite numbers in time T' such that $T'(n) = \mathrm{poly}(n) \cdot (1 + T(\mathrm{poly}(n)))$. (More generally, if the correctness of solutions for n-bit instances of some search problem can be verified in time $t(n)$ then such solutions can be found in time T' such that $T'(n) = t(n) \cdot (1 + T(O(t(n))^2))$.)

of such a machine M is defined as a function s_M such that $s_M(x)$ is the number of cells of the work-tape that are scanned by M on input x. As in the case of time complexity, we will usually refer to $S_A(n) \stackrel{\rm def}{=} \max_{x\in\{0,1\}^n}\{s_A(x)\}$.

1.2.3.5 Oracle machines

The notion of Turing-reductions, which was discussed in §1.2.3.2, is captured by the following definition of so-called *oracle machines*. Loosely speaking, an oracle machine is a machine that is augmented such that it may pose questions to the outside. (A rigorous formulation of this notion is provided below.) We consider the case in which these questions, called queries, are answered consistently by some function $f : \{0,1\}^* \to \{0,1\}^*$, called the oracle. That is, if the machine makes a query q then the answer it obtains is $f(q)$. In such a case, we say that the oracle machine is given access to the oracle f. For an oracle machine M, a string x and a function f, we denote by $M^f(x)$ the output of M on input x when given access to the oracle f. (Re-examining the second part of the proof of Theorem 1.5, observe that we have actually described an oracle machine that computes d' when given access to the oracle d.)

The notion of an oracle machine extends the notion of a standard computing device (machine), and thus a rigorous formulation of the former extends a formal model of the latter. Specifically, extending the model of Turing machines, we derive the following model of oracle Turing machines.

Definition 1.11 (using an oracle):

- *An* oracle machine *is a Turing machine with an additional tape, called the* oracle tape*, and two special states, called* oracle invocation *and* oracle spoke*.*
- *The* computation of the oracle machine M on input x and access to the oracle $f : \{0,1\}^* \to \{0,1\}^*$ *is defined based on the successive configuration function. For configurations with state different from* oracle invocation *the next configuration is defined as usual. Let γ be a configuration in which the machine's state is* oracle invocation *and suppose that the actual contents of the oracle tape is q (i.e., q is the contents of the maximal prefix of the tape that holds bit values).*[19] *Then, the configuration following γ is identical to γ, except that the state is* oracle spoke*, and the actual contents of the oracle tape is $f(q)$. The string q is called M's* query *and $f(q)$ is called the* oracle's reply*.*
- *The output of M on input x when given oracle access to f is denote $M^f(x)$.*

We stress that the running time of an oracle machine is the number of steps made during its computation, and that the oracle's reply on each query is obtained in a single step.

[19]This fits the definition of the *actual contents of a tape of a Turing machine* (cf. §1.2.3.1). A common convention is that the oracle can be invoked only when the machine's head resides at the left-most cell of the oracle tape. We comment that, in the context of space complexity, one uses two oracle tapes: a write-only tape for the query and a read-only tape for the answer.

1.2.3.6 Restricted models

We mention that restricted models of computation are often mentioned in the context of a course on computability, but they will play no role in the current book. One such model is the model of finite automata, which in some variant coincides with Turing machines that have space-complexity zero (equiv., constant).

In our opinion, the most important motivation for the study of these restricted models of computation is that they provide simple models for some natural (or artificial) phenomena. This motivation, however, seems only remotely related to the study of the complexity of various computational tasks. Thus, in our opinion, the study of these restricted models (e.g., any of the lower levels of Chomsky's Hierarchy [118, Chap. 9]) should be decoupled from the study of computability theory (let alone the study of complexity theory).

Teaching note: Indeed, we reject the common coupling of computability theory with the theory of automata and formal languages. Although the historical links between these two theories (at least in the West) can not be denied, this fact cannot justify coupling two fundamentally different theories (especially when such a coupling promotes a wrong perspective on computability theory).

1.2.4 Non-uniform Models (Circuits and Advice)

By a non-uniform model of computation we mean a model in which for each possible input length one considers a different computing device. That is, there is no "uniformity" requirement relating devices that correspond to different input lengths. Furthermore, this collection of devices is infinite by nature, and (in absence of a uniformity requirement) this collection may not even have a finite description. Nevertheless, each device in the collection has a finite description. In fact, the relationship between the size of the device (resp., the length of its description) and the length of the input that it handles will be of major concern. The hope is that the finiteness of all parameters (which refer to a single device in such a collection) will allow for the application of combinatorial techniques to analyze the limitations of certain settings of parameters.

In complexity theory, non-uniform models of computation are studied either towards the development of lower-bound techniques or as simplified upper-bounds on the ability of efficient algorithms. In both cases, the uniformity condition is eliminated in the interest of simplicity and with the hope (and belief) that nothing substantial is lost as far as the issues at hand are concerned.

We will focus on two related models of non-uniform computing devices: Boolean circuits (§1.2.4.1) and "machines that take advice" (§1.2.4.2). The former model is more adequate for the study of the evolution of computation (i.e., development of lower-bound techniques), whereas the latter is more adequate for modeling purposes (e.g., upper-bounding the ability of efficient algorithms). (These models will be further studied in Sections 3.1 and 4.1.)

1.2.4.1 Boolean Circuits

The most popular model of non-uniform computation is the one of Boolean circuits. Historically, this model was introduced for the purpose of describing the "logic operation" of real-life electronic circuits. Ironically, nowadays this model provides the stage for some of the most practically removed studies in complexity theory (which aim at developing methods that may eventually lead to an understanding of the inherent limitations of efficient algorithms).

A Boolean circuit is a directed acyclic graph[20] *with labels on the vertices*, to be discussed shortly. For sake of simplicity, we disallow isolated vertices (i.e., vertices with no in-going or out-going edges), and thus the graph's vertices are of three types: *sources*, *sinks*, and *internal vertices*.

1. Internal vertices are vertices having in-coming and out-going edges (i.e., they have in-degree and out-degree at least 1). In the context of Boolean circuits, internal vertices are called gates. Each gate is labeled by a Boolean operation, where the operations that are typically considered are $\wedge$, $\vee$ and $\neg$ (corresponding to `and`, `or` and `neg`). In addition, we require that gates labeled $\neg$ have in-degree 1. (The in-coming degree of $\wedge$-gates and $\vee$-gates may be any number greater than zero, and the same holds for the out-degree of any gate.)

2. The graph sources (i.e., vertices with no in-going edges) are called input terminals. Each input terminal is labeled by a natural number (which is to be thought of the index of an input variable). (For sake of defining formulae (see §1.2.4.3), we allow different input terminals to be labeled by the same number.)[21]

3. The graph sinks (i.e., vertices with no out-going edges) are called output terminals, and we require that they have in-degree 1. Each output terminal is labeled by a natural number such that if the circuit has m output terminals then they are labeled $1, 2, ..., m$. That is, we disallow different output terminals to be labeled by the same number, and insist that the labels of the output terminals are consecutive numbers. (Indeed, the labels of the output terminals will correspond to the indices of locations in the circuit's output.)

For sake of simplicity, we also mandate that the labels of the input terminals are consecutive numbers.[22]

[20] See Appendix G.1.

[21] This is not needed in case of general circuits, because we can just feed out-going edges of the same input terminal to many gates. Note, however, that this is not allowed in case of formulae, where all non-sinks are required to have out-degree exactly 1.

[22] This convention slightly complicates the construction of circuits that ignore some of the input values. Specifically, we use artificial gadgets that have in-coming edges from the corresponding input terminals, and compute an adequate constant. To avoid having this constant as an output terminal, we feed it into an auxiliary gate such that the value of the latter is determined by the other in-going edge (e.g., a constant 0 fed into an $\vee$-gate). See example of dealing with x_3 in Figure 1.3.

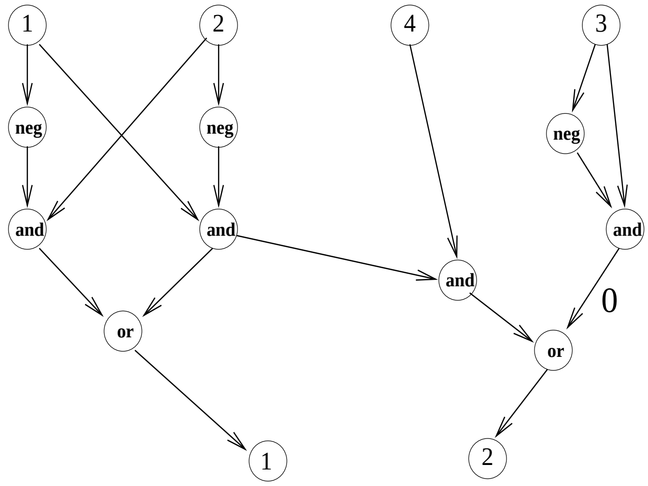


Figure 1.3: A circuit computing $f(x_1, x_2, x_3, x_4) = (x_1 \oplus x_2, x_1 \wedge \neg x_2 \wedge x_4)$.

A Boolean circuit with n different input labels and m output terminals induces (and indeed computes) a function from $\{0,1\}^n$ to $\{0,1\}^m$ defined as follows. For any fixed string $x \in \{0,1\}^n$, we iteratively define the value of vertices in the circuit such that the input terminals are assigned the corresponding bits in $x = x_1 \cdots x_n$ and the values of other vertices are determined in the natural manner. That is:

- An input terminal with label $i \in \{1, ..., n\}$ is assigned the i^{th} bit of x (i.e., the value x_i).
- If the children of a gate (of in-degree d) that is labeled $\wedge$ have values $v_1, v_2, ..., v_d$, then the gate is assigned the value $\wedge_{i=1}^{d} v_i$. The value of a gate labeled $\vee$ (or $\neg$) is determined analogously.

  Indeed, the hypothesis that the circuit is acyclic implies that the process of determining values for the circuit's vertices is well-defined: As long as the value of some vertex is undetermined, there exists a vertex such that its value is undetermined but the values of all its children are determined. Thus, the process can make progress, and terminates when the values of all vertices (including the output terminals) are determined.

The value of the circuit on input x (i.e., the output computed by the circuit on input x) is $y = y_1 \cdots y_m$, where y_i is the value assigned by the foregoing process to the output terminal labeled i. We note that *there exists a polynomial-time algorithm that, given a circuit C and a corresponding input x, outputs the value of C on input x.* This algorithm determines the values of the circuit's vertices, going from the circuit's input terminals to its output terminals.

We say that a family of circuits $(C_n)_{n \in \mathbb{N}}$ computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if for every n the circuit C_n computes the restriction of f to strings of length n. In other words, for every $x \in \{0,1\}^*$, it must hold that $C_{|x|}(x) = f(x)$.

Bounded and unbounded fan-in. We will be most interested in circuits in which each gate has at most two in-coming edges. In this case, the types of (two-argument) Boolean operations that we allow is immaterial (as long as we consider a "full basis" of such operations; i.e., a set of operations that can implement any other two-argument Boolean operation). Such circuits are called circuits of bounded fan-in. In contrast, other studies are concerned with circuits of unbounded fan-in, where each gate may have an arbitrary number of in-going edges. Needless to say, in the case of circuits of unbounded fan-in, the choice of allowed Boolean operations is important and one focuses on operations that are "uniform" (across the number of operants; e.g., $\wedge$ and $\vee$).

Circuit size as a complexity measure. The size of a circuit is the number of its edges. When considering a family of circuits $(C_n)_{n\in\mathbb{N}}$ that computes a function $f : \{0,1\}^* \to \{0,1\}^*$, we are interested in the size of C_n as a function of n. Specifically, we say that this family has size complexity $s : \mathbb{N} \to \mathbb{N}$ if for every n the size of C_n is $s(n)$. The circuit complexity of a function f, denoted s_f, is the infimum of the size complexity of all families of circuits that compute f. Alternatively, for each n we may consider the size of the smallest circuit that computes the restriction of f to n-bit strings (denoted f_n), and set $s_f(n)$ accordingly. We stress that non-uniformity is implicit in this definition, because no conditions are made regarding the relation between the various circuits used to compute the function on different input lengths.

The circuit complexity of functions. We highlight some simple facts about the circuit complexity of functions. (These facts are in clear correspondence to facts regarding Kolmogorov Complexity mentioned in §1.2.3.3.)

1. Most importantly, any Boolean function can be computed by some family of circuits, and thus the circuit complexity of any function is well-defined. Furthermore, each function has at most exponential circuit complexity.

   (Hint: $f_n : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(n2^n)$ that implements a look-up table.)

2. Some functions have polynomial circuit complexity. In particular, any function that has time complexity t (i.e., is computed by an algorithm of time complexity t) has circuit complexity $\text{poly}(t)$. Furthermore, the corresponding circuit family is uniform (in a natural sense to be discussed in the next paragraph).

   (Hint: consider a Turing machine that computes the function, and consider its computation on a generic n-bit long input. The corresponding computation can be emulated by a circuit that consists of $t(n)$ layers such that each layer represents an instantaneous configuration of the machine, and the relation between consecutive configurations is captured by ("uniform") local gadgets in the circuit. For further details see the proof of Theorem 2.20, which presents a similar emulation.)

3. Almost all Boolean functions have exponential circuit complexity. Specifically, the number of functions mapping $\{0,1\}^n$ to $\{0,1\}$ that can be computed by some circuit of size s is at most s^{2s}.

   (Hint: the number of circuits having v vertices and s edges is at most $2^v \cdot \binom{v}{2}^s$.)

Note that the first fact implies that families of circuits can compute functions that are uncomputable by algorithms. Furthermore, this phenomenon occurs also when restricting attention to families of polynomial-size circuits. See further discussion in §1.2.4.2.

Uniform families. A family of polynomial-size circuits $(C_n)_{n\in\mathbb{N}}$ is called uniform if given n one can construct the circuit C_n in poly(n)-time. Note that *if a function is computable by a uniform family of polynomial-size circuits then it is computable by a polynomial-time algorithm.* This algorithm first constructs the adequate circuit (which can be done in polynomial-time by the uniformity hypothesis), and then evaluate this circuit on the given input (which can be done in time that is polynomial in the size of the circuit).

Note that limitations on the computing power of arbitrary families of polynomial-size circuits certainly hold for uniform families (of polynomial-size), which in turn yield limitations on the computing power of polynomial-time algorithms. Thus, lower bounds on the circuit complexity of functions yield analogous lower bounds on their time complexity. Furthermore, as is often the case in mathematics and Science, disposing of an auxiliary condition that is not well-understood (i.e., uniformity) may turn out fruitful. Indeed, this has occured in the study of classes of restricted circuits, which is reviewed in §1.2.4.3 (and Appendix B).

1.2.4.2 Machines that take advice

General (non-uniform) circuit families and uniform circuit families are two extremes with respect to the "amounts of non-uniformity" in the computing device. Intuitively, in the former, non-uniformity is only bounded by the size of the device, whereas in the latter the amounts of non-uniformity is zero. Here we consider a model that allows to decouple the size of the computing device from the amount of non-uniformity, which may range from zero to the device's size. Specifically, we consider algorithms that "take a non-uniform advice" that depends only on the input length. The amount of non-uniformity will be defined to equal the length of the corresponding advice (as a function of the input length).

Definition 1.12 (taking advice): *We say that* algorithm A computes the function f using advice of length $\ell:\mathbb{N}\to\mathbb{N}$ *if there exists an infinite sequence $(a_n)_{n\in\mathbb{N}}$ such that*

1. *For every $x\in\{0,1\}^*$, it holds that $A(a_{|x|},x)=f(x)$.*
2. *For every $n\in\mathbb{N}$, it holds that $|a_n|=\ell(n)$.*

The sequence $(a_n)_{n\in\mathbb{N}}$ is called the advice sequence.

Note that any function having circuit complexity s can be computed using advice of length $O(s \log s)$, where the log factor is due to the fact that a graph with v vertices and e edges can be described by a string of length $2e \log_2 v$. Note that the model of machines that use advice allows for some sharper bounds than the ones stated in §1.2.4.1: every function can be computed using advice of length ℓ such that $\ell(n) = 2^n$, and some uncomputable functions can be computed using advice of length 1.

Theorem 1.13 (the power of advice): *There exist functions that can be computed using one-bit advice but cannot be computed without advice.*

Proof: Starting with any uncomputable Boolean function $f : \mathbb{N} \to \{0,1\}$, consider the function f' defined as $f'(x) = f(|x|)$. Note that f is Turing-reducible to f' (e.g., on input n make any n-bit query to f', and return the answer).[23] Thus, f' cannot be computed without advice. On the other hand, f' can be easily computed by using the advice sequence $(a_n)_{n\in\mathbb{N}}$ such that $a_n = f(n)$; that is, the algorithm merely outputs the advice bit (and indeed $a_{|x|} = f(|x|) = f'(x)$, for every $x \in \{0,1\}^*$). ■

1.2.4.3 Restricted models

As noted in §1.2.4.1, the model of Boolean circuits allows for the introduction of many natural subclasses of computing devices. Following is a laconic review of a few of these subclasses. For more detail, see Appendix B.2. Since we shall refer to various types of Boolean formulae in the rest of this book, we suggest not to skip the following two paragraphs.

Boolean formulae. In general Boolean circuits the non-sink vertices are allowed arbitrary out-degree. This means that the same intermediate value can be re-used (without being re-computed (and while increasing the size complexity by only one unit)). Such "free" re-usage of intermediate values is disallowed in Boolean formulae, which corresponds to a Boolean expression over Boolean variables. Formally, a Boolean formula is a circuit in which all non-sink vertices have out-degree 1, which means that the underlying graph is a tree (see §G.2) and the formula as an expression can be read by traversing the tree (and registering the vertices' labels in the order traversed). Indeed, we have allowed different input terminals to be assigned the same label in order to allow formulae in which the same variable occurs multiple times. As in case of general circuits, one is interested in the size of these restricted circuits (i.e., the size of families of formulae computing various functions). We mention that quadratic lower bounds are known for the formula size of simple functions (e.g., `parity`), whereas these functions have linear circuit complexity. This discrepancy is depicted in Figure 1.4.

[23]Indeed, this Turing-reduction is not efficient (i.e., it runs in exponential time in $|n| = \log_2 n$), but this is immaterial in the current context.

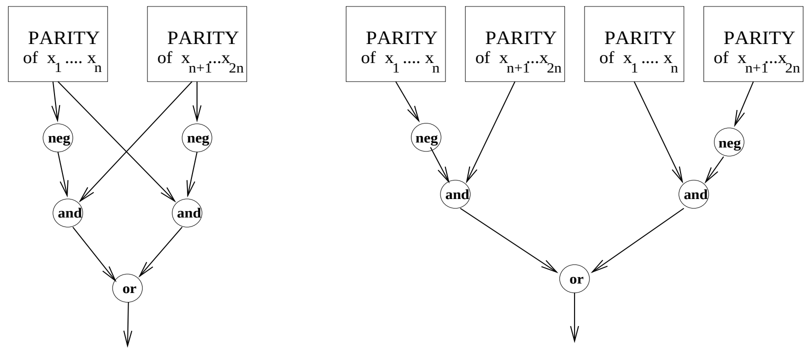


Figure 1.4: Recursive construction of parity circuits and formulae.

Formulae in CNF and DNF. A restricted type of Boolean formulae consists of formulae that are in conjunctive normal form (CNF). Such a formula consists of a conjunction of clauses, where each clause is a disjunction of literals each being either a variable or its negation. That is, such formulae are represented by layered circuits of unbounded fan-in in which the first layer consists of neg-gates that compute the negation of input variables, the second layer consist of or-gates that compute the logical-or of subsets of inputs and negated inputs, and the third layer consists of a single and-gate that computes the logical-and of the values computed in the second layer. Note that each Boolean function can be computed by a family of CNF formulae of exponential size, and that the size of CNF formulae may be exponentially larger than the size of ordinary formulae computing the same function (e.g., parity). For a constant k, a formula is said to be in k-CNF if its CNF has disjunctions of size at most k. An analogous restricted type of Boolean formulae refers to formulae that are in disjunctive normal form (DNF). Such a formula consists of a disjunction of a conjunctions of literals, and when each conjunction has at most k literals we say that the formula is in k-DNF.

Constant-depth circuits. Circuits have a "natural structure" (i.e., their structure as graphs). One natural parameter regarding this structure is the depth of a circuit, which is defined as the longest directed path from any source to any sink. Of special interest are constant-depth circuits of unbounded fan-in. We mention that sub-exponential lower bounds are known for the size of such circuits that compute a simple function (e.g., parity).

Monotone circuits. The circuit model also allows for the consideration of monotone computing devices: a monotone circuit is one having only monotone gates (e.g., gates computing $\wedge$ and $\vee$, but no negation gates (i.e., $\neg$-gates)). Needless to say, monotone circuits can only compute monotone functions, where a function $f : \{0,1\}^n \to \{0,1\}$ is called monotone if for any $x \preceq y$ it holds that $f(x) \leq f(y)$ (where $x_1 \cdots x_n \preceq y_1 \cdots y_n$ if and only if for every bit position i it holds that $x_i \leq y_i$). One natural question is whether, as far as monotone functions are concerned, there is a substantial loss in using only monotone circuits. The answer is *yes*: there exist monotone functions that have polynomial circuit complexity but require sub-exponential size monotone circuits.

1.2.5 Complexity Classes

Complexity classes are sets of computational problems. Typically, such classes are defined by fixing three parameters:

1. A type of computational problems (see Section 1.2.2). Indeed, most classes refer to decision problems, but classes of search problems, promise problems, and other types of problems will also be considered.

2. A model of computation, which may be either uniform (see Section 1.2.3) or non-uniform (see Section 1.2.4).

3. A complexity measure and a function (or a set of functions), which put together limit the class of computations of the previous item; that is, we refer to the class of computations that have complexity not exceeding the specified function (or set of functions). For example, in §1.2.3.4, we mentioned time complexity and space complexity, which apply to any uniform model of computation. We also mentioned polynomial-time computations, which are computations in which the time complexity (as a function) does not exceed some polynomial (i.e., a member of the set of polynomial functions).

The most common complexity classes refer to decision problems, and are sometimes defined as classes of sets rather than classes of the corresponding decision problems. That is, one often says that a set $S \subseteq \{0,1\}^*$ is in the class $\mathcal{C}$ rather than saying that *the problem of deciding membership in* S is in the class $\mathcal{C}$. Likewise, one talks of classes of relations rather than classes of the corresponding search problems (i.e., saying that $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is in the class $\mathcal{C}$ means that *the search problem of* R is in the class $\mathcal{C}$).

Chapter Notes

It is quite remarkable that the theories of uniform and non-uniform computational devices have emerged in two single papers. We refer to Turing's paper [214], which introduced the model of Turing machines, and to Shannon's paper [192], which introduced Boolean circuits.

In addition to introducing the Turing machine model and arguing that it corresponds to the intuitive notion of computability, Turing's paper [214] introduces universal machines and contains proofs of undecidability (e.g., of the Halting Problem).

The Church-Turing Thesis is attributed to the works of Church [53] and Turing [214]. In both works, this thesis is invoked for claiming that the fact that Turing machines cannot solve some problem implies that this problem cannot be solved in any "reasonable" model of computation. The RAM model is attributed to von Neumann's report [221].

The association of efficient computation with polynomial-time algorithms is attributed to the papers of Cobham [54] and Edmonds [66]. It is interesting to note that Cobham's starting point was his desire to present a philosophically sound concept of efficient algorithms, whereas Edmonds's starting point was his desire to articulate why certain algorithms are "good" in practice.

Rice's Theorem is proven in [183], and the undecidability of the Post Correspondence Problem is proven in [172]. The formulation of machines that take advice (as well as the equivalence to the circuit model) originates in [131].