Computational Complexity:

A Conceptual Perspective

Oded Goldreich

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

December 9, 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 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 computational 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 complexity and in proof complexity falls within this category. In contrast, a

[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).

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 to serve as an introduction to Computational Complexity that can be used either as a textbook or for self-study. Indeed, the book's primary target audience consists of students that wish to learn complexity theory and educators that intend to teach a course on complexity theory. The book is also intended to promote interest in complexity theory and make it acccessible 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., $\mathcal{BPP} \subset \mathcal{P}/\text{poly}$) 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 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 hardness to worst-case hardness.

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: $\mathcal{P}$, $\mathcal{NP}$, co$\mathcal{NP}$, $\mathcal{BPP}$, $\mathcal{RP}$, co$\mathcal{RP}$, $\mathcal{ZPP}$, $\#\mathcal{P}$, $\mathcal{PH}$, $\mathcal{E}$, $\mathcal{EXP}$, $\mathcal{NEXP}$, $\mathcal{L}$, $\mathcal{NL}$, $\mathcal{RL}$, $\mathcal{PSPACE}$, $\mathcal{P}$/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 $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$.

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 [86, Chap. 3] and [86, 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 [86, Chap. 1] and books [87, 88]; but, again, the previous texts are very different in many ways. In contrast, Appendix B was adapted with relatively little modifications from an early draft of a section of an article by Avi Wigderson and myself [103].

[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).

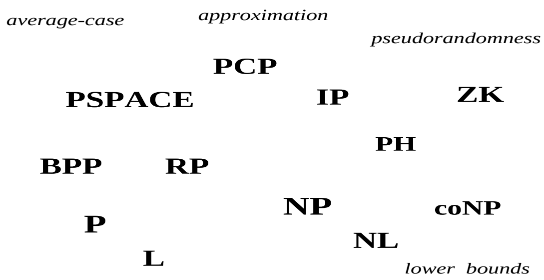
Chapter 1

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 (e.g., a specific algorithmic schema). 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 is intended to serve as an introduction to Computational Complexity that can be used either as a textbook or for self-study. It 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 speaking, the 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 field's questions and answers. 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 so 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.

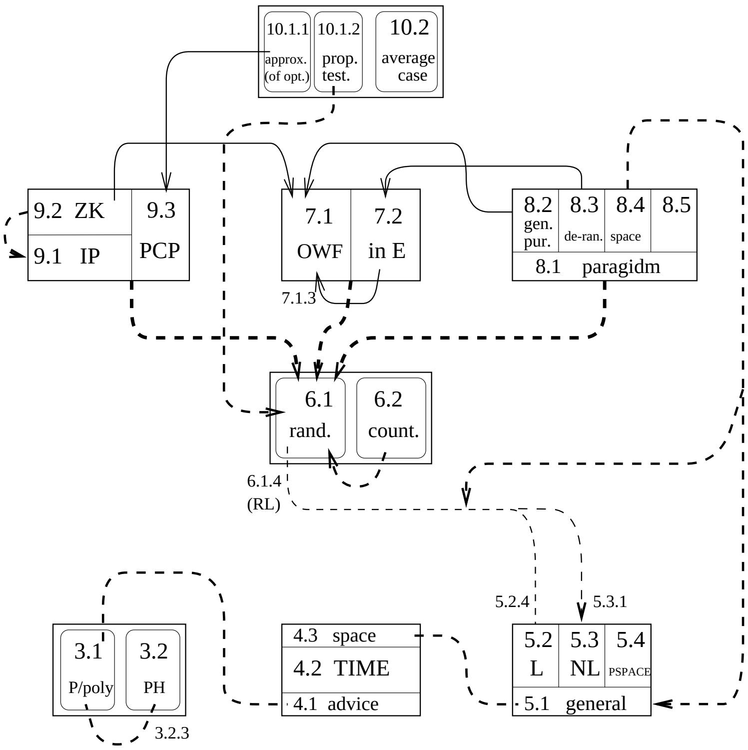
The rest of this section provides a brief summary of the contents of the various chapters and appendices. This summary is intended for the teacher and/or the expert, whereas the student is referred to the more reader-friendly summaries that appear in the book's prefix.

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.



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.

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 $\mathcal{NL} = \mathrm{co}\mathcal{NL}$, 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 $\mathcal{BPP} \subset \mathcal{P}/\mathrm{poly}$ 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 §1.1.4.4, 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, 225]) 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. [226]), *parallel computation* [134], and *communication complexity* [142]. We mention that the recent textbook of Arora and Barak [13] contains a treatment of all these topics. Finally, we mention 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* [136].

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, whereas explanations and motivations are merely 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 strongly 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 science 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 personality 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" (e.g., NP-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 rigorous 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 details

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. In some cases, exercises (augmented by adequate guidelines) are used for presenting additional related material.

The book contains material that ranges from topics that are currently taught in undergraduate courses on computability (and basic complexity) to topics that are currently taught mostly in advanced graduate courses. Although this situation may (and hopefully will) change in the future, we believe that it will remain to be the case that typical readers of the advanced chapters will be more sophisticated than typical readers of the basic chapters (i.e., Section 1.2 and Chapter 2). Accordingly, the style of presentation becomes more sophisticated as one progresses from Chapter 2 to later chapters.

As stated in the preface, this book focuses on the high-level approach to complexity theory, whereas the low-level approach (i.e., lower bounds) is only 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] Further details on the contents of the various chapters and appendices are provided in Section 1.1.3.

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 papers.

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.

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

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.

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., $x = x_1 x_2 \cdots x_{|x|}$). 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 $x_1 x_1 \cdots x_m x_m 01 y_1 \cdots y_n \in \{0,1\}^*$). 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...), 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., [200]) 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

[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.

[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.

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]

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.

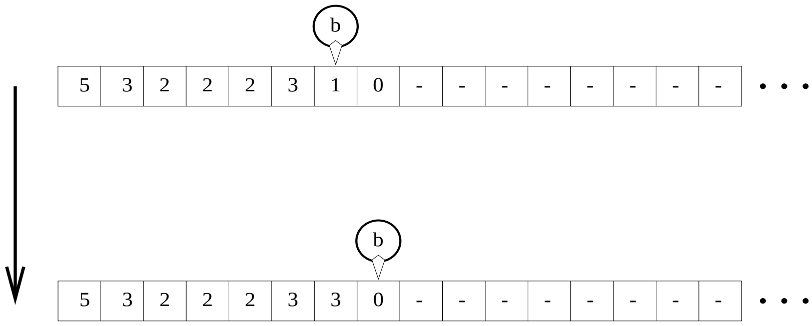


Figure 1.2: A single step by a Turing machine.

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

[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.

[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.

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.

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

[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.

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.

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 $\mathtt{d}(\langle M\rangle) \stackrel{\rm def}{=} \mathtt{h}(\langle M\rangle, \langle M\rangle))$ 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

[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.

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.*

Again, the omitted proof is by a Turing-reduction from `d` (or `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 `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 `u` is computable. In contrast, it can be shown that any extension of `u` to a total function is uncomputable. That is, for

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

[14] We mention that the reduction maps an instance $(\langle M \rangle, x)$ of `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.

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 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

[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 $x = \langle M\rangle$).

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

[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 [142, Sec. 12.2]. Intuitively, a single-tape Turing machine that decides

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 has time complexity t in some* "reasonable and general" *model of computation if and only if it has time complexity* $\mathrm{poly}(t)$ *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).

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. [142, Chap. 1]).

- *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 $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} y$ if on input x machine M halts within t steps and outputs the string y, and $\mathtt{u}'(\langle M\rangle, x, t) \stackrel{\text{def}}{=} \perp$ 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

[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 $T'(n) = t(n) \cdot (1 + T(O(t(n))^2))$.)

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 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*

[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 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.

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 [119, 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

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.

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

[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.

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]

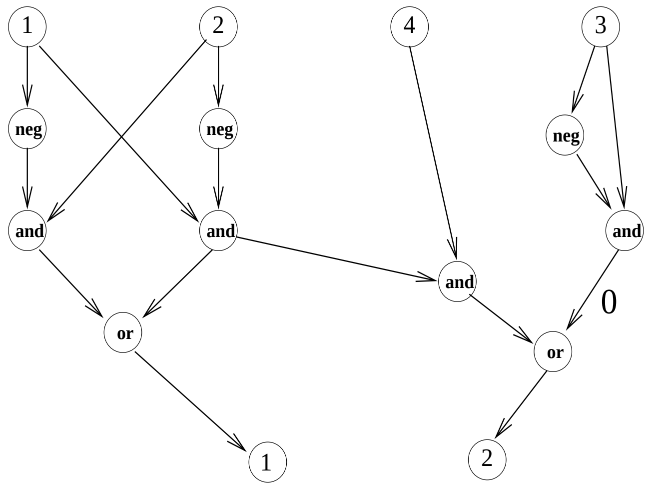


Figure 1.3: A circuit computing $f(x_1,x_2,x_3,x_4) = (x_1 \oplus x_2, x_1 \wedge \neg x_2 \wedge x_4)$.

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.

[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.

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 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,

[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.

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.

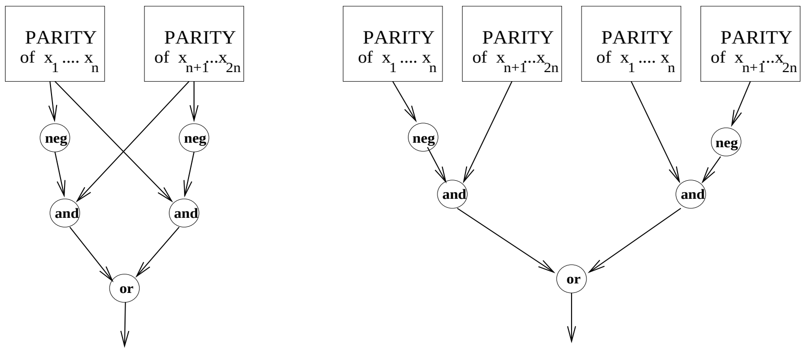


Figure 1.4: Recursive construction of parity circuits and formulae.

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 [216], which introduced the model of Turing machines, and to Shannon's paper [194], 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 [216] 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 [52] and Turing [216]. 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 [223].

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 [185], and the undecidability of the Post Correspondence Problem is proven in [174]. The formulation of machines that take advice (as well as the equivalence to the circuit model) originates in [132].

Chapter 10

Relaxing the Requirements

The philosophers have only interpreted the world, in various ways; the point is to change it.

Karl Marx, Theses on Feuerbach

In light of the apparent infeasibility of solving numerous useful computational problems, it is natural to ask whether these problems can be relaxed such that the relaxation is both useful and allows for feasible solving procedures. We stress two aspects about the foregoing question: on one hand, the relaxation should be sufficiently good for the intended applications; but, on the other hand, it should be significantly different from the original formulation of the problem so to escape the infeasibility of the latter. We note that whether a relaxation is adequate for an intended application depends on the application, and thus much of the material in this chapter is less robust (or generic) than the treatment of the non-relaxed computational problems.

Summary: We consider two types of relaxations. The first type of relaxation refers 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 (with respect to some value function). Needless to say, the specific meaning of 'sufficiently close' is part of the definition of the relaxed problem. In the context of decision problems this means that for some instances both answers are considered valid; put differently, we consider promise problems in which the no-instances are "far" from the yes-instances in some adequate sense (which is part of the definition of the relaxed problem).

The second type of relaxation deviates from the requirement that the solver provides an adequate answer on each valid instance. Instead, the behavior of the solver is analyzed with respect to a predetermined

input distribution (or a class of such distributions), and bad behavior may occur with negligible probability where the probability is taken over this input distribution. That is, we replace worst-case analysis by *average-case* (or rather *typical-case*) *analysis*. Needless to say, a major component in this approach 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 the corresponding notion of average-case hardness to the standard notion of worst-case hardness.

10.1 Approximation

The notion of approximation is a natural one, and has arisen also in other disciplines. Approximation is most commonly used in references to quantities (e.g., "the length of one meter is approximately forty inches"), but it is also used when referring to qualities (e.g., "an approximately correct account of a historical event"). In the context of computation, the notion of approximation modifies computational tasks such as search and decision problems. (In fact, we have already encountered it as a modifier of counting problems; see Section 6.2.2.)

Two major questions regarding approximation are (1) what is a "good" approximation, and (2) can it be found easier than finding an exact solution. The answer to the first question seems intimately related to the specific computational task at hand and to its role in the wider context (i.e., the higher level application): a good approximation is one that suffices for the intended application. Indeed, the importance of certain approximation problems is much more subjective than the importance of the corresponding optimization problems. This fact seems to stand in the way of attempts at providing a *comprehensive* theory of *natural* approximation problems (e.g., general classes of natural approximation problems that are shown to be computationally equivalent).

Turning to the second question, we note that in numerous cases natural approximation problems seem to be significantly easier than the corresponding original ("exact") problems. On the other hand, in numerous other cases, natural approximation problems are computationally equivalent to the original problems. We shall exemplify both cases by reviewing some specific results, but regret not being able to provide a general systematic classification.[1]

Mimicking the two standard uses of the word *approximation*, we shall distinguish between approximation problems that are of the "search type" and problems that have a clear "decisional" flavor. In the first case we shall refer to a function that assigns values to possible solutions (of a search problem); whereas in the second case we shall refer to distances between instances (of a decision problem). We note that, in some cases, the same computational problem may be cast in both ways, but for most natural approximation problems one of the two frameworks is more appealing than the other. The common theme is that in both cases we extend the set of admissible solutions. In the case of search problems, we extend the set of

[1] A systematic classification of a restricted class of approximation problems, which refer to Constraint Satisfaction Problems, has appeared in [53].

optimal solutions by including also almost-optimal solutions. In the case of decision problems, we extend the set of solutions by allowing an arbitrary answer (solution) to some instances, which may be viewed as a promise problem that disallows these instances. In this case we focus on promise problems in which the yes- and no-instances are far apart (and the instances that violate the promise are closed to yes-instances).

Teaching note: Most of the results presented in this section refer to specific computational problems and (with one exception) are presented without a proof. In view of the complexity of the corresponding proofs and the merely illustrative role of these results in the context of complexity theory, we recommend doing the same in class.

10.1.1 Search or Optimization

As noted in Section 2.2.2, many search problems involve a set of potential solutions (per each problem instance) such that different solutions are assigned different "values" (resp., "costs") by some "value" (resp., "cost") function. In such a case, one is interested in finding a solution of maximum value (resp., minimum cost). A corresponding approximation problem may refer to finding a solution of approximately maximum value (resp., approximately minimum cost), where the specification of the desired level of approximation is part of the problem's definition. Let us elaborate.

For concreteness, we focus on the case of a value that we wish to maximize. For greater flexibility, we allow the value of the solution to depend also on the instance itself. Thus, for a (polynomially bounded) binary relation R and a *value function* $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$, we consider the problem of finding solutions (with respect to R) that maximize the value of f. That is, given x (such that $R(x)\neq\emptyset$), the task is finding $y\in R(x)$ such that $f(x,y)=v_x$, where v_x is the maximum value of $f(x,y')$ over all $y'\in R(x)$. Typically, R is in $\mathcal{PC}$ and f is polynomial-time computable.[2] Indeed, without loss of generality, we may assume that for every x it holds that $R(x)=\{0,1\}^{\ell(|x|)}$ for some polynomial ℓ (see Exercise 2.8). Thus, the optimization problem is recast as the following search problem: *given x, find y such that $f(x,y)=v_x$, where $v_x=\max_{y'\in\{0,1\}^{\ell(|x|)}}\{f(x,y')\}$.*

We shall focus on *relative* approximation problems, where for some gap function $g:\{0,1\}^*\to\{r\in\mathbb{R}:r\geq 1\}$ the (maximization) task is finding y such that $f(x,y)\geq v_x/g(x)$. Indeed, in some cases the approximation factor is stated as a function of the length of the input (i.e., $g(x)=g'(|x|)$ for some $g':\mathbb{N}\to\{r\in\mathbb{R}:r\geq 1\}$), but often the approximation factor is stated in terms of some more refined parameter of the input (e.g., as a function of the number of vertices in a graph). Typically, g is polynomial-time computable.

Definition 10.1 (g-factor approximation): *Let $f:\{0,1\}^*\times\{0,1\}^*\to\mathbb{R}$, $\ell:\mathbb{N}\to\mathbb{N}$, and $g:\{0,1\}^*\to\{r\in\mathbb{R}:r\geq 1\}$.*

[2] In this case, we may assume without loss of generality that the function f depends only on the solution. This can be obtained by redefining the relation R such that each solution $y\in R(x)$ consists of a pair of the form (x,y'). Needless to say, this modification cannot be applied along with getting rid of R (as in Exercise 2.8).

Maximization version: *The* g-factor approximation of maximizing f (w.r.t ℓ) *is the search problem* R *such that* $R(x) = \{y \in \{0,1\}^{\ell(|x|)} : f(x,y) \geq v_x/g(x)\}$, *where* $v_x = \max_{y' \in \{0,1\}^{\ell(|x|)}}\{f(x,y')\}$.

Minimization version: *The* g-factor approximation of minimizing f (w.r.t ℓ) *is the search problem* R *such that* $R(x) = \{y \in \{0,1\}^{\ell(|x|)} : f(x,y) \leq g(x) \cdot c_x\}$, *where* $c_x = \min_{y' \in \{0,1\}^{\ell(|x|)}}\{f(x,y')\}$.

We note that for numerous NP-complete optimization problems, polynomial-time algorithms provide meaningful approximations. A few examples will be mentioned in §10.1.1.1. In contrast, for numerous other NP-complete optimization problems, natural approximation problems are computationally equivalent to the corresponding optimization problem. A few examples will be mentioned in §10.1.1.2, where we also introduce the notion of a *gap problem*, which is a promise problem (of the decision type) intended to capture the difficulty of the (approximate) search problem.

10.1.1.1 A few positive examples

Let us start with a trivial example. Considering a problem such as finding the maximum clique in a graph, we note that finding a linear factor approximation is trivial (i.e., given a graph $G = (V,E)$, we may output any vertex in V as a $|V|$-factor approximation of the maximum clique in G). A famous non-trivial example is presented next.

Proposition 10.2 (factor two approximation to `minimum Vertex Cover`): *There exists a polynomial-time approximation algorithm that given a graph* $G = (V,E)$ *outputs a vertex cover that is at most twice as large as the minimum vertex cover of* G.

We warn that an approximation algorithm for `minimum Vertex Cover` does not yield such an algorithm for the complementary problem (of `maximum Independent Set`). This phenomenon stands in contrast to the case of optimization, where an optimal solution for one problem (e.g., `minimum Vertex Cover`) yields an optimal solution for the complementary problem (`maximum Independent Set`).

Proof Sketch: The main observation is a connection between the set of maximal matchings and the set of vertex covers in a graph. Let M be any *maximal* matching in the graph $G = (V,E)$; that is, $M \subseteq E$ is a matching but augmenting it by any single edge yields a set that is not a matching. Then, on one hand, the set of all vertices participating in M is a vertex cover of G, and, on the other hand, each vertex cover of G must contain at least one vertex of each edge of M. Thus, we can find the desired vertex cover by finding a maximal matching, which in turn can be found by a greedy algorithm. $\square$

Another example. An instance of the traveling salesman problem (TSP) consists of a symmetric matrix of distances between pairs of points, and the task is finding a shortest tour that passes through all points. In general, no reasonable approximation is feasible for this problem (see Exercise 10.1), but here we consider two special cases in which the distances satisfy some natural constraints (and pretty good approximations are feasible).

Theorem 10.3 (approximations to special cases of TSP): *Polynomial-time algorithms exist for the following two computational problems.*

1. *Providing a 1.5-factor approximation for the special case of TSP in which the distances satisfy the triangle inequality.*

2. *For every $\varepsilon > 1$, providing a $(1+\varepsilon)$-factor approximation for the special case of Euclidean TSP* (i.e., for some constant k (e.g., $k = 2$), the points reside in a k-dimensional Euclidean space, and the distances refer to the standard Euclidean norm).

A weaker version of Part 1 is given in Exercise 10.2. A detailed survey of Part 2 is provided in [12]. We note the difference examplified by the two items of Theorem 10.3: Whereas Part 1 provides a polynomial-time approximation for a specific constant factor, Part 2 provides such an algorithm for any constant factor. Such a result is called a *polynomial-time approximation scheme* (abbreviated PTAS).

10.1.1.2 A few negative examples

Let us start again with a trivial example. Considering a problem such as finding the maximum clique in a graph, we note that given a graph $G = (V, E)$ finding a $(1 + |V|^{-1})$-factor approximation of the maximum clique in G is as hard as finding a maximum clique in G. Indeed, this "result" is not really meaningful. In contrast, building on the PCP Theorem (Theorem 9.16), one may prove that finding a $|V|^{1-o(1)}$-factor approximation of the maximum clique in G is as hard as finding a maximum clique in G. This follows from the fact that the approximation problem is NP-hard (cf. Theorem 10.5).

The statement of inapproximability results is made stronger by referring to a promise problem that consists of distinguishing instances of sufficiently far apart values. Such promise problems are called gap problems, and are typically stated with respect to two bounding functions $g_1, g_2 : \{0,1\}^* \to \mathbb{R}$ (which replace the gap function g of Definition 10.1). Typically, g_1 and g_2 are polynomial-time computable.

Definition 10.4 (gap problem for approximation of f): *Let f be as in Definition 10.1 and $g_1, g_2 : \{0,1\}^* \to \mathbb{R}$.*

Maximization version: *For $g_1 \geq g_2$, the* gap_{g_1,g_2} problem of maximizing f *consists of distinguishing between $\{x : v_x \geq g_1(x)\}$ and $\{x : v_x < g_2(x)\}$, where $v_x = \max_{y \in \{0,1\}^{\ell(|x|)}}\{f(x,y)\}$.*

Minimization version: *For* $g_1 \le g_2$*, the* gap_{g_1,g_2} problem of minimizing f *consists of distinguishing between* $\{x : c_x \le g_1(x)\}$ *and* $\{x : c_x > g_2(x)\}$*, where* $c_x = \min_{y \in \{0,1\}^{\ell(|x|)}} \{f(x,y)\}$.

For example, the gap_{g_1,g_2} problem of maximizing the size of a clique in a graph consists of distinguishing between graphs G that have a clique of size $g_1(G)$ and graphs G that have no clique of size $g_2(G)$. In this case, we typically let $g_i(G)$ be a function of the number of vertices in $G = (V,E)$; that is, $g_i(G) = g_i'(|V|)$. Indeed, letting $\omega(G)$ denote the size of the largest clique in the graph G, we let $\texttt{gapClique}_{L,s}$ denote the gap problem of distinguishing between $\{G = (V,E) : \omega(G) \ge L(|V|)\}$ and $\{G = (V,E) : \omega(G) < s(|V|)\}$, where $L \ge s$. Using this terminology, we restate (and strengthen) the aforementioned $|V|^{1-o(1)}$-factor inapproximation of the maximum clique problem.

Theorem 10.5 *For some* $L(N) = N^{1-o(1)}$ *and* $s(N) = N^{o(1)}$*, it holds that* $\texttt{gapClique}_{L,s}$ *is NP-hard.*

The proof of Theorem 10.5 is based on a major refinement of Theorem 9.16 that refers to a PCP system of amortized free-bit complexity that tends to zero (cf. §9.3.4.1). A weaker result, which follows from Theorem 9.16 itself, is presented in Exercise 10.3.

As we shall show next, results of the type of Theorem 10.5 imply the hardness of a corresponding approximation problem; that is, the hardness of deciding a gap problem implies the hardness of a search problem that refers to an analogous factor of approximation.

Proposition 10.6 *Let* f, g_1, g_2 *be as in Definition 10.4 and suppose that these functions are polynomial-time computable. Then the* gap_{g_1,g_2} *problem of maximizing* f (resp., minimizing f) *is reducible to the* g_1/g_2*-factor* (resp., g_2/g_1-factor) *approximation of maximizing* f (resp., minimizing f).

Note that a reduction in the opposite direction does not necessarily exist (even in the case that the underlying optimization problem is self-reducible in some natural sense). Indeed, this is another difference between the current context (of approximation) and the context of optimization problems, where the search problem is reducible to a related decision problem.

Proof Sketch: We focus on the maximization version. On input x, we solve the gap_{g_1,g_2} problem, by making the query x, obtaining the answer y, and ruling that x has value exceeding $g_1(x)$ if and only if $f(x,y) \ge g_2(x)$. Recall that we need to analyze this reduction only on inputs that satisfy the promise. Thus, if $v_x \ge g_1(x)$ then the oracle must return a solution y that satisfies $f(x,y) \ge v_x/(g_1(x)/g_2(x))$, which implies that $f(x,y) \ge g_2(x)$. On the other hand, if $v_x < g_2(x)$ then $f(x,y) \le v_x < g_2(x)$ holds for any possible solution y. $\square$

Additional examples. Let us consider $\texttt{gapVC}_{s,L}$, the gap_{g_s,g_L} problem of minimizing the vertex cover of a graph, where s and L are constants and $g_s(G) = s \cdot |V|$ (resp., $g_L(G) = L \cdot |V|$) for any graph $G=(V,E)$. Then, Proposition 10.2 implies (via Proposition 10.6) that, for every constant s, the problem $\texttt{gapVC}_{s,2s}$ is solvable in polynomial-time. In contrast, sufficiently narrowing the gap between the two thresholds yields an inapproximability result. In particular:

Theorem 10.7 *For some constants* $0 < s < L < 1$ (e.g., $s = 0.62$ and $L = 0.84$ will do)*, the problem* $\texttt{gapVC}_{s,L}$ *is NP-hard.*

The proof of Theorem 10.7 is based on a complicated refinement of Theorem 9.16. Again, a weaker result follows from Theorem 9.16 itself (see Exercise 10.4).

As noted, refinements of the PCP Theorem (Theorem 9.16) play a key role in establishing inapproximability results such as Theorems 10.5 and 10.7. In that respect, it is adequate to recall that Theorem 9.21 establishes the equivalence of the PCP Theorem itself and the NP-hardness of a gap problem concerning the maximization of the number of clauses that are satisfies in a given 3-CNF formula. Specifically, $\texttt{gapSAT}^3_\varepsilon$ was defined (in Definition 9.20) as the gap problem consisting of distinguishing between satisfiable 3-CNF formulae and 3-CNF formulae for which each truth assignment violates at least an ε fraction of the clauses. Although Theorem 9.21 does not specify the quantitative relation that underlies its qualitative assertion, when (refined and) combined with the best known PCP construction, it does yield the best possible bound.

Theorem 10.8 *For every* $v < 1/8$*, the problem* $\texttt{gapSAT}^3_v$ *is NP-hard.*

On the other hand, $\texttt{gapSAT}^3_{1/8}$ is solvable in polynomial-time.

Sharp thresholds. The aforementioned opposite results (regarding $\texttt{gapSAT}^3_v$) exemplify a sharp threshold on the (factor of) approximation that can be obtained by an efficient algorithm. Another appealing example refers to the following maximization problem in which the instances are systems of linear equations over GF(2) and the task is finding an assignment that satisfies as many equations as possible. Note that by merely selecting an assignment at random, we expect to satisfy half of the equations. Also note that it is easy to determine whether there exists an assignment that satisfies all equations. Let $\texttt{gapLin}_{L,s}$ denote the problem of distinguishing between systems in which one can satisfy at least an L fraction of the equations and systems in which one cannot satisfy an s fraction (or more) of the equations. Then, as just noted, $\texttt{gapLin}_{L,0.5}$ is trivial and $\texttt{gapLin}_{1,s}$ is feasible (for every $s < 1$). In contrast, moving both thresholds (slightly) away from the corresponding extremes, yields an NP-hard gap problem:

Theorem 10.9 *For every constant* $\varepsilon > 0$*, the problem* $\texttt{gapLin}_{1-\varepsilon,0.5+\varepsilon}$ *is NP-hard.*

The proof of Theorem 10.9 is based on a major refinement of Theorem 9.16. In fact, the corresponding PCP system (for NP) is merely a reformulation of Theorem 10.9: the verifier makes three queries and tests a linear condition regarding the answers, while using a logarithmic number of coin tosses. This verifier accepts any yes-instance with probability at least $1-\varepsilon$ (when given oracle access to a suitable proof), and rejects any no-instance with probability at least $0.5-\varepsilon$ (regardless of the oracle being accessed). A weaker result, which follows from Theorem 9.16 itself, is presented in Exercise 10.5.

Gap location. Theorems 10.8 and 10.9 illustrate two opposite situations with respect to the "location" of the "gap" for which the corresponding promise problem is hard. Recall that both `gapSAT` and `gapLin` are formulated with respect to two thresholds, where each threshold bounds the fraction of "local" conditions (i.e., clauses or equations) that are satisfiable in the case of yes- and no-instances, respectively. In the case of `gapSAT`, the high threshold (referring to yes-instances) was set to 1, and thus only the low threshold (referring to no-instances) remained a free parameter. Nevertheless, a hardness result was established for `gapSAT`, and furthermore this was achieved for an optimal value of the low threshold (cf. the foregoing discussion of sharp thresholds). In contrast, in the case of `gapLin`, setting the high threshold to 1 makes the gap problem efficiently solvable. Thus, the hardness of `gapLin` was established at a different location of the high threshold. Specifically, hardness (for an optimal value of the ratio of thresholds) was established when setting the high threshold to $1-\varepsilon$, for any $\varepsilon>0$.

A final comment. All the aforementioned inapproximability results refer to approximation (resp., gap) problems that are relaxations of optimization problems in NP (i.e., the optimization problem is computationally equivalent to a decision problem in $\mathcal{NP}$; see Section 2.2.2). In these cases, the NP-hardness of the approximation (resp., gap) problem implies that the corresponding optimization problem is reducible to the approximation (resp., gap) problem. In other words, in these cases nothing is gained by relaxing the original optimization problem, because the relaxed version remains just as hard.

10.1.2 Decision or Property Testing

A natural notion of relaxation for decision problems arises when considering the distance between instances, where a natural notion of distance is the Hamming distance (i.e., the fraction of bits on which two strings disagree). Loosely speaking, this relaxation (called *property testing*) refers to distinguishing inputs that reside in a predetermined set S from inputs that are "relatively far" from any input that resides in the set. Two natural types of promise problems emerge (with respect to any predetermined set S (and the Hamming distance between strings)):

1. *Relaxed decision w.r.t a fixed distance*: Fixing a distance parameter δ, we consider the problem of distinguishing inputs in S from inputs in $\Gamma_\delta(S)$, where
$$\Gamma_\delta(S) \stackrel{\rm def}{=} \{x : \forall z\in S\cap\{0,1\}^{|x|}\ \ \Delta(x,z)>\delta\cdot|x|\} \tag{10.1}$$
and $\Delta(x_1\cdots x_m, z_1\cdots z_m) = |\{i : x_i\neq z_i\}|$ denotes the number of bits on which $x=x_1\cdots x_m$ and $z=z_1\cdots z_m$ disagree. Thus, here we consider a

promise problem that is a restriction (or a special case) of the problem of deciding membership in S.

2. *Relaxed decision w.r.t a variable distance*: Here the instances are pairs (x, δ), where x is as in Type 1 and $\delta \in [0,1]$ is a distance parameter. The yes-instances are pairs (x, δ) such that $x \in S$, whereas (x, δ) is a no-instance if $x \in \Gamma_\delta(S)$.

We shall focus on Type 1 formulation, which seems to capture the essential question of whether or not these relaxations lower the complexity of the original decision problem. The study of Type 2 formulation refers to a relatively secondary question, which assumes a positive answer to the first question; that is, assuming that the relaxed form is easier than the original form, we ask how is the complexity of the problem affected by making the distance parameter smaller (which means making the relaxed problem "tighter" and ultimately equivalent to the original problem).

We note that for numerous NP-complete problems there exist natural (Type 1) relaxations that are solvable in polynomial-time. Actually, these algorithms run in *sub-linear* time (specifically, polylogarithmic time), when given direct access to the input. A few examples will be presented in §10.1.2.2. As indicated in §10.1.2.2, this is not a generic phenomenon. But before turning to these results, we discuss several important definitional issues.

10.1.2.1 Definitional issues

Property testing is concerned not only with solving relaxed versions of NP-hard problems, but rather solving these problems (as well as problems in $\mathcal{P}$) in *sub-linear time*. Needless to say, such results assume a model of computation in which algorithms have direct access to bits in the (representation of the) input (see Definition 10.10).

Definition 10.10 (a direct access model – conventions): *An algorithm with* direct access to its input *is given its* main input *on a special input device that is accessed as an oracle* (see §1.2.3.5). *In addition, the algorithm is given the length of the input and possibly other parameters on a* secondary input device. *The complexity of such an algorithm is stated in terms of the length of its main input.*

Indeed, the description in §5.2.4.2 refers to such a model, but there the main input is viewed as an oracle and the secondary input is viewed as the input. In this model, polylogarithmic time means time that is polylogarithmic in the length of the main input, which means time that is polynomial in the length of the binary representation of the length of the main input. Thus, polylogarithmic time yields a robust notion of extremely efficient computations.

Definition 10.11 (property testing for S): *For any fixed $\delta > 0$, the promise problem of distinguishing S from $\Gamma_\delta(S)$ is called* property testing for S (with respect to δ).

Recall that we say that a randomized algorithm solves a promise problem if it accepts every yes-instance (resp., rejects every no-instance) with probability at least 2/3. Thus, a (randomized) property testing for S accepts every input in S (resp., rejects every input in $\Gamma_\delta(S)$) with probability at least 2/3.

The question of representation. The specific representation of the input is of major concern in the current context. This is due to (1) the *effect of the representation on the distance measure* and to (2) the *dependence of direct access machines on the specific representation of the input.* Let us elaborate on both aspects.

1. Recall that we defined the distance between objects in terms of the Hamming distance between their representations. Clearly, in such a case, the choice of representation is crucial and different representations may yield different distance measures. Furthermore, in this case, the distance between objects is not preserved under various (natural) representations that are considered "equivalent" in standard studies of computational complexity. For example, in previous parts of this book, when referring to computational problems concerning graphs, we did not care whether the graphs were represented by their adjacency matrix or by their incidence-lists. In contrast, these two representations induce very different distance measures and correspondingly different property testing problems (see §10.1.2.2). Likewise, the use of padding (and other trivial syntactic conventions) becomes problematic (e.g., when using a significant amount of padding, all objects are deemed close to one another (and property testing for any set becomes trivial)).

2. Since our focus is on sub-linear time algorithms, we may not afford transforming the input from one natural format to another. Thus, representations that are considered equivalent with respect to polynomial-time algorithms, may not be equivalent with respect to sub-linear time algorithms that have a direct access to the representation of the object. For example, adjacency queries and incidence queries cannot emulate one another in small time (i.e., in time that is sub-linear in the number of vertices).

Both aspects are further clarified by the examples provided in §10.1.2.2.

The essential role of the promise. Recall that, for a fixed constant $\delta > 0$, we consider the promise problem of distinguishing S from $\Gamma_\delta(S)$. The promise means that all instances that are neither in S nor far from S (i.e., not in $\Gamma_\delta(S)$) are ignored, which is essential for sub-linear algorithms for natural problems. This makes the property testing task potentially easier than the corresponding standard decision task (cf. §10.1.2.2). To demonstrate the point, consider the set S consisting of strings that have a majority of 1's. Then, deciding membership in S requires linear time, because random n-bit long strings with $\lfloor n/2 \rfloor$ ones cannot be distinguished from random n-bit long strings with $\lfloor n/2 \rfloor + 1$ ones by probing a sub-linear number of locations (even if randomization and error probability are allowed – see Exercise 10.8). On the other hand, the fraction of 1's in the input can be approximated by a randomized polylogarithmic time algorithm (which yields a property tester for S; see Exercise 10.9). Thus, for some sets, deciding membership requires linear time, while property testing can be done in polylogarithmic time.

The essential role of randomization. Referring to the foregoing example, we note that randomization is essential for any sub-linear time algorithm that distinguishes this set S from, say, $\Gamma_{0.4}(S)$. Specifically, a sub-linear time deterministic algorithm cannot distinguish 1^n from any input that has 1's in each position probed by that algorithm on input 1^n. In general, on input x, a (sub-linear time) deterministic algorithm always reads the same bits of x and thus cannot distinguish x from any z that agrees with x on these bit locations.

Note that, in both cases, we are able to prove lower-bounds on the time complexity of algorithms. This success is due to the fact that these lower-bounds are actually information theoretic in nature; that is, these lower-bounds actually refer to the number of queries performed by these algorithms.

10.1.2.2 Two models for testing graph properties

In this subsection we consider the complexity of property testing for sets of graphs that are *closed under graph isomorphism*; such sets are called graph properties. In view of the importance of representation in the context of property testing, we consider two standard representations of graphs (cf. Appendix G.1), which indeed yield two different models of testing graph properties.

1. The adjacency matrix representation. Here a graph $G = ([N], E)$ is represented (in a somewhat redundant form) by an N-by-N Boolean matrix $M_G = (m_{i,j})_{i,j\in[N]}$ such that $m_{i,j} = 1$ if and only if $\{i, j\} \in E$.

2. Bounded incidence-lists representation. For a fixed parameter d, a graph $G = ([N], E)$ of degree at most d is represented (in a somewhat redundant form) by a mapping $\mu_G : [N] \times [d] \to [N] \cup \{\perp\}$ such that $\mu_G(u, i) = v$ if v is the i^{th} neighbor of u and $\mu_G(u, i) = \perp$ if v has less than i neighbors.

We stress that the aforementioned representations determine both the notion of distance between graphs and the type of queries performed by the algorithm. As we shall see, the difference between these two representations yields a big difference in the complexity of corresponding property testing problems.

Theorem 10.12 (property testing in the adjacency matrix representation): *For any fixed $\delta > 0$ and each of the following sets, there exists a polylogarithmic time randomized algorithm that solves the corresponding property testing problem* (with respect to δ).

- *For every fixed $k \geq 2$, the set of k-colorable graphs.*
- *For every fixed $\rho > 0$, the set of graphs having a clique* (resp., independent set) *of density ρ.*

- *For every fixed $\rho > 0$, the set of N-vertex graphs having a cut*[3] *with at least $\rho \cdot N^2$ edges.*

- *For every fixed $\rho > 0$, the set of N-vertex graphs having a bisection*[3] *with at most $\rho \cdot N^2$ edges.*

In contrast, for some $\delta > 0$, there exists a graph property in $\mathcal{NP}$ for which property testing (with respect to δ) *requires linear time.*

The testing algorithms use a constant number of queries, where this constant is polynomial in the constant $1/\delta$. We highlight the fact that exact decision procedures for the corresponding sets require a linear number of queries. The running time of the aforementioned algorithms hides a constant that is exponential in their query complexity (except for the case of 2-colorability where the hidden constant is polynomial in $1/\delta$). Note that such dependencies seem essential, since setting $\delta = 1/N^2$ regains the original (non-relaxed) decision problems (which, with the exception of 2-colorability, are all NP-complete). Turning to the lower-bound, we note that the graph property for which this bound is proved is not a natural one. Again, the lower-bound on the time complexity follows from a lower-bound on the query complexity.

Theorem 10.12 exhibits a dichotomy between graph properties for which property testing is possible by a constant number of queries and graph properties for which property testing requires a linear number of queries. A combinatorial characterization of the graph properties for which property testing is possible (in the adjacency matrix representation) when using a constant number of queries is known.[4] We note that the constant in this characterization may depend arbitrarily on δ (and indeed, in some cases, it is a function growing faster than a tower of $1/\delta$ exponents). For example, property testing for the set of *triangle-free* graphs is possible by using a number of queries that depends only on δ, but it is known that this number must grow faster than any polynomial in $1/\delta$.

Turning back to Theorem 10.12, we note that the results regarding property testing for the sets corresponding to max-cut and min-bisection yield approximation algorithms with an additive error term (of δN^2). For dense graphs (i.e., N-vertex graphs having $\Omega(N^2)$ edges), this yields a constant factor approximation for the standard approximation problem (as in Definition 10.1). That is, for every constant $c > 1$, we obtain a *c-factor approximation* of the problem of maximizing the size of a cut (resp., minimizing the size of a bisection) *in dense graphs*. On the other hand, the result regarding clique yields a so called dual-approximation for maximum clique; that is, we approximate the minimum number of missing edges in the densest induced subgraph of a given size.

---

[3] A cut in a graph $G = ([N], E)$ is a partition (S_1, S_2) of the set of vertices (i.e., $S_1 \cup S_2 = [N]$ and $S_1 \cap S_2 = \emptyset$), and the edges of the cut are the edges with exactly one endpoint in S_1. A bisection is a cut of the graph to two parts of equal cardinality.

[4] Describing this fascinating result of Alon *et. al.* [8], which refers to the notion of regular partitions (introduced by Szemerédi), is beyond the scope of the current text.

Indeed, Theorem 10.12 is meaningful only for dense graphs. This holds, in general, for any graph property in the adjacency matrix representation.[5] Also note that property testing is trivial, under the adjacency matrix representation, for any graph property S satisfying $\Gamma_{o(1)}(S) = \emptyset$ (e.g., the set of connected graphs, the set of Hamiltonian graphs, etc).

We now turn to the bounded incidence-lists representation, which is relevant only for bounded degree graphs. The problems of max-cut, min-bisection and clique (as in Theorem 10.12) are trivial under this representation, but graph connectivity becomes non-trivial, and the complexity of property testing for the set of bipartite graphs changes dramatically.

Theorem 10.13 (property testing in the bounded incidence-lists representation): *The following assertions refer to the representation of graphs by incidence-lists of length d.*

- *For any fixed d and $\delta > 0$, there exists a polylogarithmic time randomized algorithm that solves the property testing problem for the set of connected graphs of degree at most d.*
- *For any fixed d and $\delta > 0$, there exists a sub-linear randomized algorithm that solves the property testing problem for the set of bipartite graphs of degree at most d. Specifically, on input an N-vertex graph, the algorithm runs for $\widetilde{O}(\sqrt{N})$ time.*
- *For any fixed $d \geq 3$ and some $\delta > 0$, property testing for the set of N-vertex* (3-regular) *bipartite graphs requires $\Omega(\sqrt{N})$ queries.*
- *For some fixed d and $\delta > 0$, property testing for the set of N-vertex 3-colorable graphs requires $\Omega(N)$ queries.*

The running time of the algorithms hides a constant that is polynomial in $1/\delta$. Providing a characterization of graph properties according to the complexity of the corresponding tester (in the bounded incidence-lists representation) is an interesting open problem.

Decoupling the distance from the representation. So far, we have confined our attention to the Hamming distance between the representations of graphs. This made the choice of representation even more important than usual (i.e., more crucial than is common in complexity theory). In contrast, it is natural to consider a notion of distance between graphs that is independent of their representation. For example, the distance between $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be defined

[5] In this model, as shown next, property testing of non-dense graphs is trivial. Specifically, fixing the distance parameter δ, we call a N-vertex graph non-dense if it has less than $(\delta/2) \cdot \binom{N}{2}$ edges. The point is that, for non-dense graphs, the property testing problem for any set S is trivial, because we may just accept any non-dense graph if and only if S contains some non-dense graph. Clearly, the decision is correct in the case that S does not contain non-dense graphs. However, the decision is admissible also in the case that S does contain some non-dense graph, because in this case every non-dense graph is "δ-close" to S (i.e., it is not in $\Gamma_\delta(S)$).

as the minimum of the size of symmetric difference between E_1 and the set of edges in a graph that is isomorphic to G_2. The corresponding relative distance may be defined as the distance divided by $|E_1| + |E_2|$ (or by $\max(|E_1|, |E_2|)$).

10.1.2.3 Beyond graph properties

Property testing has been applied to a variety of computational problems beyond the domain of graph theory. In fact, this area first emerged in the algebraic domain, where the instances (to be viewed as inputs to the testing algorithm) are functions and the relevant properties are sets of algebraic functions. The archetypical example is the set of low-degree polynomials; that is, m-variate polynomials of total (or individual) degree d over some finite field $\mathrm{GF}(q)$, where m, d and q are parameters that may depend on the length of the input (or satisfy some relationships; e.g., $q = d^3 = m^6$). Note that, in this case, the input is the ("full" or "explicit") description of an m-variate function over $\mathrm{GF}(q)$, which means that it has length $q^m \cdot \log_2 q$. Viewing the problem instance as a function suggests a natural measure of distance (i.e., the fraction of arguments on which the functions disagree) as well as a natural way of accessing the instance (i.e., querying the function for the value of selected arguments).

Note that we have referred to these computational problems, under a different terminology, in §9.3.2.2 and in §9.3.2.1. In particular, in §9.3.2.1 we refereed to the special case of linear Boolean functions (i.e., individual degree 1 and $q = 2$), whereas in §9.3.2.2 we used the setting $q = \mathrm{poly}(d)$ and $m = d/\log d$ (where d is a bound on the total degree).

Other domains of computational problems in which property testing was studied include geometry (e.g., clustering problems), formal languages (e.g., testing membership in regular sets), coding theory (cf. Appendix E.1.2), probability theory (e.g., testing equality of distributions), and combinatorics (e.g., monotone and junta functions). As discuss at the end of §10.1.2.2, it is often natural to decouple the distance measure from the representation of the objects (i.e., the way of accessing the problem instance). This is done by introducing a representation-independent notion of distance between instances, which should be natural in the context of the problem at hand.

10.2 Average Case Complexity

Teaching note: We view average-case complexity as referring to the performance on average (or typical) instances, and not as the average performance on random instances. This choice is justified in §10.2.1.1. Thus, it may be more justified to refer to the following theory by the name typical-case complexity. Still, the name average-case was retained for historical reasons.

Our approach so far (including in Section 10.1) is termed worst-case complexity, because it refers to the performance of potential algorithms on each legitimate instance (and hence to the performance on the worst possible instance). That is, computational problems were defined as referring to a set of instances and performance guarantees were required to hold for each instance in this set. In contrast, average-case complexity allows ignoring a negligible measure of the possible instances, where *the identity of the ignored instances is determined by the analysis of potential solvers and not by the problem's statement*.

A few comments are in place. Firstly, as just hinted, the standard statement of the worst-case complexity of a computational problem (especially one having a promise) may also ignores some instances (i.e., those considered inadmissible or violating the promise), but these instances are determined by the problem's statement. In contrast, the inputs ignored in average-case complexity are not inadmissible in any inherent sense (and are certainly not identified as such by the problem's statement). It is just that they are viewed as exceptional when claiming that a specific algorithm solve the problem; furthermore, these exceptional instances are determined by the analysis of that algorithm. Needless to say, these exceptional instances ought to be rare (i.e., occur with negligible probability).

The last sentence raises a couple of issues. Firstly, a distribution on the set of admissible instances has to be specified. In fact, we shall consider a new type of computational problems, each consisting of a standard computational problem coupled with a probability distribution on instances. Consequently, the question of which distributions should be considered in the theory of average-case complexity arises. This question and numerous other definitional issues will be addressed in §10.2.1.1.

Before proceeding, let us spell out the rather straightforward motivation to the study of the average-case complexity of computational problems: It is that, in real-life applications, one may be perfectly happy with an algorithm that solves the problem fast on almost all instances that arise in the relevant application. That is, one may be willing to tolerate error provided that it occurs with negligible probability, where the probability is taken over the distribution of instances encountered in the application. The study of average-case complexity is aimed at exploring the possible benefit of such a relaxation, distinguishing cases in which a benefit exists from cases in which it does not exist. A key aspect in such a study is a good modeling of the type of distributions (of instances) that are encountered in natural algorithmic applications.

A preliminary question that arises is *whether every natural computational problem be solve efficiently when restricting attention to typical instances?* The conjecture that underlies this section is that, for a well-motivated choice of definitions, the answer is negative; that is, our conjecture is that the "distributional version" of NP is not contained in the average-case (or typical-case) version of P. This means that some NP problems are not merely hard in the worst-case, but are rather "typically hard" (i.e., hard on typical instances drawn from some simple distribution). Specifically, *hard instances may occur in natural algorithmic applications* (and not only in cryptographic (or other "adversarial") applications that are design on purpose to produce hard instances).[6]

[6]We highlight two differences between the current context (of natural algorithmic applications) and the context of cryptography. Firstly, in the current context and when referring to problems

The foregoing conjecture motivates the development of an average-case analogue of NP-completeness, which will be presented in this section. Indeed, the entire section may be viewed as an average-case analogue of Chapter 2. In particular, this (average-case) theory identifies distributional problems that are "typically hard" provided that distributional problems that are "typically hard" exist at all. If one believes the foregoing conjecture then, for such complete (distributional) problems, one should not seek algorithms that solve these problems efficiently on typical instances.

Organization. A major part of our exposition is devoted to the definitional issues that arise when developing a general theory of average-case complexity. These issues are discussed in §10.2.1.1. In §10.2.1.2 we prove the existence of distributional problems that are "NP-complete" in the average-case complexity sense. In particular, we show how to obtain such a distributional version for any natural NP-complete decision problem. In §10.2.1.3 we extend the treatment to randomized algorithms. Additional ramifications are presented in Section 10.2.2.

10.2.1 The basic theory

In this section we provide a basic treatment of the theory of average-case complexity, while postponing important ramifications to Section 10.2.2. The basic treatment consists of the preferred definitional choices for the main concepts as well as the identification of complete problems for a natural class of average-case computational problems.

10.2.1.1 Definitional issues

The theory of average-case complexity is more subtle than may appear at first thought. In addition to the generic difficulty involved in defining relaxations, difficulties arise from the "interface" between standard probabilistic analysis and the conventions of complexity theory. This is most striking in the definition of the class of feasible average-case computations. Referring to the theory of worst-case complexity as a guideline, we shall address the following aspects of the analogous theory of average-case complexity.

1. *Setting the general framework.* We shall consider distributional problems, which are standard computational problems (see Section 1.2.2) coupled with distributions on the relevant instances.

2. *Identifying the class of feasible* (distributional) *problems.* Seeking an average-case analogue of classes such as $\mathcal{P}$, we shall reject the first definition that comes to mind (i.e., the naive notion of "average polynomial-time"), briefly

that are typically hard, the simplicity of the underlying input distribution is of great concern: the simpler this distribution, the more appealing the hardness assertion becomes. This concern is irrelevant in the context of cryptography. On the other hand (see discussion at the beginning of Section 7.1.1 and/or at end of §10.2.2.2), cryptographic applications require the ability to efficiently generate hard instances *together with corresponding solutions.*

discuss several related alternatives, and adopt one of them for the main treatment.

3. *Identifying the class of interesting* (distributional) *problems.* Seeking an average-case analogue of the class $\mathcal{NP}$, we shall avoid both the extreme of allowing arbitrary distributions (which collapses average-case hardness to worst-case hardness) and the opposite extreme of confining the treatment to a single distribution such as the uniform distribution.

4. *Developing an adequate notion of reduction among* (distributional) *problems.* As in the theory of worst-case complexity, this notion should preserve feasible solveability (in the current distributional context).

We now turn to the actual treatment of each of the aforementioned aspects.

Step 1: Defining distributional problems. Focusing on decision problems, we define distributional problems as pairs consisting of a decision problem and a probability ensemble.[7] For simplicity, here a probability ensemble $\{X_n\}_{n\in\mathbb{N}}$ is a sequence of random variables such that X_n ranges over $\{0,1\}^n$. Thus, $(S, \{X_n\}_{n\in\mathbb{N}})$ is the distributional problem consisting of the problem of deciding membership in the set S with respect to the probability ensemble $\{X_n\}_{n\in\mathbb{N}}$. (The treatment of search problem is similar; see §10.2.2.1.) We denote the uniform probability ensemble by $U = \{U_n\}_{n\in\mathbb{N}}$; that is, U_n is uniform over $\{0,1\}^n$.

Step 2: Identifying the class of feasible problems. The first idea that comes to mind is defining the problem $(S, \{X_n\}_{n\in\mathbb{N}})$ as feasible (on the average) if there exists an algorithm A that solves S such that the *average running time* of A on X_n is bounded by a polynomial in n (i.e., there exists a polynomial p such that $\mathsf{E}[t_A(X_n)] \leq p(n)$, where $t_A(x)$ denotes the running-time of A on input x). The problem with this definition is that it very sensitive to the model of computation and is not closed under algorithmic composition. Both deficiencies are a consequence of the fact that t_A may be polynomial on the average with respect to $\{X_n\}_{n\in\mathbb{N}}$ but t_A^2 may fail to be so (e.g., consider $t_A(x'x'') = 2^{|x'|}$ if $x' = x''$ and $t_A(x'x'') = |x'x''|^2$ otherwise, coupled with the uniform distribution over $\{0,1\}^n$). We conclude that the *average running-time* of algorithms is not a robust notion. We also doubt the naive appeal of this notion, and view the *typical running time* of algorithms (as defined next) as a more natural notion. Thus, we shall consider an algorithm as feasible if its running-time is typically polynomial.[8]

[7] We mention that even this choice is not evident. Specifically, Levin [148] (see discussion in [85]) advocates the use of a single probability distribution defined over the set of all strings. His argument is that this makes the theory less representation-dependent. At the time we were convinced of his argument (see [85]), but currently we feel that the representation-dependent effects discussed in [85] are legitimate. Furthermore, the alternative formulation of [148, 85] comes across as unnatural and tends to confuse some readers.

[8] An alternative choice, taken by Levin [148] (see discussion in [85]), is considering as feasible (w.r.t $X = \{X_n\}_{n\in\mathbb{N}}$) any algorithm that runs in time that is polynomial in a function that is linear on the average (w.r.t X); that is, requiring that there exists a polynomial p and a function

We say that A is typically polynomial-time on $X = \{X_n\}_{n\in\mathbb{N}}$ if there exists a polynomial p such that the probability that A runs more that $p(n)$ steps on X_n is *negligible* (i.e., for every polynomial q and all sufficiently large n it holds that $\Pr[t_A(X_n) > p(n)] < 1/q(n)$). The question is what is required in the "untypical" cases, and two possible definitions follow.

1. The simpler option is saying that $(S, \{X_n\}_{n\in\mathbb{N}})$ is (typically) feasible if there exists an algorithm A that solves S such that A is typically polynomial-time on $X = \{X_n\}_{n\in\mathbb{N}}$. This effectively requires A to *correctly solve S on each instance*, which is more than was required in the motivational discussion. (Indeed, if the underlying motivation is ignoring rare cases, then we should ignore them altogether rather than ignoring them in a partial manner (i.e., only ignore their affect on the running-time).)

2. The alternative, which fits the motivational discussion, is saying that (S, X) is (typically) feasible if there exists an algorithm A such that A typically solves S on X in polynomial-time; that is, there exists a polynomial p such that *the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible.* This formulation totally ignores the untypical instances. Indeed, in this case we may assume, without loss of generality, that A always runs in polynomial-time (see Exercise 10.11), but we shall not do so here (in order to facilitate viewing the first option as a special case of the current option).

We stress that both alternatives actually define *typical* feasibility and not *average-case* feasibility. To illustrate the difference between the two options, consider the distributional problem of deciding whether a uniformly selected (n-vertex) graph contains a Hamiltonian path. Intuitively, this problem is "typically trivial" (with respect to the uniform distribution)[9] because the algorithm may always say `yes` and be wrong with exponentially vanishing probability. Indeed, this trivial algorithm is admissible by the second approach, but not by the first approach. In light of the foregoing discussions, we adopt the second approach.

Definition 10.14 (the class tpc$\mathcal{P}$): *We say that A* typically solves $(S, \{X_n\}_{n\in\mathbb{N}})$ in polynomial-time *if there exists a polynomial p such that the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible.*[10] *We denote by* tpc$\mathcal{P}$ *the class of distributional problems that are typically solvable in polynomial-time.*

$\ell : \{0,1\}^* \to \mathbb{N}$ such that $t(x) \leq p(\ell(x))$ and $\mathsf{E}[\ell(X_n)] = O(n)$. This definition is robust (i.e., it does not suffer from the aforementioned deficiencies) and is arguably as "natural" as the naive definition (i.e., $\mathsf{E}[t_A(X_n)] \leq \text{poly}(n)$).

[9]In contrast, testing whether a given graph contains a Hamiltonian path seems "typically hard" for other distributions (see Exercise 10.27). Needless to say, in the latter distributions both yes-instances and no-instances appear with noticeable probability.

[10]Recall that a function $\mu : \mathbb{N} \to \mathbb{N}$ is negligible if for every positive polynomial q and all sufficiently large n it holds that $\mu(n) < 1/q(n)$. We say that A errs on x if $A(x)$ differs from the indicator value of the predicate $x \in S$.

Clearly, for every $S \in \mathcal{P}$ and every probability ensemble X, it holds that $(S, X) \in$ tpc$\mathcal{P}$. However, tpc$\mathcal{P}$ contains also distributional problems (S, X) with $S \notin \mathcal{P}$ (see Exercises 10.12 and 10.13). The big question, which underlies the theory of average-case complexity, is whether *natural distributional versions* of $\mathcal{NP}$ are in tpc$\mathcal{P}$. Thus, we turn to identify such versions.

Step 3: Identifying the class of interesting problems. Seeking to identify reasonable distributional versions of $\mathcal{NP}$, we note that two extreme choices should be avoided. On one hand, we must limit the class of admissible distributions so to prevent the collapse of average-case hardness to worst-case hardness (by a selection of a pathological distribution that resides on the "worst case" instances). On the other hand, we should allow for various types of natural distributions rather than confining attention merely to the uniform distribution.[11] Recall that our aim is addressing all possible input distributions that may occur in applications, and thus there is no justification for confining attention to the uniform distribution. Still, arguably, the distributions occuring in applications are "relatively simple" and so we seek to identify a class of simple distributions. One such notion (of simple distributions) underlies the following definition, while a more liberal notion will be presented in §10.2.2.2.

Definition 10.15 (the class dist$\mathcal{NP}$): *We say that a probability ensemble $X = \{X_n\}_{n \in \mathbb{N}}$ is* simple *if there exists a polynomial time algorithm that, on any input $x \in \{0,1\}^*$, outputs* $\mathsf{Pr}[X_{|x|} \leq x]$*, where the inequality refers to the standard lexicographic order of strings. We denote by* dist$\mathcal{NP}$ *the class of distributional problems consisting of decision problems in $\mathcal{NP}$ coupled with simple probability ensembles.*

Note that the uniform probability ensemble is simple, but so are many other "simple" probability ensembles. Actually, it makes sense to relax the definition such that the algorithm is only required to output an approximation of $\mathsf{Pr}[X_{|x|} \leq x]$, say, to within a factor of $1 \pm 2^{-2|x|}$. We note that Definition 10.15 interprets simplicity in computational terms; specifically, as the feasibility of answering very basic questions regarding the probability distribution (i.e., determining the probability mass assigned to a single (n-bit long) string and even to an interval of such strings). This simplicity condition is closely related to being polynomial-time sampleable via a *monotone* mapping (see Exercise 10.14).

Teaching note: The following two paragraphs attempt to address some doubts regarding Definition 10.15. One may postpone such discussions to a later stage.

We admit that the indentification of simple distributions as the class of interesting distribution is significantly more questionable than any other indentification advocated in this book. Nevertheless, we believe that we were fully justified in rejecting both the aforementioned extremes (i.e., of either allowing all distributions

[11] Confining attention to the uniform distribution seems misguided by the naive belief according to which this distribution is the *only* one relevant to applications. In contrast, we believe that, for most natural applications, the uniform distribution over instances is not relevant at all.

or allowing only the uniform distribution). Yet, the reader may wonder whether or not we have struck the right balance between "generality" and "simplicity" (in the intuitive sense). One specific concern is that we might have restricted the class of distributions too much. We briefly address this concern next.

A more intuitive and very robust class of distributions, which seems to contain all distributions that may occur in applications, is the class of polynomial-time sampleable probability ensembles (treated in §10.2.2.2). Fortunately, the combination of the results presented in §10.2.1.2 and §10.2.2.2 seems to retrospectively endorse the choice underlying Definition 10.15. Speficically, we note that enlarging the class of distributions weakens the *conjecture* that the corresponding class of distributional NP problems contains infeasible problems. On the other hand, the *conclusion* that a specific distributional problem is not feasible becomes stronger when the problem belongs to a smaller class that corresponds to a restricted definition of admissible distributions. Now, the combined results of §10.2.1.2 and §10.2.2.2 assert that a conjecture that refers to the larger class of polynomial-time sampleable ensembles implies a conclusion that refers to a (very) simple probability ensemble (which resides in the smaller class). Thus, the current setting in which both the conjecture and the conclusion refer to simple probability ensembles may be viewed as just an intermediate step.

Indeed, the big question in the current context is whether dist$\mathcal{NP}$ is contained in tpc$\mathcal{P}$. A positive answer (especially if extended to sampleable ensembles) would deem the P-vs-NP Question of little practical significant. However, our daily experience as well as much research effort indicate that some NP problems are not merely hard in the worst-case, but rather "typically hard". This leads to the *conjecture that* dist$\mathcal{NP}$ *is not contained in* tpc$\mathcal{P}$.

Needless to say, the latter conjecture implies $\mathcal{P} \neq \mathcal{NP}$, and thus we should not expect to see a proof of it. In particular, we should not expect to see a proof that some specific problem in dist$\mathcal{NP}$ is not in tpc$\mathcal{P}$. What we may hope to see is "dist$\mathcal{NP}$-complete" problems; that is, problems in dist$\mathcal{NP}$ that are not in tpc$\mathcal{P}$ unless the entire class dist$\mathcal{NP}$ is contained in tpc$\mathcal{P}$. An adequate notion of a reduction is used towards formulating this possibility (which in turn is captured by the notion of "dist$\mathcal{NP}$-complete" problems).

Step 4: Defining reductions among (distributional) problems. Intuitively, such reductions must preserve average-case feasibility. Thus, in addition to the standard conditions (i.e., that the reduction be efficiently computable and yield a correct result), we require that the reduction "respects" the probability distribution of the corresponding distributional problems. Specifically, the reduction should not map very likely instances of the first ("starting") problem to rare instances of the second ("target") problem. Otherwise, having a typically polynomial-time algorithm for the second distributional problem does not necessarily yield such an algorithm for the first distributional problem. Following is the adequate analogue of a Cook reduction (i.e., general polynomial-time reduction), where the analogue of a Karp-reduction (many-to-one reduction) can be easily derived as a special case.

Teaching note: One may prefer presenting in class only the special case of many-to-one reductions, which suffices for Theorem 10.17. See Footnote 13.

Definition 10.16 (reductions among distributional problems): *We say that the oracle machine M* reduces *the distributional problem (S,X) to the distributional problem (T,Y) if the following three conditions hold.*

1. Efficiency: *The machine M runs in polynomial-time.*[12]
2. Validity: *For every $x \in \{0,1\}^*$, it holds that $M^T(x) = 1$ if an only if $x \in S$, where $M^T(x)$ denotes the output of the oracle machine M on input x and access to an oracle for T.*
3. Domination:[13] *The probability that, on input X_n and oracle access to T, machine M makes the query y is upper-bounded by $\mathrm{poly}(|y|) \cdot \mathsf{Pr}[Y_{|y|} = y]$. That is, there exists a polynomial p such that, for every $y \in \{0,1\}^*$ and every $n \in \mathbb{N}$, it holds that*

$$\mathsf{Pr}[Q(X_n) \ni y] \leq p(|y|) \cdot \mathsf{Pr}[Y_{|y|} = y], \tag{10.2}$$

where $Q(x)$ denotes the set of queries made by M on input x and oracle access to T.

In addition, we require that the reduction does not make too short queries; that is, there exists a polynomial p' such that if $y \in Q(x)$ then $p'(|y|) \geq |x|$.

The l.h.s. of Eq. (10.2) refers to the probability that, on input distributed as X_n, the reduction makes the query y. This probability is required not to exceed the probability that y occurs in the distribution $Y_{|y|}$ by more than a polynomial factor in $|y|$. In this case we say that the l.h.s. of Eq. (10.2) is dominated by $\mathsf{Pr}[Y_{|y|} = y]$.

Indeed, the domination condition is the only aspect of Definition 10.16 that extends beyond the worst-case treatment of reductions and refers to the distributional setting. The domination condition does not insist that the distribution induced by $Q(X)$ equals Y, but rather allows some slackness that, in turn, is bounded so to guarantee preservation of typical feasibility (see Exercise 10.15).[14]

We note that the reducibility arguments extensively used in Chapters 7 and 8 (see discussion in Section 7.1.2) are actually reductions in the spirit of Definition 10.16 (except that they refer to different types of computational tasks).

[12] In fact, one may relax the requirement and only require that M is typically polynomial-time with respect to X. The validity condition may also be relaxed similarly.

[13] Let us spell out the meaning of Eq. (10.2) in the special case of many-to-one reductions (i.e., $M^T(x) = 1$ if and only if $f(x) \in T$, where f is a polynomial-time computable function): in this case $\mathsf{Pr}[Q(X_n) \ni y]$ is replaced by $\mathsf{Pr}[f(X_n) = y]$. That is, Eq. (10.2) simplifies to $\mathsf{Pr}[f(X_n) = y] \leq p(|y|) \cdot \mathsf{Pr}[Y_{|y|} = y]$. Indeed, this condition holds vacuously for any y that is not in the image of f.

[14] We stress that the notion of domination is incomparable to the notion of statistical (resp., computational) indistinguishability. On one hand, domination is a local requirement (i.e., it compares the two distribution on a point-by-point basis), whereas indistinguishability is a global requirement (which allows rare exceptions). On the other hand, domination does not require approximately equal values, but rather a ratio that is bounded in one direction. Indeed, domination is not symmetric. We comment that a more relaxed notion of domination that allows rare violations (as in Footnote 12) suffices for the preservation of typical feasibility.

10.2.1.2 Complete problems

Recall that our conjecture is that dist$\mathcal{NP}$ is not contained in tpc$\mathcal{P}$, which in turn strengthens the conjecture $\mathcal{P} \neq \mathcal{NP}$ (making infeasibility a typical phenomenon rather than a worst-case one). Having no hope of proving that dist$\mathcal{NP}$ is not contained in tpc$\mathcal{P}$, we turn to the study of complete problems with respect to that conjecture. Specifically, we say that a distributional problem (S, X) is dist$\mathcal{NP}$-complete if $(S, X) \in$ dist$\mathcal{NP}$ and every $(S', X') \in$ dist$\mathcal{NP}$ is reducible to (S, X) (under Definition 10.16).

Recall that it is quite easy to prove the mere existence of NP-complete problems and many natural problems are NP-complete. In contrast, in the current context, establishing completeness results is quite hard. This should not be surprising in light of the restricted type of reductions allowed in the current context. The restriction (captured by the domination condition) requires that "typical" instances of one problem should not be mapped to "untypical" instances of the other problem. However, it is fair to say that standard Karp-reductions (used in establishing NP-completeness results) map "typical" instances of one problem to somewhat "bizarre" instances of the second problem. Thus, the current subsection may be viewed as a study of reductions that do not commit this sin.[15]

Theorem 10.17 (dist$\mathcal{NP}$-completeness): *dist$\mathcal{NP}$ contains a distributional problem (T, Y) such that each distributional problem in* dist$\mathcal{NP}$ *is reducible* (per Definition 10.16) *to (T, Y). Furthermore, the reductions are via many-to-one mappings.*

Proof: We start by introducing such a (distributional) problem, which is a natural distributional version of the decision problem $S_{\mathbf{u}}$ (used in the proof of Theorem 2.18). Recall that $S_{\mathbf{u}}$ contains the instance $\langle M, x, 1^t \rangle$ if there exists $y \in \cup_{i \leq t}\{0,1\}^i$ such that M accepts the input pair (x, y) within t steps. We couple $S_{\mathbf{u}}$ with the "quasi-uniform" probability ensemble U' that assigns to the instance $\langle M, x, 1^t \rangle$ a probability mass proportional to $2^{-(|M|+|x|)}$. Specifically, for every $\langle M, x, 1^t \rangle$ it holds that

$$\mathsf{Pr}[U'_n = \langle M, x, 1^t \rangle] = \frac{2^{-(|M|+|x|)}}{\binom{n}{2}} \tag{10.3}$$

where $n \stackrel{\text{def}}{=} |\langle M, x, 1^t \rangle| \stackrel{\text{def}}{=} |M| + |x| + t$. Note that, under a suitable natural encoding, the ensemble U' is indeed simple.[16]

The reader can easily verify that the generic reduction used when reducing any set in $\mathcal{NP}$ to $S_{\mathbf{u}}$ (see the proof of Theorem 2.18), fails to reduce dist$\mathcal{NP}$ to $(S_{\mathbf{u}}, U')$. Specifically, in some cases (see next paragraph), these reductions do not satisfy the domination condition. Indeed, the difficulty is that we have to

[15] The latter assertion is somewhat controversial. While it seems totally justified with respect to the proof of Theorem 10.17, opinions regarding the proof of Theorem 10.19 may differ.

[16] For example, we may encode $\langle M, x, 1^t \rangle$, where $M = \sigma_1 \cdots \sigma_k \in \{0,1\}^k$ and $x = \tau_1 \cdots \tau_\ell \in \{0,1\}^\ell$, by the string $\sigma_1\sigma_1 \cdots \sigma_k\sigma_k 01 \tau_1\tau_1 \cdots \tau_\ell\tau_\ell 01^t$. Then $\binom{n}{2} \cdot \mathsf{Pr}[U'_n \leq \langle M, x, 1^t \rangle]$ equals $(i_{|M|,|x|,t} - 1) + 2^{-|M|} \cdot |\{M' \in \{0,1\}^{|M|} : M' < M\}| + 2^{-(|M|+|x|)} \cdot |\{x' \in \{0,1\}^{|x|} : x' \leq x\}|$, where $i_{k,\ell,t}$ is the ranking of $\{k, k+\ell\}$ among all 2-subsets of $[k+\ell+t]$.

reduce all dist$\mathcal{NP}$ problems (i.e., pairs consisting of decision problems and simple distributions) to one single distributional problem (i.e., $(S_{\mathtt{u}}, U')$). Applying the aforementioned reductions, we end up with many distributional versions of $S_{\mathtt{u}}$, and furthermore the corresponding distributions are very different (and are not necessarily dominated by a single distribution).

Let us take a closer look at the aforementioned generic reduction, when applied to an arbitrary $(S, X) \in$ dist$\mathcal{NP}$. This reduction maps an instance x to a triple $(M_S, x, 1^{p_S(|x|)})$, where M_S is a machine verifying membership in S (while using adequate NP-witnesses) and p_S is an adequate polynomial. The problem is that x may have relatively large probability mass (i.e., it may be that $\mathsf{Pr}[X_{|x|} = x] \gg 2^{-|x|}$) while $(M_S, x, 1^{p_S(|x|)})$ has "uniform" probability mass (i.e., $\langle M_S, x, 1^{p_S(|x|)}\rangle$ has probability mass smaller than $2^{-|x|}$ in U'). This violates the domination condition (see Exercise 10.18), and thus an alternative reduction is required.

The key to the alternative reduction is an (efficiently computable) encoding of strings taken from an arbitrary *simple* distribution by strings that have a similar probability mass under the uniform distribution. This means that the encoding should shrink strings that have relatively large probability mass under the original distribution. Specifically, this encoding will map x (taken from the ensemble $\{X_n\}_{n\in\mathbb{N}}$) to a codeword x' of length that is upper-bounded by the logarithm of $1/\mathsf{Pr}[X_{|x|} = x]$, ensuring that $\mathsf{Pr}[X_{|x|} = x] = O(2^{-|x'|})$. Accordingly, the reduction will map x to a triple $(M_{S,X}, x', 1^{p'(|x|)})$, where $|x'| < O(1) + \log_2(1/\mathsf{Pr}[X_{|x|} = x])$ and $M_{S,X}$ is an algorithm that (given x' and x) first verifies that x' is a proper encoding of x and next applies the standard verification (i.e., M_S) of the problem S. Such a reduction will be shown to satisfy all three conditions (i.e., efficiency, validity, and domination). Thus, instead of forcing the structure of the original distribution X on the target distribution U', the reduction will incorporate the structure of X in the reduced instance. A key ingredient in making this possible is the fact that X is simple (as per Definition 10.15).

With the foregoing motivation in mind, we now turn to the actual proof; that is, proving that any $(S, X) \in$ dist$\mathcal{NP}$ is reducible to $(S_{\mathtt{u}}, U')$. The following technical lemma is the basis of the reduction. In this lemma as well as in the sequel, it will be convenient to consider the (accumulative) distribution function of the probability ensemble X. That is, we consider $\mu(x) \stackrel{\text{def}}{=} \mathsf{Pr}[X_{|x|} \leq x]$, and note that $\mu : \{0,1\}^* \to [0,1]$ is polynomial-time computable (because X satisfies Definition 10.15).

Coding Lemma:[17] Let $\mu : \{0,1\}^* \to [0,1]$ be a polynomial-time computable function that is monotonically non-decreasing over $\{0,1\}^n$ for every n (i.e., $\mu(x') \leq \mu(x'')$ for any $x' < x'' \in \{0,1\}^{|x'|}$). For $x \in \{0,1\}^n \setminus \{0^n\}$, let $x - 1$ denote the string preceding x in the lexicographic order of n-bit long strings. Then there exist an encoding function C_μ that satisfies the following three conditions.

[17] The lemma actually refers to $\{0,1\}^n$, for any fixed value of n, but the efficiency condition is stated more easily when allowing n to vary (and using the standard asymptotic analysis of algorithms). Actually, the lemma is somewhat easier to state and establish for polynomial-time computable functions that are monotonically non-decreasing over $\{0,1\}^*$ (rather than over $\{0,1\}^n$). See further discussion in Exercise 10.19.

1. Compression: For every x it holds that $|C_\mu(x)| \leq 1+\min\{|x|, \log_2(1/\mu'(x))\}$, where $\mu'(x) \stackrel{\rm def}{=} \mu(x) - \mu(x-1)$ if $x \notin \{0\}^*$ and $\mu'(0^n) \stackrel{\rm def}{=} \mu(0^n)$ otherwise.
2. Efficient Encoding: The function C_μ is computable in polynomial-time.
3. Unique Decoding: For every $n \in \mathbb{N}$, when restricted to $\{0,1\}^n$, the function C_μ is one-to-one (i.e., if $C_\mu(x) = C_\mu(x')$ and $|x| = |x'|$ then $x = x'$).

Proof: The function C_μ is defined as follows. If $\mu'(x) \leq 2^{-|x|}$ then $C_\mu(x) = 0x$ (i.e., in this case x serves as its own encoding). Otherwise (i.e., $\mu'(x) > 2^{-|x|}$) then $C_\mu(x) = 1z$, where z is chosen such that $|z| \leq \log_2(1/\mu'(x))$ and the mapping of n-bit strings to their encoding is one-to-one. Loosely speaking, z is selected to equal the shortest binary expansion of a number in the interval $(\mu(x)-\mu'(x), \mu(x)]$. Bearing in mind that this interval has length $\mu'(x)$ and that the different intervals are disjoint, we obtain the desired encoding. Details follows.

We focus on the case that $\mu'(x) > 2^{-|x|}$, and detail the way that z is selected (for the encoding $C_\mu(x) = 1z$). If $x > 0^{|x|}$ and $\mu(x) < 1$, then we let z be the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$; for example, if $\mu(1010) = 0.10010$ and $\mu(1011) = 0.10101111$ then $C_\mu(1011) = 1z$ with $z = 10$. Thus, in this case $0.z1$ is in the interval $(\mu(x-1), \mu(x)]$ (i.e., $\mu(x-1) < 0.z1 \leq \mu(x)$). For $x = 0^{|x|}$, we let z be the longest common prefix of the binary expansions of 0 and $\mu(x)$ and again $0.z1$ is in the relevant interval (i.e., $(0, \mu(x)]$). Finally, for x such that $\mu(x) = 1$ and $\mu(x-1) < 1$, we let z be the longest common prefix of the binary expansions of $\mu(x-1)$ and $1-2^{-|x|-1}$, and again $0.z1$ is in $(\mu(x-1), \mu(x)]$ (because $\mu'(x) > 2^{-|x|}$ and $\mu(x-1) < \mu(x) = 1$ imply that $\mu(x-1) < 1-2^{-|x|} < \mu(x)$). Note that if $\mu(x) = \mu(x-1) = 1$ then $\mu'(x) = 0 < 2^{-|x|}$.

We now verify that the foregoing C_μ satisfies the conditions of the lemma. We start with the compression condition. Clearly, if $\mu'(x) \leq 2^{-|x|}$ then $|C_\mu(x)| = 1+|x| \leq 1+\log_2(1/\mu'(x))$. On the other hand, suppose that $\mu'(x) > 2^{-|x|}$ and let us focus on the sub-case that $x > 0^{|x|}$ and $\mu(x) < 1$. Let $z = z_1 \cdots z_\ell$ be the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$. Then, $\mu(x-1) = 0.z0u$ and $\mu(x) = 0.z1v$, where $u, v \in \{0,1\}^*$. We infer that

$$\mu'(x) \;=\; \mu(x) - \mu(x-1) \;\leq\; \left(\sum_{i=1}^{\ell} 2^{-i} z_i + \sum_{i=\ell+1}^{\mathrm{poly}(|x|)} 2^{-i}\right) - \sum_{i=1}^{\ell} 2^{-i} z_i \;<\; 2^{-|z|},$$

and $|z| < \log_2(1/\mu'(x)) \leq |x|$ follows. Thus, $|C_\mu(x)| \leq 1+\min(|x|, \log_2(1/\mu'(x)))$ holds in both cases. Clearly, C_μ can be computed in polynomial-time by computing $\mu(x-1)$ and $\mu(x)$. Finally, note that C_μ satisfies the unique decoding condition, by separately considering the two aforementioned cases (i.e., $C_\mu(x) = 0x$ and $C_\mu(x) = 1z$). Specifically, in the second case (i.e., $C_\mu(x) = 1z$), use the fact that $\mu(x-1) < 0.z1 \leq \mu(x)$. $\square$

To obtain an encoding that is one-to-one when applied to strings of different lengths we augment C_μ in the obvious manner; that is, we consider $C'_\mu(x) \stackrel{\rm def}{=} (|x|, C_\mu(x))$, which may be implemented as $C'_\mu(x) = \sigma_1\sigma_1 \cdots \sigma_\ell\sigma_\ell 01 C_\mu(x)$ where

$\sigma_1 \cdots \sigma_\ell$ is the binary expansion of $|x|$. Note that $|C'_\mu(x)| = O(\log|x|) + |C_\mu(x)|$ and that C'_μ is one-to-one.

The machine associated with (S, X). Let μ be the accumulative probability function associated with the probability ensemble X, and M_S be the polynomial-time machine that verifies membership in S while using adequate NP-witnesses (i.e., $x \in S$ if and only if there exists $y \in \{0,1\}^{\text{poly}(|x|)}$ such that $M(x,y) = 1$). Using the encoding function C'_μ, we introduce an algorithm $M_{S,\mu}$ with the intension of reducing the distributional problem (S, X) to $(S_{\mathtt{u}}, U')$ such that all instances (of S) are mapped to triples in which the first element equals $M_{S,\mu}$. Machine $M_{S,\mu}$ is given an alleged encoding (under C'_μ) of an instance to S along with an alleged proof that the corresponding instance is in S, and verifies these claims in the obvious manner. That is, on input x' and $\langle x, y\rangle$, machine $M_{S,\mu}$ first verifies that $x' = C'_\mu(x)$, and next verifiers that $x \in S$ by running $M_S(x,y)$. Thus, $M_{S,\mu}$ verifies membership in the set $S' = \{C'_\mu(x) : x \in S\}$, while using proofs of the form $\langle x, y\rangle$ such that $M_S(x,y) = 1$ (for the instance $C'_\mu(x)$).[18]

The reduction. We maps an instance x (of S) to the triple $(M_{S,\mu}, C'_\mu(x), 1^{p(|x|)})$, where $p(n) \stackrel{\text{def}}{=} p_S(n) + p_C(n)$ such that p_S is a polynomial representing the running-time of M_S and p_C is a polynomial representing the running-time of the encoding algorithm.

Analyzing the reduction. Our goal is proving that *the foregoing mapping constitutes a reduction of* (S, X) *to* $(S_{\mathtt{u}}, U')$. We verify the corresponding three requirements (of Definition 10.16).

1. Using the fact that C_μ is polynomial-time computable (and noting that p is a polynomial), it follows that the foregoing mapping can be computed in polynomial-time.

2. Recall that, on input $(x', \langle x, y\rangle)$, machine $M_{S,\mu}$ accepts if and only if $x' = C'_\mu(x)$ and M_S accepts (x,y) within $p_S(|x|)$ steps. Using the fact that $C'_\mu(x)$ uniquely determines x, it follows that $x \in S$ if and only if there exists a string y of length at most $p(|x|)$ such that $M_{S,\mu}$ accepts $(C'_\mu(x), \langle x, y\rangle)$ in at most $p(|x|)$ steps. Thus, $x \in S$ if and only if $(M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}) \in S_{\mathtt{u}}$, and the validity condition follows.

3. In order to verify the domination condition, we first note that the foregoing mapping is one-to-one (because the transformation $x \to C'_\mu(x)$ is one-to-one). Next, we note that it suffices to consider instances of $S_{\mathtt{u}}$ that have a preimage under the foregoing mapping (since instances with no preimage trivially satisfy the domination condition). Each of these instances (i.e., each image of this mapping) is a triple with the first element equal to $M_{S,\mu}$ and

[18] Note that $|y| = \text{poly}(|x|)$, but $|x| = \text{poly}(|C'_\mu(x)|)$ does not necessarily hold (and so S' is not necessarily in $\mathcal{NP}$). As we shall see, the latter point is immaterial.

the second element being an encoding under C'_μ. By the definition of U', for every such image $\langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle \in \{0,1\}^n$, it holds that

$$\begin{aligned}\Pr[U'_n = \langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle] &= \binom{n}{2}^{-1} \cdot 2^{-(|M_{S,\mu}|+|C'_\mu(x)|)} \\ &> c \cdot n^{-2} \cdot 2^{-(|C_\mu(x)|+O(\log|x|))},\end{aligned}$$

where $c = 2^{-|M_{S,\mu}|-1}$ is a constant depending only on S and μ (i.e., on the distributional problem (S,X)). Thus, for some positive polynomial q, we have

$$\Pr[U'_n = \langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle] > 2^{-|C_\mu(x)|}/q(n). \tag{10.4}$$

By virtue of the compression condition (of the Coding Lemma), we have $2^{-|C_\mu(x)|} \geq 2^{-1-\min(|x|,\log_2(1/\mu'(x)))}$. It follows that

$$2^{-|C_\mu(x)|} \geq \Pr[X_{|x|} = x]/2. \tag{10.5}$$

Recalling that x is the only preimage that is mapped to $\langle M_{S,\mu}, C'_\mu(x), 1^{p(|x|)}\rangle$ and combining Eq. (10.4) & (10.5), we establish the domination condition.

The theorem follows. ■

Reflections. The proof of Theorem 10.17 highlights the fact that the reduction used in the proof of Theorem 2.18 does not introduce much structure in the reduced instances (i.e., does not reduce the original problem to a "highly structured special case" of the target problem). Put in other words, unlike more advanced worst-case reductions, this reduction does not map "random" (i.e., uniformly distributed) instances to highly structured instances (which occur with negligible probability under the uniform distribution). Thus, the reduction used in the proof of Theorem 2.18 suffices for reducing any distributional problem in dist$\mathcal{NP}$ to a distributional problem consisting of $S_{\mathbf{u}}$ coupled with *some* simple probability ensemble (see Exercise 10.20).[19]

However, Theorem 10.17 states more than the latter assertion. That is, it states that any distributional problem in dist$\mathcal{NP}$ is reducible to the *same* distributional version of $S_{\mathbf{u}}$. Indeed, the effort involved in proving Theorem 10.17 was due to the need for mapping instances taken from any simple probability ensemble (which may not be the uniform ensemble) to instances distributed in a manner that is dominated by a single probability ensemble (i.e., the quasi-uniform ensemble U').

Once we have established the existence of one dist$\mathcal{NP}$-complete problem, we may establish the dist$\mathcal{NP}$-completeness of other problems (in dist$\mathcal{NP}$) by reducing some dist$\mathcal{NP}$-complete problem to them (and relying on the transitivity of reductions (see Exercise 10.17)). Thus, the difficulties encountered in the proof of

[19]Note that this cannot be said of most known Karp-reductions, which do map random instances to highly structured ones. Furthermore, the same (structure creating property) holds for the reductions obtained by Exercise 2.31.

Theorem 10.17 are no longer relevant. Unfortunately, a seemingly more severe difficulty arises: almost all known reductions in the theory of NP-completeness work by introducing much structure in the reduced instances (i.e., they actually reduce to highly structured special cases). Furthermore, this structure is too complex in the sense that the distribution of reduced instances does not seem simple (in the sense of Definition 10.15). Actually, as demonstrated next, the problem is not the existence of a structure in the reduced instances but rather the complexity of this structure. In particular, if the aforementioned reduction is "monotone" and "length regular" then the distribution of the reduced instances is simple enough.

Proposition 10.18 (sufficient condition for dist$\mathcal{NP}$-completeness): *Suppose that f is a Karp-reduction of the set S to the set T such that, for every $x', x'' \in \{0,1\}^*$, the following two conditions hold:*

1. (f is monotone): *If $x' < x''$ then $f(x') < f(x'')$, where the inequalities refer to the standard lexicographic order of strings.*[20]
2. (f is length-regular): *$|x'| = |x''|$ if and only if $|f(x')| = |f(x'')|$.*

Then if there exists an ensemble X such that (S, X) is dist$\mathcal{NP}$-complete then there exists an ensemble Y such that (T, Y) is dist$\mathcal{NP}$-complete.

Proof Sketch: Note that the monotonicity of f implies that f is one-to-one and that for every x it holds that $f(x) \geq x$. Furthermore, as shown next, f is polynomial-time invertible. Intuitively, the fact that f is both monotone and polynomial-time computable implies that a preimage can be found by a binary search. Specifically, given $y = f(x)$, we search for x by iteratively halving the interval of potential solutions, which is initialized to $[0, y]$ (since $x \leq f(x)$). Note that if this search is invoked on a string y that is not in the image of f, then it terminates while detecting this fact.

Relying on the fact that f is one-to-one (and length-regular), we define the probability ensemble $Y = \{Y_n\}_n$ such that for every x it holds that $\Pr[Y_{|f(x)|} = f(x)] = \Pr[X_{|x|} = x]$. Specifically, letting $\ell(m) = |f(1^m)|$ and noting that ℓ is one-to-one and monotonically non-decreasing, we define

$$\Pr[Y_{|y|} = y] = \begin{cases} \Pr[X_{|x|} = x] & \text{if } x = f^{-1}(y) \\ 0 & \text{if } \exists m \text{ s.t. } y \in \{0,1\}^{\ell(m)} \setminus \{f(x) : x \in \{0,1\}^m\} \\ 2^{-|y|} & \text{otherwise (i.e., if } |y| \notin \{\ell(m) : m \in \mathbb{N}\})^{21}. \end{cases}$$

Clearly, (S, X) is reducible to (T, Y) (via the Karp-reduction f, which, due to our construction of Y, also satisfies the domination condition). Thus, using the hypothesis that dist$\mathcal{NP}$ is reducible to (S, X) and the transitivity of reductions (see Exercise 10.17), it follows that every problem in dist$\mathcal{NP}$ is reducible to (T, Y). The

[20] In particular, if $|z'| < |z''|$ then $z' < z''$. Recall that for $|z'| = |z''|$ it holds that $z' < z''$ if and only if there exists $w, u', u'' \in \{0,1\}^*$ such that $z' = w0u'$ and $z'' = w1u''$.

[21] Having Y_n be uniform in this case is a rather arbitrary choice, which is merely aimed at guaranteeing a "simple" distribution on n-bit strings (also in this case).

key observation, to be established next, is that Y is a simple probability ensemble, and it follows that (T, Y) is in dist$\mathcal{NP}$.

Loosely speaking, the simplicity of Y follows by combining the simplicity of X and the properties of f (i.e., the fact that f is monotone, length-regular, and polynomial-time invertible). The monotonicity and length-regularity of f implies that $\Pr[Y_{|f(x)|} \leq f(x)] = \Pr[X_{|x|} \leq x]$. More generally, for any $y \in \{0,1\}^{\ell(m)}$, it holds that $\Pr[Y_{\ell(m)} \leq y] = \Pr[X_m \leq x]$, where x is the lexicographicly largest string such that $f(x) \leq y$ (and, indeed, if $|x| < m$ then $\Pr[Y_{\ell(m)} \leq y] = \Pr[X_m \leq x] = 0$). Note that this x can be found in polynomial-time by the inverting algorithm sketched in the first paragraph of the proof. Thus, we may compute $\Pr[Y_{|y|} \leq y]$ by finding the adequate x and computing $\Pr[X_{|x|} \leq x]$. Using the hypothesis that X is simple, it follows that Y is simple (and the proposition follows). $\square$

On the existence of adequate Karp-reductions. Proposition 10.18 implies that a sufficient condition for dist$\mathcal{NP}$-completeness of a distributional version of some (NP-complete) set T is the existence of an adequate Karp-reduction from the set $S_{\mathbf{u}}$ to the set T; that is, this Karp-reduction should be monotone and length-regular. While the length-regularity condition seems easy to impose (by using adequate padding), the monotonicity condition seems more problematic. Fortunately, it turns out that the monotonicity condition can also be imposed by using adequate padding (or rather an adequate "marking" – see Exercises 2.30 and 10.21). We highlight the fact that the existence of an adequate padding (or "marking") is a property of the set T itself. In Exercise 10.21 we review a method for modifying any Karp-reduction to a "monotonically markable" set T into a Karp-reduction (to T) that is monotone and length-regular. In Exercise 10.23 we provide evidence to the thesis that all natural NP-complete sets are monotonically markable. Combining all these facts, we conclude that *any natural NP-complete decision problem can be coupled with a simple probability ensemble such that the resulting distributional problem is* dist$\mathcal{NP}$*-complete.* As a concrete illustration we state the (formal) result for the twenty-one NP-complete problems treated in Karp's paper on NP-completeness [131].

Theorem 10.19 (a modest version of a general thesis): *For each of the twenty-one NP-complete problems treated in [131] there exists a simple probability ensemble such that the combined distributional problem is* dist$\mathcal{NP}$*-complete.*

The said list of problems includes `SAT`, `Clique`, and `3-Colorability`.

10.2.1.3 Probabilistic versions

The definitions in §10.2.1.1 can be extended so that to account also for randomized computations. For example, extending Definition 10.14, we have:

Definition 10.20 (the class tpc$\mathcal{BPP}$): *For a probabilistic algorithm A, a Boolean function f, and a time-bound function $t:\mathbb{N}\to\mathbb{N}$, we say that the string x is t-*bad for A with respect to f *if with probability exceeding $1/3$, on input x, either $A(x) \neq f(x)$*

or A runs more that $t(|x|)$ steps. We say that A typically solves $(S,\{X_n\}_{n\in\mathbb{N}})$ in probabilistic polynomial-time *if there exists a polynomial p such that the probability that X_n is p-bad for A with respect to the characteristic function of S is negligible. We denote by* tpc$\mathcal{BPP}$ *the class of distributional problems that are typically solvable in probabilistic polynomial-time.*

The definition of reductions can be similarly extended. This means that in Definition 10.16, both $M^T(x)$ and $Q(x)$ (mentioned in Items 2 and 3, respectively) are random variables rather than fixed objects. Furthermore, validity is required to hold (for every input) only with probability 2/3, where the probability space refers only to the internal coin tosses of the reduction. Randomized reductions are closed under composition and preserve typical feasibility (see Exercise 10.24).

Randomized reductions allow the presentation of a dist$\mathcal{NP}$-complete problem that refers to the (perfectly) uniform ensemble. Recall that Theorem 10.17 establishes the dist$\mathcal{NP}$-completeness of $(S_{\mathtt{u}},U')$, where U' is a quasi-uniform ensemble (i.e., $\Pr[U'_n=\langle M,x,1^t\rangle]=2^{-(|M|+|x|)}/\binom{n}{2}$, where $n=|\langle M,x,1^t\rangle|$). We first note that $(S_{\mathtt{u}},U')$ can be randomly reduced to $(S'_{\mathtt{u}},U'')$, where $S'_{\mathtt{u}}=\{\langle M,x,z\rangle : \langle M,x,1^{|z|}\rangle\in S_{\mathtt{u}}\}$ and $\Pr[U''_n=\langle M,x,z\rangle]=2^{-(|M|+|x|+|z|)}/\binom{n}{2}$ for every $\langle M,x,z\rangle\in\{0,1\}^n$. The randomized reduction consists of mapping $\langle M,x,1^t\rangle$ to $\langle M,x,z\rangle$, where z is uniformly selected in $\{0,1\}^t$. Recalling that $U=\{U_n\}_{n\in\mathbb{N}}$ denotes the uniform probability ensemble (i.e., U_n is uniformly distributed on strings of length n) and using a suitable encoding we get.

Proposition 10.21 *There exists $S\in\mathcal{NP}$ such that every $(S',X')\in$ dist$\mathcal{NP}$ is randomly reducible to (S,U).*

Proof Sketch: By the forgoing discussion, every $(S',X')\in$ dist$\mathcal{NP}$ is randomly reducible to $(S'_{\mathtt{u}},U'')$, where the reduction goes through $(S_{\mathtt{u}},U')$. Thus, we focus on reducing $(S'_{\mathtt{u}},U'')$ to $(S''_{\mathtt{u}},U)$, where $S''_{\mathtt{u}}\in\mathcal{NP}$ is defined as follows. The string $\mathrm{bin}_\ell(|u|)\cdot\mathrm{bin}_\ell(|v|)\cdot u\cdot v\cdot w$ is in $S''_{\mathtt{u}}$ if and only if $\langle u,v,w\rangle\in S'_{\mathtt{u}}$ and $\ell=\lceil\log_2|uvw|\rceil+1$, where $\mathrm{bin}_\ell(i)$ denotes the ℓ-bit long binary encoding of the integer $i\in[2^{\ell-1}]$ (i.e., the encoding is padded with zeros to a total length of ℓ). The reduction maps $\langle M,x,z\rangle$ to the string $\mathrm{bin}_\ell(|x|)\cdot\mathrm{bin}_\ell(|M|)\cdot Mxz$, where $\ell=\lceil\log_2(|M|+|x|+|z|)\rceil+1$. Noting that this reduction satisfies all conditions of Definition 10.16, the proposition follows. □

10.2.2 Ramifications

In our opinion, the most problematic aspect of the theory described in Section 10.2.1 is the choice to focus on simple probability ensembles, which in turn restricts "distributional versions of NP" to the class dist$\mathcal{NP}$ (Definition 10.15). As indicated §10.2.1.1, this restriction raises two opposite concerns (i.e., that dist$\mathcal{NP}$ is either too wide or too narrow).[22] Here we address the concern that the class of simple probability ensembles is too restricted, and consequently that the conjecture

[22] On one hand, if the definition of dist$\mathcal{NP}$ were too liberal then membership in dist$\mathcal{NP}$ would mean less than one may desire. On the other hand, if dist$\mathcal{NP}$ were too restricted then the conjecture that dist$\mathcal{NP}$ contains hard problems would have been very questionable.

dist$\mathcal{NP} \not\subseteq$ tpc$\mathcal{BPP}$ is too strong (which would mean that dist$\mathcal{NP}$-completeness is a weak evidence for typical-case hardness). An appealing extension of the class of simple probability ensembles is presented in §10.2.2.2, yielding an corresponding extension of dist$\mathcal{NP}$, and it is shown that if this extension of dist$\mathcal{NP}$ is not contained in tpc$\mathcal{BPP}$ then dist$\mathcal{NP}$ itself is not contained in tpc$\mathcal{BPP}$. Consequently, dist$\mathcal{NP}$-complete problems enjoy the benefit of both being in the more restricted class (i.e., dist$\mathcal{NP}$) and being hard as long as some problems in the extended class is hard.

Another extension appears in §10.2.2.1, where we extend the treatment from decision problems to search problems. This extension is motivated by the realization that search problem are actually of greater importance to real-life applications (cf. Section 2.1.1), and hence a theory motivated by real-life applications must address such problems, as we do next.

Prerequisites: For the technical development of §10.2.2.1, we assume familiarity with the notion of unique solution and results regarding it as presented in Section 6.2.3. For the technical development of §10.2.2.2, we assume familiarity with hashing functions as presented in Appendix D.2.

10.2.2.1 Search versus Decision

Indeed, as in the case of worst-case complexity, search problems are at least as important as decision problems. Thus, an average-case treatment of search problems is indeed called for. We first present distributional versions of $\mathcal{PF}$ and $\mathcal{PC}$ (cf. Section 2.1.1), following the underlying principles of the definitions of tpc$\mathcal{P}$ and dist$\mathcal{NP}$.

Definition 10.22 (the classes tpc$\mathcal{PF}$ and dist$\mathcal{PC}$): *As in Section 2.1.1, we consider only polynomially bounded search problems; that is, binary relations $R \subseteq \{0,1\}^* \times \{0,1\}^*$ such that for some polynomial q it holds that $(x,y) \in R$ implies $|y| \leq q(|x|)$. Recall that $R(x) \stackrel{\rm def}{=} \{y : (x,y) \in R\}$ and $S_R \stackrel{\rm def}{=} \{x : R(x) \neq \emptyset\}$.*

- *A* distributional search problem *consists of a polynomially bounded search problem coupled with a probability ensemble.*
- *The class* tpc$\mathcal{PF}$ *consists of all distributional search problems that are typically solvable in polynomial-time. That is, $(R, \{X_n\}_{n\in\mathbb{N}}) \in$ tpc$\mathcal{PF}$ if there exists an algorithm A and a polynomial p such that the probability that on input X_n algorithm A either errs or runs more that $p(n)$ steps is negligible, where A errs on $x \in S_R$ if $A(x) \notin R(x)$ and errs on $x \notin S_R$ if $A(x) \neq \perp$.*
- *A distributional search problem (R, X) is in* dist$\mathcal{PC}$ *if $R \in \mathcal{PC}$ and X is simple* (as in Definition 10.15).

Likewise, the class tpc$\mathcal{BPPF}$ consists of all distributional search problems that are typically solvable in *probabilistic* polynomial-time (cf., Definition 10.20). The definitions of *reductions among distributional problems*, presented in the context of decision problem, extend to search problems.

Fortunately, as in the context of worst-case complexity, the study of distributional search problems "reduces" to the study of distributional decision problems.

Theorem 10.23 (reducing search to decision): dist$\mathcal{PC} \subseteq$ tpc$\mathcal{BPPF}$ *if and only if* dist$\mathcal{NP} \subseteq$ tpc$\mathcal{BPP}$. *Furthermore, every problem in* dist$\mathcal{NP}$ *is reducible to some problem in* dist$\mathcal{PC}$*, and every problem in* dist$\mathcal{PC}$ *is* randomly *reducible to some problem in* dist$\mathcal{NP}$*.*

Proof Sketch: The furthermore part is analogous to the actual contents of the proof of Theorem 2.6 (see also Step 1 in the proof of Theorem 2.15). Indeed the reduction of $\mathcal{NP}$ to $\mathcal{PC}$ presented in the proof of Theorem 2.6 extends to the current context. Specifically, for any $S \in \mathcal{NP}$, we consider a relation $R \in \mathcal{PC}$ such that $S = \{x : R(x) \neq \emptyset\}$, and note that, for any probability ensemble X, the identity transformation reduces (S, X) to (R, X).

A difficulty arises in the opposite direction. Recall that in the proof of Theorem 2.6 we reduced the search problem of $R \in \mathcal{PC}$ to deciding membership in $S'_R \stackrel{\rm def}{=} \{\langle x, y'\rangle : \exists y'' \text{ s.t. } (x, y'y'') \in R\} \in \mathcal{NP}$. The difficulty encountered here is that, on input x, this reduction makes queries of the form $\langle x, y'\rangle$, where y' is a prefix of some string in $R(x)$. These queries may induce a distribution that is not dominated by any simple distribution. Thus, we seek an alternative reduction.

As a warm-up, let us assume for a moment that R has unique solutions (in the sense of Definition 6.28); that is, for every x it holds that $|R(x)| \leq 1$. In this case we may easily reduce the search problem of $R \in \mathcal{PC}$ to deciding membership in $S''_R \in \mathcal{NP}$, *where* $\langle x, i, \sigma\rangle \in S''_R$ *if and only if* $R(x)$ *contains a string in which the* $i^{\rm th}$ *bit equals* σ. Specifically, on input x, the reduction issues the queries $\langle x, i, \sigma\rangle$, where $i \in [\ell]$ (with $\ell = \text{poly}(|x|)$) and $\sigma \in \{0, 1\}$, which allows for determining the single string in the set $R(x) \subseteq \{0, 1\}^\ell$ (whenever $|R(x)| = 1$). The point is that this reduction can be used to reduce any $(R, X) \in$ dist$\mathcal{PC}$ (having unique solutions) to $(S''_R, X'') \in$ dist$\mathcal{NP}$, *where* X'' *equally distributes the probability mass of* x (under X) *to all the tuples* $\langle x, i, \sigma\rangle$*; that is, for every* $i \in [\ell]$ *and* $\sigma \in \{0, 1\}$*, it holds that* $\Pr[X''_{|\langle x,i,\sigma\rangle|} = \langle x, i, \sigma\rangle]$ *equals* $\Pr[X_{|x|} = x]/2\ell$.

Unfortunately, in the general case, R may not have unique solutions. Nevertheless, applying the main idea that underlies the proof of Theorem 6.29, this difficulty can be overcome. We first note that the foregoing mapping of instances of the distributional problem $(R, X) \in$ dist$\mathcal{PC}$ to instances of $(S''_R, X'') \in$ dist$\mathcal{NP}$ satisfies the efficiency and domination conditions even in the case that R does not have unique solutions. What may possibly fail (in the general case) is the validity condition (i.e., if $|R(x)| > 1$ then we may fail to recover any element of $R(x)$).

Recall that the main part of the proof of Theorem 6.29 is a randomized reduction that maps instances of R to triples of the form (x, m, h) such that m is uniformly distributed in $[\ell]$ and h is uniformly distributed in a family of hashing function H^m_ℓ, where $\ell = \text{poly}(|x|)$ and H^m_ℓ is as in Appendix D.2. Furthermore, if $R(x) \neq \emptyset$ then, with probability $\Omega(1/\ell)$ over the choices of $m \in [\ell]$ and $h \in H^m_\ell$, there exists a unique $y \in R(x)$ such that $h(y) = 0^m$. Defining $R'(x, m, h) \stackrel{\rm def}{=} \{y \in R : h(y) = 0^m\}$, this yields a randomized reduction of the search problem of R to the search problem of R' such that with noticeable probability[23] the reduction maps instances that have solutions to instances having a unique solution. Furthermore, this reduction can be used to reduce any $(R,X) \in \text{dist}\mathcal{PC}$ to $(R',X') \in \text{dist}\mathcal{PC}$, *where X' distributes the probability mass of x* (under X) *to all the triples (x,m,h) such that for every $m \in [\ell]$ and $h \in H_\ell^m$ it holds that* $\Pr[X'_{|(x,m,h)|} = (x,m,h)]$ *equals* $\Pr[X_{|x|} = x]/(\ell \cdot |H_\ell^m|)$. (Note that with a suitable encoding, X' is indeed simple.)

The theorem follows by combining the two aforementioned reductions. That is, we first apply the randomized reduction of (R,X) to (R',X'), and next reduce the resulting instance to an instance of the corresponding decision problem $(S''_{R'},X'')$, where X'' is obtained by modifying X' (rather than X). The combined randomized mapping satisfies the efficiency and domination conditions, and is valid with noticeable probability. The error probability can be made negligible by straightforward amplification (see Exercise 10.24). □

10.2.2.2 Simple versus sampleable distributions

Recall that the definition of simple probability ensembles (underlying Definition 10.15) requires that the accumulating distribution function is polynomial-time computable. Recall that $\mu : \{0,1\}^* \to [0,1]$ is called the accumulating distribution function of $X = \{X_n\}_{n\in\mathbb{N}}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$ it holds that $\mu(x) \stackrel{\text{def}}{=} \Pr[X_n \le x]$, where the inequality refers to the standard lexicographic order of n-bit strings.

As argued in §10.2.1.1, the requirement that the accumulating distribution function is polynomial-time computable imposes severe restrictions on the set of admissible ensembles. Furthermore, it seems that these simple ensembles are indeed "simple" in some intuitive sense, and that they represent a reasonable (alas disputable) model of distributions that may occur in practice. Still, in light of the fear that this model is too restrictive (and consequently that dist$\mathcal{NP}$-hardness is weak evidence for typical-case hardness), we seek a maximalistic model of distributions that may occur in practice. Such a model is provided by the notion of polynomial-time sampleable ensembles (underlying Definition 10.24). Our maximality thesis is based on the belief that the real world should be modeled as a *feasible* randomized process (rather than as an arbitrary process). This belief implies that all objects encountered in the world may be viewed as samples generated by a feasible randomized process.

Definition 10.24 (sampleable ensembles and the class samp$\mathcal{NP}$): *We say that a probability ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is* (polynomial-time) sampleable *if there exists a probabilistic polynomial-time algorithm A such that for every $x \in \{0,1\}^*$ it holds that* $\Pr[A(1^{|x|}) = x] = \Pr[X_{|x|} = x]$. *We denote by* samp$\mathcal{NP}$ *the class of distributional problems consisting of decision problems in $\mathcal{NP}$ coupled with sampleable probability ensembles.*

[23] Recall that the probability of an event is said to be noticeable (in a relevant parameter) if it is greater than the reciprocal of some positive polynomial. In the context of randomized reductions, the relevant parameter is the length of the input to the reduction.

We first note that all simple probability ensembles are indeed sampleable (see Exercise 10.25), and thus dist$\mathcal{NP} \subseteq$ samp$\mathcal{NP}$. On the other hand, there exist sampleable probability ensembles that do not seem simple (see Exercise 10.26).

Extending the scope of distributional problems (from dist$\mathcal{NP}$ to samp$\mathcal{NP}$) allows proving that every NP-complete problem has a distributional version in samp$\mathcal{NP}$ that is dist$\mathcal{NP}$-hard (see Exercise 10.27). Furthermore, it is possible to prove that all natural NP-complete problem have distributional versions that are samp$\mathcal{NP}$-complete.

Theorem 10.25 (samp$\mathcal{NP}$-completeness): *Suppose that $S \in \mathcal{NP}$ and that every set in $\mathcal{NP}$ is reducible to S by a Karp-reduction that does not shrink the input. Then there exists a polynomial-time sampleable ensemble X such that any problem in* samp$\mathcal{NP}$ *is reducible to* (S, X)

The proof of Theorem 10.25 is based on the observation that *there exists a polynomial-time sampleable ensemble that dominates all polynomial-time sampleable ensembles.* The existence of this ensemble is based on the notion of a universal (sampling) machine. For further details see Exercise 10.28.

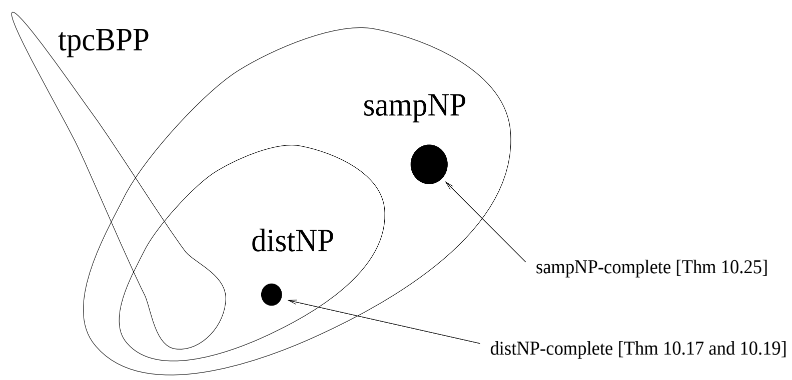


Figure 10.1: Two types of average-case completeness

Theorem 10.25 establishes a rich theory of samp$\mathcal{NP}$-completeness, but does not relate this theory to the previously presented theory of dist$\mathcal{NP}$-completeness (see Figure 10.1). This is done in the next theorem, which asserts that the existence of typically hard problems in samp$\mathcal{NP}$ implies their existence in dist$\mathcal{NP}$.

Theorem 10.26 (samp$\mathcal{NP}$-completeness versus dist$\mathcal{NP}$-completeness): *If* samp$\mathcal{NP}$ *is not contained in* tpc$\mathcal{BPP}$ *then* dist$\mathcal{NP}$ *is not contained in* tpc$\mathcal{BPP}$.

Thus, the two "typical-case complexity" versions of the P-vs-NP Question are equivalent. That is, if some "sampleable distribution" versions of NP are not typically feasible then some "simple distribution" versions of NP are not typically feasible. In particular, if samp$\mathcal{NP}$-complete problems are not in tpc$\mathcal{BPP}$ then dist$\mathcal{NP}$-complete problems are not in tpc$\mathcal{BPP}$.

The foregoing assertions would all follow if samp$\mathcal{NP}$ were (randomly) reducible to dist$\mathcal{NP}$ (i.e., if every problem in samp$\mathcal{NP}$ were reducible (under a randomized version of Definition 10.16) to some problem in dist$\mathcal{NP}$); but, unfortunately, we do not know whether such reductions exist. Yet, underlying the proof of Theorem 10.26 is a more liberal notion of a reduction among distributional problem.

Proof Sketch: We shall prove that if dist$\mathcal{NP}$ is contained in tpc$\mathcal{BPP}$ then the same holds for samp$\mathcal{NP}$ (i.e., samp$\mathcal{NP}$ is contained in tpc$\mathcal{BPP}$). Actually, we shall show that if dist$\mathcal{PC}$ is contained in tpc$\mathcal{BPPF}$ then the sampleable version of dist$\mathcal{PC}$, denoted samp$\mathcal{PC}$, is contained in tpc$\mathcal{BPPF}$ (and refer to Exercise 10.29). Specifically, we shall show that under a relaxed notion of a randomized reduction, every problem in samp$\mathcal{PC}$ is reduced to some problem in dist$\mathcal{PC}$. Loosely speaking, this relaxed notion (of a randomized reduction) only requires that the validity and domination conditions (of Definition 10.16 (when adapted to randomized reductions)) hold with respect to a noticeable fraction of the probability space of the reduction.[24] We start by formulating this notion, when referring to distributional *search* problems.

Teaching note: The following proof is quite involved and is better left for advanced reading. Its main idea is related in one of the central ideas underlying the currently known proof of Theorem 8.11. This fact as well as numerous other applications of this idea, provide a good motivation for getting familiar with this idea.

Definition: A relaxed reduction of the distributional problem (R, X) to the distributional problem (T, Y) is a probabilistic polynomial-time oracle machine M that satisfies the following conditions:

Notation: For every $x \in \{0,1\}^*$, we denote by $m(|x|) = \text{poly}(|x|)$ the number of internal coin tosses of M on input x, and denote by $M^T(x,r)$ the execution of M on input x, internal coins $r \in \{0,1\}^m$, and oracle access to T.

Validity: For some noticeable function $\rho : \mathbb{N} \to [0,1]$ (i.e., $\rho(n) > 1/\text{poly}(n)$) it holds that for every $x \in \{0,1\}^*$, there exists a set $\Omega_x \subseteq \{0,1\}^{m(|x|)}$ of size at least $\rho(|x|) \cdot 2^{m(|x|)}$ such that for every $r \in \Omega_x$ the reduction yields a correct answer (i.e., $M^T(x,r) \in R(x)$ if $R(x) \neq \emptyset$ and $M^T(x,r) = \perp$ otherwise).

Domination: There exists a positive polynomial p such that, for every $y \in \{0,1\}^*$ and every $n \in \mathbb{N}$, it holds that

$$\Pr[Q'(X_n) \ni y] \leq p(|y|) \cdot \Pr[Y_{|y|} = y], \tag{10.6}$$

where $Q'(x)$ is a random variable, defined over the set Ω_x (of the validity condition), representing the set of queries made by M on input x and oracle

[24]We warn that the existence of such a relaxed reduction between two specific distributional problems does not necessarily imply the existence of a corresponding (standard average-case) reduction. Specifically, although standard validity can be guaranteed (for problems in $\mathcal{PC}$) by repeated invocations of the reduction, such a process will *not* redeem the violation of the standard domination condition.

access to T. That is, $Q'(x)$ is defined by uniformly selecting $r \in \Omega_x$ and considering the set of queries made by M on input x, internal coins r, and oracle access to T. (In addition, as in Definition 10.16, we also require that the reduction does not make too short queries.)

The reader may verify that this relaxed notion of a reduction preserves typical feasibility; that is, for $R \in \mathcal{PC}$, if there exists a relaxed reduction of (R, X) to (T, Y) and (T, Y) is in tpc$\mathcal{BPPF}$ then (R, X) is in tpc$\mathcal{BPPF}$. The key observation is that the analysis may discard the case that, on input x, the reduction selects coins not in Ω_x. Indeed, the queries made in that case may be untypical and the answers received may be wrong, but this is immaterial. What matter is that, on input x, with noticeable probability the reduction selects coins in Ω_x, and produces "typical with respect to Y" queries (by virtue of the relaxed domination condition). Such typical queries are answered correctly by the algorithm that typically solves (T, Y), and if x has a solution then these answers yield a correct solution to x (by virtue of the relaxed validity condition). Thus, if x has a solution then with noticeable probability the reduction outputs a correct solution. On the other hand, the reduction never outputs a wrong solution (even when using coins not in Ω_x), because incorrect solutions are detected by relying on $R \in \mathcal{PC}$.

Our goal is presenting, for every $(R, X) \in$ samp$\mathcal{PC}$, a relaxed reduction of (R, X) to a related problem $(R', X') \in$ dist$\mathcal{PC}$, where (as usual) $X = \{X_n\}_{n\in\mathbb{N}}$ and $X' = \{X'_n\}_{n\in\mathbb{N}}$.

An oversimplified case: For starters, *suppose that X_n is uniformly distributed on some set $S_n \subseteq \{0,1\}^n$ and that there is a polynomial-time computable and invertible mapping μ of S_n to $\{0,1\}^{\ell(n)}$, where $\ell(n) = \log_2 |S_n|$.* Then, mapping x to $1^{|x|-\ell(|x|)}0\mu(x)$, we obtain a reduction of (R, X) to (R', X'), where X'_{n+1} is uniform over $\{1^{n-\ell(n)}0v : v \in \{0,1\}^{\ell(n)}\}$ and $R'(1^{n-\ell(n)}0v) = R(\mu^{-1}(v))$ (or, equivalently, $R(x) = R'(1^{|x|-\ell(|x|)}0\mu(x))$). Note that X' is a simple ensemble and $R' \in \mathcal{PC}$; hence, $(R', X') \in$ dist$\mathcal{PC}$. Also note that the foregoing mapping is indeed a valid reduction (i.e., it satisfies the efficiency, validity, and domination conditions). Thus, (R, X) is reduced to a problem in dist$\mathcal{PC}$ (and indeed the relaxation was not used here).

A simple but more instructive case: Next, we drop the assumption that there is a polynomial-time computable and invertible mapping μ of S_n to $\{0,1\}^{\ell(n)}$, but maintain the assumption that X_n is uniform on some set $S_n \subseteq \{0,1\}^n$ and assume that $|S_n| = 2^{\ell(n)}$ is easily computable (from n). In this case, we may map $x \in \{0,1\}^n$ to its image under a suitable randomly chosen hashing function h, which in particular maps n-bit strings to $\ell(n)$-bit strings. That is, we randomly map x to $(h, 1^{n-\ell(n)}0h(x))$, where h is uniformly selected in a set $H_n^{\ell(n)}$ of suitable hash functions (see Appendix D.2). This calls for redefining R' such that $R'(h, 1^{n-\ell(n)}0v)$ corresponds to the preimages of v under h that are in S_n. Assuming that h is a 1-1 mapping of S_n to $\{0,1\}^{\ell(n)}$, we may define $R'(h, 1^{n-\ell(n)}0v) = R(x)$ where x is the unique string satisfying $x \in S_n$ and $h(x) = v$, where the condition $x \in S_n$ may be *verified by providing the internal coins of the sampling procedure that generate x*. Denoting the sampling procedure of X by S, and letting $S(1^n, r)$ denote the output of S on input 1^n and internal coins r, we actually redefine R' as

$$R'(h, 1^{n-\ell(n)}0v) = \{\langle r,y\rangle : h(S(1^n,r))=v \wedge y \in R(S(1^n,r))\}. \tag{10.7}$$

We note that $\langle r,y\rangle \in R'(h,1^{|x|-\ell(|x|)}0h(x))$ yields a solution $y \in R(x)$ if $S(1^{|x|},r) = x$, but otherwise "all bets are off" (as y will be a solution for $S(1^{|x|},r) \neq x$). Now, although typically h will not be a 1-1 mapping of S_n to $\{0,1\}^{\ell(n)}$, it is the case that *for each $x \in S_n$, with constant probability over the choice of h, it holds that $h(x)$ has a unique preimage in S_n under h.* (See the proof of Theorem 6.29.) In this case $\langle r,y\rangle \in R'(h,1^{|x|-\ell(|x|)}0h(x))$ implies $S(1^{|x|},r) = x$ (which, in turn, implies $y \in R(x)$). We claim that *the randomized mapping of x to $(h, 1^{n-\ell(n)}0h(x))$, where h is uniformly selected in $H_{|x|}^{\ell(|x|)}$, yields a relaxed reduction of (R,X) to (R',X'), where $X'_{n'}$ is uniform over $H_n^{\ell(n)} \times \{1^{n-\ell(n)}0v : v \in \{0,1\}^{\ell(n)}\}$.* Needless to say, the claim refers to the reduction that makes the query $(h, 1^{n-\ell(n)}0h(x))$ and returns y if the oracle answer equals $\langle r,y\rangle$ and $y \in R(x)$.

The claim is proved by considering the set Ω_x of choices of $h \in H_{|x|}^{\ell(|x|)}$ for which $x \in S_n$ is the only preimage of $h(x)$ under h that resides in S_n (i.e., $|\{x' \in S_n : h(x') = h(x)\}| = 1$). In this case (i.e., $h \in \Omega_x$) it holds that $\langle r,y\rangle \in R'(h,1^{|x|-\ell(|x|)}0h(x))$ implies that $S(1^{|x|},r) = x$ and $y \in R(x)$, and the (relaxed) validity condition follows. The (relaxed) domination condition follows by noting that $\mathsf{Pr}[X_n = x] \approx 2^{-\ell(|x|)}$, that x is mapped to $(h,1^{|x|-\ell(|x|)}0h(x))$ with probability $1/|H_{|x|}^{\ell(|x|)}|$, and that x is the only preimage of $(h,1^{|x|-\ell(|x|)}0h(x))$ under the mapping (among $x' \in S_n$ such that $\Omega_{x'} \ni h$).

Before going any further, let us highlight the importance of hashing X_n to $\ell(n)$-bit strings. On one hand, this mapping is "sufficiently" one-to-one, and thus (with constant probability) the solution provided for the hashed instance (i.e., $h(x)$) yield a solution for the original instance (i.e., x). This guarantees the validity of the reduction. On the other hand, for a typical h, the mapping of X_n to $h(X_n)$ covers the relevant range almost uniformly. This guarantees that the reduction satisfies the domination condition. Note that these two phenomena impose conflicting requirements that are both met at the correct value of ℓ; that is, the one-to-one condition requires $\ell(n) \geq \log_2|S_n|$, whereas an almost uniform cover requires $\ell(n) \leq \log_2|S_n|$. Also note that $\ell(n) = \log_2(1/\mathsf{Pr}[X_n = x])$ for every x in the support of X_n; the latter quantity will be in our focus in the general case.

The general case: Finally, get rid of the assumption that X_n is *uniformly distributed* over some subset of $\{0,1\}^n$. All that we know is that there exists a probabilistic polynomial-time ("sampling") algorithm S such that $S(1^n)$ is distributed identically to X_n. In this (general) case, we map instances of (R,X) according to their probability mass such that x is mapped to an instance (of R') that consists of $(h,h(x))$ and additional information, where h is a random hash function mapping n-bit long strings to ℓ_x-bit long strings such that

$$\ell_x \stackrel{\text{def}}{=} \lceil \log_2(1/\mathsf{Pr}[X_{|x|}=x]) \rceil. \tag{10.8}$$

Since (in the general case) there may be more than 2^{ℓ_x} strings in the support of X_n, we need to augment the reduced instance in order to ensure that it is uniquely associated with x. The basic idea is augmenting the mapping of x to $(h, h(x))$ with additional information that restricts X_n to strings that occur with probability at least $2^{-\ell_x}$. Indeed, when X_n is restricted in this way, the value of $h(X_n)$ uniquely determines X_n.

Let $q(n)$ denote the randomness complexity of S and $S(1^n, r)$ denote the output of S on input 1^n and internal coin tosses $r \in \{0,1\}^{q(n)}$. Then, we randomly map x to $(h, h(x), h', v')$, where $h : \{0,1\}^{|x|} \to \{0,1\}^{\ell_x}$ and $h' : \{0,1\}^{q(|x|)} \to \{0,1\}^{q(|x|)-\ell_x}$ are random hash functions and $v' \in \{0,1\}^{q(|x|)-\ell_x}$ is uniformly distributed. The instance (h, v, h', v') of the redefined search problem R' has solutions that consists of pairs $\langle r, y\rangle$ such that $h(S(1^n, r)) = v \wedge h'(r) = v'$ and $y \in R(S(1^n, r))$. As we shall see, this augmentation guarantees that, with constant probability (over the choice of h, h', v'), the solutions to the reduced instance $(h, h(x), h', v')$ correspond to the solutions to the original instance x.

The foregoing description assumes that, on input x, we can determine ℓ_x, which is an assumption that cannot be justified. Instead, we select ℓ uniformly in $\{0, 1, ..., q(|x|)\}$, and so with noticeable probability we do select the correct value (i.e., $\Pr[\ell = \ell_x] = 1/(q(|x|) + 1) = 1/\mathrm{poly}(|x|)$). For clarity, we make n and ℓ explicit in the reduced instance. Thus, we randomly map $x \in \{0,1\}^n$ to $(1^n, 1^\ell, h, h(x), h', v') \in \{0,1\}^{n'}$, where $\ell \in \{0, 1, ..., q(n)\}$, $h \in H_n^\ell$, $h' \in H_{q(n)}^{q(n)-\ell}$, and $v' \in \{0,1\}^{q(n)-\ell}$ are uniformly distributed in the corresponding sets.[25] This mapping will be used to reduce (R, X) to (R', X'), where R' and $X' = \{X'_{n'}\}_{n' \in \mathbb{N}}$ are redefined (yet again). Specifically, we let

$$R'(1^n, 1^\ell, h, v, h', v') = \{\langle r, y\rangle : h(S(1^n, r)) = v \wedge h'(r) = v' \wedge y \in R(S(1^n, r))\} \quad (10.9)$$

and $X'_{n'}$ assigns equal probability to each $X_{n',\ell}$ (for $\ell \in \{0, 1, ..., n\}$), where each $X_{n',\ell}$ is isomorphic to the uniform distribution over $H_n^\ell \times \{0,1\}^\ell \times H_{q(n)}^{q(n)-\ell} \times \{0,1\}^{q(n)-\ell}$. Note that indeed $(R', X') \in \mathrm{dist}\mathcal{PC}$.

The aforementioned randomized mapping is analyzed by considering the correct choice for ℓ; that is, on input x, we focus on the choice $\ell = \ell_x$. Under this conditioning (as we shall show), *with constant probability over the choice of* h, h' *and* v', *the instance* x *is the only value in the support of* X_n *that is mapped to* $(1^n, 1^{\ell_x}, h, h(x), h', v')$ *and satisfies* $\{r : h(S(1^n, r)) = h(x) \wedge h'(r) = v'\} \neq \emptyset$. It follows that (for such h, h' and v') any solution $\langle r, y\rangle \in R'(1^n, 1^{\ell_x}, h, h(x), h', v')$ satisfies $S(1^n, r) = x$ and thus $y \in R(x)$, which means that the (relaxed) validity condition is satisfied. The (relaxed) domination condition is satisfied too, because (conditioned on $\ell = \ell_x$ and for such h, h', v') the probability that X_n is mapped to $(1^n, 1^{\ell_x}, h, h(x), h', v')$ approximately equals $\Pr[X'_{n',\ell_x} = (1^n, 1^{\ell_x}, h, h(x), h', v')]$.

We now turn to analyze the probability, over the choice of h, h' and v', that the instance x is the only value in the support of X_n that is mapped to $(1^n, 1^{\ell_x}, h, h(x), h', v')$ and satisfies $\{r : h(S(1^n, r)) = h(x) \wedge h'(r) = v'\} \neq \emptyset$. Firstly, we note that

[25] As in other places, a suitable encoding will be used such that the reduction maps strings of the same length to strings of the same length (i.e., n-bit string are mapped to n'-bit strings, for $n' = \mathrm{poly}(n)$). For example, we may encode $\langle 1^n, 1^\ell, h, h(x), h', v'\rangle$ as $1^n 0 1^\ell 0 1^{q(n)-\ell} 0 \langle h\rangle\langle h(x)\rangle\langle h'\rangle\langle v'\rangle$, where each $\langle w\rangle$ denotes an encoding of w by a string of length $(n' - (n + q(n) + 3))/4$.

$|\{r : S(1^n, r) = x\}| \geq 2^{q(n)-\ell_x}$, and thus, with constant probability over the choice of $h' \in H_{q(n)}^{q(n)-\ell_x}$ and $v' \in \{0,1\}^{q(n)-\ell_x}$, there exists r that satisfies $S(1^n, r) = x$ and $h'(r) = v'$. Next, we note that, with constant probability over the choice of $h \in H_n^{\ell_x}$, it holds that x is the only string having probability mass at least $2^{-\ell_x}$ (under X_n) that is mapped to $h(x)$ under h. Finally, we prove that, with constant probability over the choice of $h \in H_n^{\ell_x}$ and $h' \in H_{q(n)}^{q(n)-\ell_x}$ (and even when conditioning on the previous items), the mapping $r \mapsto (h(S(1^n, r)), h'(r))$ maps the set $\{r : \Pr[X_n = S(1^n, r)] \leq 2^{-\ell_x}\}$ to $\{0,1\}^{q(n)}$ in an almost 1-1 manner. Specifically, with constant probability, no other r is mapped to the aforementioned pair $(h(x), v')$. Thus, the claim follows and so does the theorem. □

Reflection. Theorem 10.26 implies that if samp$\mathcal{NP}$ is not contained in tpc$\mathcal{BPP}$ then every dist$\mathcal{NP}$-complete problem is not in tpc$\mathcal{BPP}$. This means that the hardness of some distributional problems that refer to sampleable distributions implies the hardness of some distributional problems that refer to simple distributions. Furthermore, by Proposition 10.21, this implies the hardness of distributional problems that refer to the uniform distribution. Thus, hardness with respect to some distribution in an utmost wide class (which arguably captures all distributions that may occur in practice) implies hardness with respect to a single simple distribution (which arguably is the simplest one).

Relation to one-way functions. We note that the existence of one-way functions (see Section 7.1) implies the existence of problems in samp$\mathcal{PC}$ that are not in tpc$\mathcal{BPPF}$ (which in turn implies the existence of such problems in dist$\mathcal{PC}$). Specifically, for a length-preserving one-way function f, consider the distributional search problem $(R_f, \{f(U_n)\}_{n \in \mathbb{N}})$, where $R_f = \{(f(r), r) : r \in \{0,1\}^*\}$.[26] On the other hand, it is not known whether the existence of a problem in samp$\mathcal{PC} \setminus$ tpc$\mathcal{BPPF}$ implies the existence of one-way functions. In particular, the existence of a problem (R, X) in samp$\mathcal{PC} \setminus$ tpc$\mathcal{BPPF}$ represents the feasibility of generating hard instances for the search problem R, whereas the existence of one-way function represents the feasibility of generating instance-solution pairs such that the instances are hard to solve (see Section 7.1.1). Indeed, the gap refers to whether or not *hard instances can be efficiently generated together with corresponding solutions.* Our world view is thus depicted in Figure 10.2, where lower levels indicate seemingly weaker assumptions.

Chapter Notes

In this chapter, we presented two different approaches to the relaxation of computational problems. The first approach refers to the concept of approximation, while the second approach refers to average-case analysis. We demonstrated that

[26]Note that the distribution $f(U_n)$ is uniform in the special case that f is a permutation over $\{0,1\}^n$.

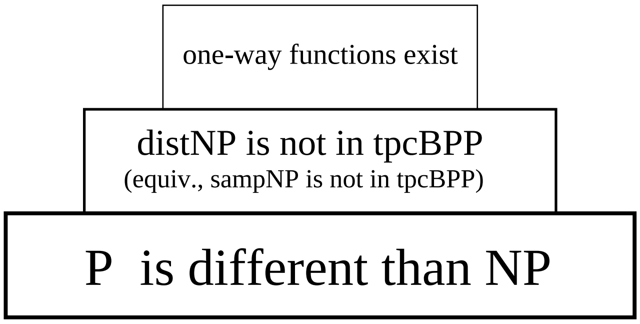


Figure 10.2: Worst-case vs average-case assumptions

various natural notions of approximation can be cast within the standard frameworks, where the framework of promise problems (presented in Section 2.4.1) is the least-standard framework we used (and it suffices for casting gap problems and property testing). In contrast, the study of average-case complexity requires the introduction of a new conceptual framework and addressing of various definitional issues.

A natural question at this point is what have we gained by relaxing the requirements. In the context of approximation, the answer is mixed: in some natural cases we gain a lot (i.e., we obtained feasible relaxations of hard problems), while in other natural cases we gain nothing (i.e., even extreme relaxations remain as intractable as the original version). In the context of average-case complexity, the negative side seems more prevailing (at least in the sense of being more systematic). In particular, assuming the existence of one-way functions, every natural NP-complete problem has a distributional version that is (typical-case) hard, where this version refers to a sampleable ensemble (and, in fact, even to a simple ensemble). Furthermore, in this case, some problems in NP have hard distributional versions that refer to the uniform distribution.

Another difference between the two approaches is that the theory of approximation seems to lack a comprehensive structure, whereas the theory of average-case complexity seems to have a too rigid structure (which seems to foil attempts to present more appealing dist$\mathcal{NP}$-complete problems).

Approximation

The following bibliographic comments are quite laconic and neglect mentioning various important works (including credits for some of the results mentioned in our text). As usual, the interested reader is referred to corresponding surveys.

Search or Optimization. The interest in approximation algorithms increased considerably following the demonstration of the NP-completeness of many natural optimization problems. But, with some exceptions (most notably [172]), the systematic study of the complexity of such problems stalled till the discovery of the "PCP connection" (see Section 9.3.3) by Feige, Goldwasser, Lovász, and Safra [69]. Indeed the relatively "tight" inapproximation results for max-Clique, max-SAT, and the maximization of linear equations, due to Håstad [112, 113], build on previous work regarding PCP and their connection to approximation (cf., e.g., [70, 15, 14, 27, 178]). Specifically, Theorem 10.5 is due to [112], while Theorems 10.8 and 10.9 are due to [113]. The best known inapproximation result for minimum Vertex Cover (see Theorem 10.7) is due to [65], but we doubt it is tight (see, e.g., [137]). Reductions among approximation problems were defined and presented in [172]; see Exercise 10.7, which presents a major technique introduced in [172]. For general texts on approximation algorithms and problems (as discussed in Section 10.1.1), the interested reader is referred to the surveys collected in [118]. A compendium of NP optimization problems is available at [61].

Recall that a different type of approximation problems, which are naturally associated with search problems, refer to approximately counting the number of solutions. These approximation problems were treated in Section 6.2.2 in a rather *ad hoc* manner. We note that a more systematic treatment of approximate counting problems can be obtained by using the definitional framework of Section 10.1.1 (e.g., the notions of gap problems, polynomial-time approximation schemes, etc).

Property testing. The study of property testing was initiated by Rubinfeld and Sudan [188] and re-initiated by Goldreich, Goldwasser, and Ron [93]. While the focus of [188] was on algebraic properties such as low-degree polynomials, the focus of [93] was on graph properties (and Theorem 10.12 is taken from [93]). The model of bounded-degree graphs was introduced in [99] and Theorem 10.13 combines results from [99, 100, 39]. For surveys of the area, the interested reader is referred to [73, 187].

Average-case complexity

The theory of average-case complexity was initiated by Levin [148], who in particular proved Theorem 10.17. In light of the laconic nature of the original text [148], we refer the interested reader to a survey [85], which provides a more detailed exposition of the definitions suggested by Levin as well as a discussion of the considerations underlying these suggestions. (This survey [85] provides also a brief account of further developments.)

As noted in §10.2.1.1, the current text uses a variant of the original definitions. In particular, our definition of "typical-case feasibility" differs from the original definition of "average-case feasibility" in totally discarding exceptional instances and in even allowing the algorithm to fail on them (and not merely run for an excessive amount of time). The alternative definition was suggested by several researchers, and appears as a special case of the general treatment provided in [41].

Turning to §10.2.1.2, we note that while the existence of dist$\mathcal{NP}$-complete problems (cf. Theorem 10.17) was established in Levin's original paper [148], the existence of dist$\mathcal{NP}$-complete versions of all natural NP-complete decision problems (cf. Theorem 10.19) was established more than two decades later in [151].

Section 10.2.2 is based on [28, 123]. Specifically, Theorem 10.23 (or rather the reduction of search to decision) is due to [28] and so is the introduction of the class samp$\mathcal{NP}$. A version of Theorem 10.26 was proven in [123], and our proof follows their ideas, which in turn are closely related to the ideas underlying the proof of Theorem 8.11 (proved in [114]).

Recall that we know of the existence of problems in dist$\mathcal{NP}$ that are hard provided samp$\mathcal{NP}$ contains hard problems. However, these distributional problems do not seem very natural (i.e., they either refer to somewhat generic decision problems such as $S_{\mathtt{u}}$ or to somewhat contrived probability ensembles (cf. Theorem 10.19)). The presentation of dist$\mathcal{NP}$-complete problems that combine a more natural decision problem (like `SAT` or `Clique`) with a natural probability ensemble is an open problem.

Exercises

Exercise 10.1 (general TSP) For any function g, prove that the following approximation problem is NP-Hard. Given a general TSP instance I, represented by a symmetric matrix of pairwise distances, the task is finding a tour of length that is at most a factor $g(I)$ of the minimum. Show that the result holds with $g(I) = \exp(\text{poly}(|I|))$ and for instances in which all distances are positive,

Guideline: By reduction from Hamiltonian path. Specifically, reduce the instance $G = ([n], E)$ to an n-by-n distance matrix $D = (d_{i,j})_{i,j\in[n]}$ such that $d_{i,j} = \exp(\text{poly}(n))$ if $\{i,j\} \in E$ and $d_{i,j} = 1$.

Exercise 10.2 (TSP with triangle inequalities) Provide a polynomial-time 2-factor approximation for the special case of TSP in which the distances satisfy the triangle inequality.

Guideline: First note that the length of any tour is lower-bounded by the weight of a minimum spanning tree in the corresponding weighted graph. Next note that such a tree yields a tour (of length twice the weight of this tree) that may visit some points several times. The triangle inequality guarantees that the tour does not become longer by "shortcuts" that eliminate multiple visits at the same point.

Exercise 10.3 (a weak version of Theorem 10.5) Using Theorem 9.16 prove that, for some constants $0 < a < b < 1$ when setting $L(N) = N^b$ and $s(N) = N^a$, it holds that $\mathtt{gapClique}_{L,s}$ is NP-hard.

Guideline: Starting with Theorem 9.16, apply the Expander Random Walk Generator (of Proposition 8.29) in order to derive a PCP system with logarithmic randomness and query complexities that accepts no-instances of length n with probability at most $1/n$. The claim follows by applying the FGLSS-reduction (of Exercise 9.14), while noting that x is reduced to a graph of size poly$(|x|)$ such that the gap between yes- and no-instances is at least a factor of $|x|$.

Exercise 10.4 (a weak version of Theorem 10.7) Using Theorem 9.16 prove that, for some constants $0 < s < L < 1$, the problem $\mathtt{gapVC}_{s,L}$ is NP-hard.

Guideline: Note that combining Theorem 9.16 and Exercise 9.14 implies that for some constants $b < 1$ it holds that $\texttt{gapClique}_{L,s}$ is NP-hard, where $L(N) = b \cdot N$ and $s(N) = (b/2) \cdot N$. The claim follows using the relations between cliques, independent sets, and vertex covers.

Exercise 10.5 (a weak version of Theorem 10.9) Using Theorem 9.16 prove that, for some constants $0.5 < s < L < 1$, the problem $\texttt{gapLin}_{L,s}$ is NP-hard.

Guideline: Recall that by Theorems 9.16 and 9.21, the gap problem $\texttt{gapSAT}^3_\varepsilon$ is NP-Hard. Note that the result holds even if we restrict the instances to have exactly three (not necessarily different) literals in each clause. Applying the reduction of Exercise 2.24, note that, for any assignment τ, a clause that is satisfied by τ is mapped to seven equations of which exactly three are violated by τ, whereas a clause that is not satisfied by τ is mapped to seven equations that are all violated by τ.

Exercise 10.6 (natural inapproximability without the PCP Theorem) In contrast to the inapproximability results reviewed in §10.1.1.2, the NP-completeness of the following gap problem can be established (rather easily) without referring to the PCP Theorem. The instances of this problem are systems of quadratic equations over GF(2) (as in Exercise 2.25), yes-instances are systems that have a solution, and no-instances are systems for which any assignment violates at least one third of the equations.

Guideline: By Exercise 2.25, when given such a quadratic system, it is NP-hard to determine whether or not there exists an assignment that satisfies all the equations. Using an adequate small-bias generator (cf. Section 8.5.2), present an amplifying reduction (cf. Section 9.3.3) of the foregoing problem to itself. Specifically, if the input system has m equations then we use a generator that defines a sample space of poly(m) many m-bit strings, and consider the corresponding linear combinations of the input equations. Note that it suffices to bound the bias of the generator by $1/6$, whereas using an ε-biased generator yields an analogous result with $1/3$ replaced by $0.5 - \varepsilon$.

Exercise 10.7 (enforcing multi-way equalities via expanders) The aim of this exercise is presenting a major technique of Papadimitriou and Yannakakis [172], which is useful for designing reductions among approximation problems. Recalling that $\texttt{gapSAT}^3_{0.1}$ is NP-hard, our goal is proving NP-hard of the following gap problem, denoted $\texttt{gapSAT}^{3,c}_\varepsilon$, which is a special case of $\texttt{gapSAT}^3_\varepsilon$. Specifically, the instances are restricted to 3CNF formulae with each variable appearing in at most c clauses, where c (as ε) is a fixed constant. Note that the standard reduction of 3SAT to the corresponding special case (see proof of Proposition 2.22) does not preserve an approximation gap.[27] The idea is enforcing equality of the values assigned to the

[27] Recall that in this reduction each occurrence of each Boolean variable is replaced by a new copy of this variable, and clauses are added for enforcing the assignment of the same value to all these copies. Specifically, the m occurrence of variable z are replaced by the variables $z^{(1)}, ..., z^{(m)}$, while adding the clauses $z^{(i)} \vee \neg z^{(i+1)}$ and $z^{(i+1)} \vee \neg z^{(i)}$ (for $i = 1, ..., m-1$). The problem is that almost all clauses of the reduced formula may be satisfied by an assignment in which half of the copies of each variable are assigned one value and the rest are assigned an opposite value. That is, an assignment in which $z^{(1)} = \cdots = z^{(i)} \neq z^{(i+1)} = \cdots = z^{(m)}$ violates only one of the

auxiliary variables (i.e., the copies of each original variable) by introducing equality constraints only for pairs of variables that correspond to edges of an expander graph (see Appendix E.2). For example, we enforce equality among the values of $z^{(1)},...,z^{(m)}$ by adding the clauses $z^{(i)} \vee \neg z^{(j)}$ for every $\{i,j\} \in E$, where E is the set of edges of am m-vertex expander graph. Prove that, for some constants c and $\varepsilon > 0$, the corresponding mapping reduces $\texttt{gapSAT}^3_{0.1}$ to $\texttt{gapSAT}^{3,c}_{\varepsilon}$.

Guideline: Using d-regular expanders, we map 3CNF to instances in which each variable appears in at most $2d+1$ clauses. Note that the number of added clauses is linearly related to the number of original clauses. Clearly, if the original formula is satisfiable then so is the reduced one. On the other hand, consider an arbitrary assignment τ' to the reduced formula ϕ' (i.e., the formula obtained by mapping ϕ). For each original variable z, if τ' assigns the same value to almost all copies of z then we consider the corresponding assignment in ϕ. Otherwise, by virtue of the added clauses, τ' does not satisfy a constant fraction of the clauses containing a copy of z.

Exercise 10.8 (deciding majority requires linear time) Prove that deciding majority requires linear-time even in a direct access model and when using a randomized algorithm that may err with probability at most 1/3.

Guideline: Consider the problem of distinguishing X_n from Y_n, where X_n (resp., Y_n) is uniformly distributed over the set of n-bit strings having exactly $\lfloor n/2 \rfloor$ (resp., $\lfloor n/2 \rfloor + 1$) ones. For any fixed set $I \subset [n]$, denote the projection of X_n (resp., Y_n) on I by X'_n (resp., Y'_n). Prove that the statistical difference between X'_n and Y'_n is bounded by $O(|I|/n)$. Note that the argument needs to be extended to the case that the examined locations are selected adaptively.

Exercise 10.9 (testing majority in polylogarithmic time) Show that testing majority (with respect to δ) can be done in polylogarithmic time by probing the input at a constant number of randomly selected locations.

Exercise 10.10 (on the triviality of some testing problems) Show that the following sets are trivially testable in the adjacency matrix representation (i.e., for every $\delta > 0$ and any such set S, there exists a trivial algorithm that distinguishes S from $\Gamma_\delta(S)$).

1. The set of connected graphs.
2. The set of Hamiltonian graphs.
3. The set of Eulerian graphs.

Indeed, show that in each case $\Gamma_\delta(S) = \emptyset$.

Guideline (for Item 3): Note that, *in general*, the fact that the sets S' and S'' are testable within some complexity does *not* imply the same for the set $S' \cap S''$.

auxiliary clauses introduced for enforcing equality among the copies of z. Using an alternative reduction that adds the clauses $z^{(i)} \vee \neg z^{(j)}$ for every $i,j \in [m]$ will not do either, because the number of added clauses may be quadratic in the number of original clauses.

Exercise 10.11 (an equivalent definition of $\mathrm{tpc}\mathcal{P}$**)** Prove that $(S, X) \in \mathrm{tpc}\mathcal{P}$ if and only if there exists a polynomial-time algorithm A such that the probability that $A(X_n)$ errs (in determining membership in S) is a negligible function in n.

Exercise 10.12 ($\mathrm{tpc}\mathcal{P}$ **versus** $\mathcal{P}$ **– Part 1)** Prove that $\mathrm{tpc}\mathcal{P}$ contains a problem (S, X) such that S is not even recursive. Furthermore, use $X = U$.

Guideline: Let $S = \{0^{|x|}x : x \in S'\}$, where S' is an arbitrary (non-recursive) set.

Exercise 10.13 ($\mathrm{tpc}\mathcal{P}$ **versus** $\mathcal{P}$ **– Part 2)** Prove that there exists a distributional problem (S, X) such that $S \notin \mathcal{P}$ and yet there exists an algorithm solving S (correctly on all inputs) in time that is typically polynomial with respect to X. Furthermore, use $X = U$.

Guideline: For any time-constructible function $t : \mathbb{N} \to \mathbb{N}$ that is super-polynomial and sub-exponential, use $S = \{0^{|x|}x : x \in S'\}$ for any $S' \in \mathrm{DTIME}(t) \setminus \mathcal{P}$.

Exercise 10.14 (simple distributions and monotone sampling) We say that a probability ensemble $X = \{X_n\}_{n \in \mathbb{N}}$ is polynomial-time sampleable via a monotone mapping if there exists a polynomial p and a polynomial-time computable function f such that the following two conditions hold:

1. For every n, the random variables $f(U_{p(n)})$ and X_n are identically distributed.
2. For every n and every $r' < r'' \in \{0,1\}^{p(n)}$ it holds that $f(r') \leq f(r'')$, where the inequalities refers to the standard lexicographic order of strings.

Prove that X is simple if and only if it is polynomial-time sampleable via a monotone mapping.

Guideline: Suppose that X is simple, and let p be a polynomial bounding the running-time of the algorithm that on input x outputs $\Pr[X_{|x|} \leq x]$. Consider a mapping, denoted π, of $[0,1]$ to $\{0,1\}^n$ such that $r \in [0,1]$ is mapped to $x \in \{0,1\}^n$ if and only if $r \in [\Pr[X_n < x], \Pr[X_n \leq x])$. The desired function $f : \{0,1\}^{p(n)} \to \{0,1\}^n$ can be obtained from π by considering the binary representation of the numbers in $[0,1]$ (and recalling that the binary representation of $\Pr[X_{|x|} \leq x]$ has length at most $p(|x|)$). Note that f can be computed by binary search, using the fact that X is simple. Turning to the opposite direction, we note that any efficiently computable and monotone mapping $f : \{0,1\}^{p(n)} \to \{0,1\}^n$ can be efficiently inverted by a binary search. Furthermore, similar methods allow for efficiently determining the interval of $p(n)$-bit long strings that are mapped to any given n-bit long string.

Exercise 10.15 (reductions preserve typical polynomial-time solveability) Prove that if the distributional problem (S, X) is reducible to the distributional problem (S', X') and $(S', X') \in \mathrm{tpc}\mathcal{P}$, then (S, X) is in $\mathrm{tpc}\mathcal{P}$.

Guideline: Let B' denote the set of exceptional instances for the distributional problem (S', X'); that is, B' is the set of instances on which the solver in the hypothesis either errs or exceeds the typical running-time. Prove that $\Pr[Q(X_n) \cap B' \neq \emptyset]$ is a negligible function (in n), using both $\Pr[y \in Q(X_n)] \leq p(|y|) \cdot \Pr[X'_{|y|} = y]$ and $|x| \leq p'(|y|)$ for every

$y \in Q(x)$. Specifically, use the latter condition for inferring that $\sum_{y \in B'} \Pr[y \in Q(X_n)]$ equals $\sum_{y \in \{y' \in B' : p'(|y'|) \geq n\}} \Pr[y \in Q(X_n)]$, which guarantees that a negligible function in $|y|$ for any $y \in Q(X_n)$ is negligible in n.

Exercise 10.16 (reductions preserve error-less solveability) In continuation to Exercise 10.15, prove that reductions preserve error-less solveability (i.e., solveability by algorithms that never err and typically run in polynomial-time).

Exercise 10.17 (transitivity of reductions) Prove that reductions among distributional problems (as in Definition 10.16) are transitive.

Guideline: The point is establishing the domination property of the composed reduction. The hypothesis that reductions do not make too short queries is instrumental here.

Exercise 10.18 For any $S \in \mathcal{NP}$ present a simple probability ensemble X such that the generic reduction used in the proof of Theorem 2.18, when applied to (S, X), violates the domination condition with respect to $(S_{\mathtt{u}}, U')$.

Guideline: Consider $X = \{X_n\}_{n \in \mathbb{N}}$ such that X_n is uniform over $\{0^{n/2}x' : x' \in \{0,1\}^{n/2}\}$.

Exercise 10.19 (variants of the Coding Lemma) Prove the following two variants of the Coding Lemma (which is stated in the proof of Theorem 10.17).

1. A variant that refers to any efficiently computable function $\mu : \{0,1\}^* \to [0,1]$ that is monotonically non-decreasing over $\{0,1\}^*$ (i.e., $\mu(x') \leq \mu(x'')$ for any $x' < x'' \in \{0,1\}^*$). That is, unlike in the proof of Theorem 10.17, here it holds that $\mu(0^{n+1}) \geq \mu(1^n)$ for every n.

2. As in Part 1, except that in this variant the function μ is strictly increasing and the compression condition requires that $|C_\mu(x)| \leq \log_2(1/\mu'(x))$ rather than $|C_\mu(x)| \leq 1 + \min\{|x|, \log_2(1/\mu'(x))\}$, where $\mu'(x) \stackrel{\text{def}}{=} \mu(x) - \mu(x-1)$.

In both cases, the proof is less cumbersome than the one presented in the main text.

Exercise 10.20 Prove that for any problem (S, X) in dist$\mathcal{NP}$ there exists a simple probability ensemble Y such that the reduction used in the proof of Theorem 2.18 suffices for reducing (S, X) to $(S_{\mathtt{u}}, Y)$.

Guideline: Consider $Y = \{Y_n\}_{n \in \mathbb{N}}$ such that Y_n assigns to the instance $\langle M, x, 1^t \rangle$ a probability mass proportional to $\pi_x \stackrel{\text{def}}{=} \Pr[X_{|x|} = x]$. Specifically, for every $\langle M, x, 1^t \rangle$ it holds that $\Pr[Y_n = \langle M, x, 1^t \rangle] = 2^{-|M|} \cdot \pi_x / \binom{n}{2}$, where $n \stackrel{\text{def}}{=} |\langle M, x, 1^t \rangle| \stackrel{\text{def}}{=} |M| + |x| + t$. Alternatively, we may set $\Pr[Y_n = \langle M, x, 1^t \rangle] = \pi_x$ if $M = M_S$ and $t = p_S(|x|)$ and $\Pr[Y_n = \langle M, x, 1^t \rangle] = 0$ otherwise, where M_S and P_S are as in the proof of Theorem 2.18.

Exercise 10.21 (monotone markability and monotone reductions) In continuation to Exercise 2.30, we say that a set T is monotonically markable if there exists a polynomial-time (marking) algorithm M such that

1. For every $z, \alpha \in \{0,1\}^*$ it holds that $M(z,\alpha) \in T$ if and only if $z \in T$.
2. For every $|z'| = |z''|$ and $|\alpha'| = |\alpha''|$, it holds that
   (a) If $\alpha' < \alpha''$ then $M(z',\alpha') < M(z'',\alpha'')$.
   (b) $|M(z',\alpha')| = |M(z'',\alpha'')|$.
3. For every ℓ there exists $\ell' \in [\ell, \mathrm{poly}(\ell)]$ such that for every $z \in \cup_{i=1}^{\ell}\{0,1\}^i$ there exists $\ell'' \in [\ell']$ such that $|M(z, 1^{\ell''})| = \ell'$.

Note that Condition 1 is reproduced from Exercise 2.30, whereas Conditions 2 and 3 are new. Prove that if the set S is Karp-reducible to the set T and T is monotonically markable then S is Karp-reducible to T by a reduction that is monotone and length-regular (i.e., the reduction satisfies the conditions of Proposition 10.18).

Guideline: Given a Karp-reduction f from S to T, first obtain a length-regular reduction f' from S to T (by applying the marking algorithm to $f(x)$, while using Conditions 1 and 3). Next, obtain a reduction f'' that is also monotone (e.g., by letting $f''(x) = M(f'(x), x)$, while using Conditions 1 and 2).

Exercise 10.22 (monotone markability and markability) Prove that if a set is monotonically markable (as per Exercise 10.21) then it is markable (as per Exercise 2.30).

Exercise 10.23 (some monotonically markable sets) Referring to Exercise 10.21, verify that each of the twenty-one NP-complete problems treated in in Karp's first paper on NP-completeness [131] is monotonically markable. For starters, consider the sets `SAT`, `Clique`, and `3-Colorability`.

Guideline: For `SAT` consider the following marking algorithm M. This algorithm uses two (fixed) satisfiable formulae of the same length, denoted ψ_0, ψ_1, such that $\psi_0 < \psi_1$. For any formula ϕ and $\sigma_1 \cdots \sigma_m \in \{0,1\}^m$, it holds that $M(\phi, \sigma_1 \cdots \sigma_m) = \psi_{\sigma_1} \wedge \cdots \wedge \psi_{\sigma_m} \wedge \phi$, where ψ_0, ψ_1 use variables that do not appear in ϕ. Note that the multiple occurrences of ψ_σ can be easily avoided (by using "variations" of ψ_σ).

Exercise 10.24 (randomized reductions) Following the outline in §10.2.1.3, provide a definition of randomized reductions among distributional problems.

1. In analogy to Exercise 10.15, prove that randomized reductions preserve feasible solveability (i.e., typical solveability in probabilistic polynomial-time). That is, if the distributional problem (S,X) is randomly reducible to the distributional problem (S',X') and $(S',X') \in \mathrm{tpc}\mathcal{BPP}$, then (S,X) is in $\mathrm{tpc}\mathcal{BPP}$.
2. In analogy to Exercise 10.16, prove that randomized reductions preserve solveability by probabilistic algorithms that err with probability at most 1/3 on each input and typically run in polynomial-time.
3. Prove that randomized reductions are transitive (cf. Exercise 10.17).

4. Show that the error probability of randomized reductions can be reduced (while preserving the domination condition).

Extend the foregoing to reductions that involve distributional *search* problems.

Exercise 10.25 (simple vs sampleable ensembles – Part 1) Prove that any simple probability ensemble is polynomial-time sampleable.

Guideline: See Exercise 10.14.

Exercise 10.26 (simple vs sampleable ensembles – Part 2) Assuming that $\#\mathcal{P}$ contains functions that are not computable in polynomial-time, prove that there exists polynomial-time sampleable ensembles that are not simple.

Guideline: Consider any $R \in \mathcal{PC}$ and suppose that p is a polynomial such that $(x,y) \in R$ implies $|y| = p(|x|)$. Then consider the sampling algorithm A that, on input 1^n, uniformly selects $(x,y) \in \{0,1\}^{n-1} \times \{0,1\}^{p(n-1)}$ and outputs $x1$ if $(x,y) \in R$ and $x0$ otherwise. Note that $\#R(x) = 2^{p(|x|-1)} \cdot \mathsf{Pr}[A(1^{|x|-1})=x1]$.

Exercise 10.27 (distributional versions of NPC problems – Part 1 [28]) Prove that for any NP-complete problem S there exists a polynomial-time sampleable ensemble X such that any problem in dist$\mathcal{NP}$ is reducible to (S,X). We actually assume that the many-to-one reductions establishing the NP-completeness of S do not shrink the length of the input.

Guideline: Prove that the guaranteed reduction of $S_{\mathtt{u}}$ to S also reduces $(S_{\mathtt{u}},U')$ to (S,X), for some sampleable probability ensemble X. Consider first the case that the standard reduction of $S_{\mathtt{u}}$ to S is length preserving, and prove that, when applied to a sampleable probability ensemble, it induces a sampleable distribution on the instances of S. (Note that U' is sampleable (by Exercise 10.25).) Next extend the treatment to the general case, where applying the standard reduction to U'_n induces a distribution on $\cup_{m=n}^{\mathrm{poly}(n)}\{0,1\}^m$ (rather than a distribution on $\{0,1\}^n$).

Exercise 10.28 (distributional versions of NPC problems – Part 2 [28]) Prove Theorem 10.25 (i.e., for any NP-complete problem S there exists a polynomial-time sampleable ensemble X such that any problem in samp$\mathcal{NP}$ is reducible to (S,X)). As in Exercise 10.27, we actually assume that the many-to-one reductions establishing the NP-completeness of S do not shrink the length of the input.

Guideline: We establish the claim for $S_{\mathtt{u}}$, and the general claim follows by using the reduction of $S_{\mathtt{u}}$ to S (as in Exercise 10.27). Thus, we focus on showing that, for some (suitably chosen) sampleable ensemble X, any $(S',X') \in$ samp$\mathcal{NP}$ is reducible to $(S_{\mathtt{u}},X)$. Loosely speaking, X will be an adequate convex combination of all sampleable distributions (and thus X will not equal U' or U). Specifically, $X = \{X_n\}_{n\in\mathbb{N}}$ is defined such that X_n uniformly selects $i \in [n]$, emulates the execution of the i^{th} algorithm (in lexicographic order) on input 1^n for n^3 steps,[28] and outputs whatever the latter has output

[28]Needless to say, the choice to consider n algorithms in the definition of X_n is quite arbitrary. Any other unbounded function of n that is at most a polynomial (and is computable in polynomial-time) will do. (More generally, we may select the i^{th} algorithm with p_i, as long as p_i is a noticeable function of n.) Likewise, the choice to emulate each algorithm for a cubic number of steps (rather some other fixed polynomial number of steps) is quite arbitrary.

(or 0^n in case the said algorithm has not halted within n^3 steps). Prove that, for any $(S'', X'') \in \mathrm{samp}\mathcal{NP}$ such that X'' is sampleable in cubic time, the standard reduction of S'' to $S_{\mathtt{u}}$ reduces (S'', X'') to $(S_{\mathtt{u}}, X)$ (as per Definition 10.15; i.e., in particular, it satisfies the domination condition).[29] Finally, using adequate padding, reduce any $(S', X') \in \mathrm{samp}\mathcal{NP}$ to some $(S'', X'') \in \mathrm{samp}\mathcal{NP}$ such that X'' is sampleable in cubic time.

Exercise 10.29 (search vs decision in the context of sampleable ensembles) Prove that every problem in $\mathrm{samp}\mathcal{NP}$ is reducible to some problem in $\mathrm{samp}\mathcal{PC}$, and every problem in $\mathrm{samp}\mathcal{PC}$ is *randomly* reducible to some problem in $\mathrm{samp}\mathcal{NP}$.

Guideline: See proof of Theorem 10.23.

[29] Note that applying this reduction to X'' yields an ensembles that is also sampleable in cubic time. This claim uses the fact that the standard reduction runs in time that is less than cubic (and in fact almost linear) in its output, and the fact that the output is longer than the input.

Bibliography

[1] S. Aaronson. Complexity Zoo. A continueously updated web-site at `http://qwiki.caltech.edu/wiki/Complexity_Zoo/`.

[2] L.M. Adleman and M. Huang. *Primality Testing and Abelian Varieties Over Finite Fields.* Springer-Verlag Lecture Notes in Computer Science (Vol. 1512), 1992. Preliminary version in *19th STOC*, 1987.

[3] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, Vol. 160 (2), pages 781–793, 2004.

[4] M. Ajtai, J. Komlos, E. Szemerédi. Deterministic Simulation in LogSpace. In *19th ACM Symposium on the Theory of Computing*, pages 132–140, 1987.

[5] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[6] N. Alon, L. Babai and A. Itai. A fast and Simple Randomized Algorithm for the Maximal Independent Set Problem. *J. of Algorithms*, Vol. 7, pages 567–583, 1986.

[7] N. Alon and R. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, Vol. 7 (1), pages 1–22, 1987.

[8] N. Alon, E. Fischer, I. Newman, and A. Shapira. A Combinatorial Characterization of the Testable Graph Properties: It's All About Regularity. In *38th ACM Symposium on the Theory of Computing*, pages 251–260, 2006.

[9] N. Alon, O. Goldreich, J. Håstad, R. Peralta. Simple Constructions of Almost k-wise Independent Random Variables. *Journal of Random Structures and Algorithms*, Vol. 3, No. 3, pages 289–304, 1992. Preliminary version in *31st FOCS*, 1990.

[10] N. Alon and J.H. Spencer. *The Probabilistic Method.* John Wiley & Sons, Inc., 1992.

[11] R. Armoni. On the derandomization of space-bounded computations. In the proceedings of *Random98*, Springer-Verlag, Lecture Notes in Computer Science (Vol. 1518), pages 49–57, 1998.

[12] S. Arora. Approximation schemes for NP-hard geometric optimization problems: A survey. *Math. Programming*, Vol. 97, pages 43–69, July 2003.

[13] S. Arora abd B. Barak. *Complexity Theory: A Modern Approach.* Cambridge University Press, to appear.

[14] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *Journal of the ACM*, Vol. 45, pages 501–555, 1998. Preliminary version in *33rd FOCS*, 1992.

[15] S. Arora and S. Safra. Probabilistic Checkable Proofs: A New Characterization of NP. *Journal of the ACM*, Vol. 45, pages 70–122, 1998. Preliminary version in *33rd FOCS*, 1992.

[16] H. Attiya and J. Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.

[17] L. Babai. Trading Group Theory for Randomness. In *17th ACM Symposium on the Theory of Computing*, pages 421–429, 1985.

[18] L. Babai, L. Fortnow, and C. Lund. Non-Deterministic Exponential Time has Two-Prover Interactive Protocols. *Computational Complexity*, Vol. 1, No. 1, pages 3–40, 1991. Preliminary version in *31st FOCS*, 1990.

[19] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking Computations in Polylogarithmic Time. In *23rd ACM Symposium on the Theory of Computing*, pages 21–31, 1991.

[20] L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

[21] L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System and a Hierarchy of Complexity Classes. *Journal of Computer and System Science*, Vol. 36, pp. 254–276, 1988.

[22] E. Bach and J. Shallit. *Algorithmic Number Theory* (Volume I: Efficient Algorithms). MIT Press, 1996.

[23] B. Barak. Non-Black-Box Techniques in Crypptography. PhD Thesis, Weizmann Institute of Science, 2004.

[24] W. Baur and V. Strassen. The Complexity of Partial Derivatives. *Theor. Comput. Sci.* 22, pages 317–330, 1983.

[25] P. Beame and T. Pitassi. Propositional Proof Complexity: Past, Present, and Future. In *Bulletin of the European Association for Theoretical Computer Science*, Vol. 65, June 1998, pp. 66–89.

[26] M. Bellare, O. Goldreich, and E. Petrank. Uniform Generation of NP-witnesses using an NP-oracle. *Information and Computation*, Vol. 163, pages 510–526, 2000.

[27] M. Bellare, O. Goldreich and M. Sudan. Free Bits, PCPs and Non-Approximability – Towards Tight Results. *SIAM Journal on Computing*, Vol. 27, No. 3, pages 804–915, 1998. Extended abstract in *36th FOCS*, 1995.

[28] S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *Journal of Computer and System Science*, Vol. 44 (2), pages 193–219, 1992.

[29] A. Ben-Dor and S. Halevi. In *2nd Israel Symp. on Theory of Computing and Systems*, IEEE Computer Society Press, pages 108-117, 1993.

[30] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali and P. Rogaway. Everything Provable is Probable in Zero-Knowledge. In *Crypto88*, Springer-Verlag Lecture Notes in Computer Science (Vol. 403), pages 37–56, 1990

[31] M. Ben-Or, S. Goldwasser, J. Kilian and A. Wigderson. Multi-Prover Interactive Proofs: How to Remove Intractability. In *20th ACM Symposium on the Theory of Computing*, pages 113–131, 1988.

[32] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20th ACM Symposium on the Theory of Computing*, pages 1–10, 1988.

[33] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, Shorter PCPs and Applications to Coding. In *36th ACM Symposium on the Theory of Computing*, pages 1–10, 2004. Full version in *ECCC*, TR04-021, 2004.

[34] E. Ben-Sasson and M. Sudan. Simple PCPs with Poly-log Rate and Query Complexity. *ECCC*, TR04-060, 2004.

[35] L. Berman and J. Hartmanis. On isomorphisms and density of NP and other complete sets. *SIAM Journal on Computing*, Vol. 6 (2), 1977, pages 305–322.

[36] M. Blum. A Machine-Independent Theory of the Complexity of Recursive Functions. *Journal of the ACM*, Vol. 14 (2), pages 290–305, 1967.

[37] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

[38] M. Blum, M. Luby and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *Journal of Computer and System Science*, Vol. 47, No. 3, pages 549–595, 1993.

[39] A. Bogdanov, K. Obata, and L. Trevisan. A lower bound for testing 3-colorability in bounded-degree graphs. In *43rd IEEE Symposium on Foundations of Computer Science*, pages 93–102, 2002.

[40] A. Bogdanov and L. Trevisan. On worst-case to average-case reductions for NP problems. In *Proc. 44th IEEE Symposium on Foundations of Computer Science*, pages 308–317, 2003.

[41] A. Bogdanov and L. Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, to appear.

[42] R. Boppana, J. Håstad, and S. Zachos. Does Co-NP Have Short Interactive Proofs? *Information Processing Letters*, 25, May 1987, pages 127-132.

[43] R. Boppana and M. Sipser. The complexity of finite functions. In *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pages 757–804.

[44] A. Borodin. Computational Complexity and the Existence of Complexity Gaps. *Journal of the ACM*, Vol. 19 (1), pages 158–174, 1972.

[45] A. Borodin. On Relating Time and Space to Size and Depth. *SIAM Journal on Computing*, Vol. 6 (4), pages 733–744, 1977.

[46] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *Journal of Computer and System Science*, Vol. 37, No. 2, pages 156–189, 1988. Preliminary version by Brassard and Crépeau in *27th FOCS*, 1986.

[47] L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Science*, Vol. 18, 1979, pages 143–154.

[48] G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *Journal of the ACM*, Vol. 13, pages 547–570, 1966.

[49] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer. Alternation. *Journal of the ACM*, Vol. 28, pages 114–133, 1981.

[50] D. Chaum, C. Crépeau and I. Damgård. Multi-party unconditionally Secure Protocols. In *20th ACM Symposium on the Theory of Computing*, pages 11–19, 1988.

[51] B. Chor and O. Goldreich. On the Power of Two–Point Based Sampling. *Jour. of Complexity*, Vol 5, 1989, pages 96–106. Preliminary version dates 1985.

[52] A. Church. An Unsolvable Problem of Elementary Number Theory. *Amer. J. of Math.*, Vol. 58, pages 345–363, 1936.

[53] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems. SIAM Monographs on Discrete Mathematics and Applications*, 2001.

[54] A. Cobham. The Intristic Computational Difficulty of Functions. In *Proc. 1964 Iternational Congress for Logic Methodology and Philosophy of Science*, pages 24–30, 1964.

[55] S.A. Cook. The Complexity of Theorem Proving Procedures. In *3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[56] S.A. Cook. A overview of Computational Complexity. Turing Award Lecture. *CACM*, Vol. 26 (6), pages 401–408, 1983.

[57] S.A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Information and Control*, Vol. 64, pages 2–22, 1985.

[58] S.A. Cook and R.A. Reckhow. Stephen A. Cook, Robert A. Reckhow: The Relative Efficiency of Propositional Proof Systems. *J. of Symbolic Logic*, Vol. 44 (1), pages 36–50, 1979.

[59] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9, pages 251–280, 1990.

[60] T.M. Cover and G.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New-York, 1991.

[61] P. Crescenzi and V. Kann. A compendium of NP Optimization problems. Available at `http://www.nada.kth.se/~viggo/wwwcompendium/`

[62] W. Diffie, and M.E. Hellman. New Directions in Cryptography. *IEEE Trans. on Info. Theory*, IT-22 (Nov. 1976), pages 644–654.

[63] I. Dinur. The PCP Theorem by Gap Amplification. In *38th ACM Symposium on the Theory of Computing*, pages 241–250, 2006.

[64] I. Dinur and O. Reingold. Assignment-testers: Towards a combinatorial proof of the PCP-Theorem. In *45th IEEE Symposium on Foundations of Computer Science*, pages 155–164, 2004.

[65] I. Dinur and S. Safra. The importance of being biased. In *34th ACM Symposium on the Theory of Computing*, pages 33–42, 2002.

[66] J. Edmonds. Paths, Trees, and Flowers. *Canad. J. Math.*, Vol. 17, pages 449–467, 1965.

[67] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[68] S. Even, A.L. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. *Information and Control*, Vol. 61, pages 159–173, 1984.

[69] U. Feige, S. Goldwasser, L. Lovász and S. Safra. On the Complexity of Approximating the Maximum Size of a Clique. Unpublished manuscript, 1990.

[70] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *Journal of the ACM*, Vol. 43, pages 268–292, 1996. Preliminary version in *32nd FOCS*, 1991.

[71] U. Feige, D. Lapidot, and A. Shamir. Multiple Non-Interactive Zero-Knowledge Proofs Under General Assumptions. *SIAM Journal on Computing*, Vol. 29 (1), pages 1–28, 1999.

[72] U. Feige and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. In *22nd ACM Symposium on the Theory of Computing*, pages 416–426, 1990.

[73] E. Fischer. The art of uninformed decisions: A primer to property testing. *Bulletin of the European Association for Theoretical Computer Science*, Vol. 75, pages 97–126, 2001.

[74] G.D. Forney. *Concatenated Codes*. MIT Press, Cambridge, MA 1966.

[75] L. Fortnow, R. Lipton, D. van Melkebeek, and A. Viglas. Time-space lower bounds for satisfiability. *Journal of the ACM*, Vol. 52 (6), pages 835–865, November 2005.

[76] L. Fortnow, J. Rompel and M. Sipser. On the power of multi-prover interactive protocols. In *3rd IEEE Symp. on Structure in Complexity Theory*, pages 156–161, 1988. See errata in *5th IEEE Symp. on Structure in Complexity Theory*, pages 318–319, 1990.

[77] S. Fortune. A Note on Sparse Complete Sets. *SIAM Journal on Computing*, Vol. 8, pages 431–433, 1979.

[78] M. Fürer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos. On Completeness and Soundness in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 429–442, 1989.

[79] M.L. Furst, J.B. Saxe, and M. Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, Vol. 17 (1), pages 13–27, 1984. Preliminary version in *22nd FOCS*, 1981.

[80] O. Gaber and Z. Galil. Explicit Constructions of Linear Size Superconcentrators. *Journal of Computer and System Science*, Vol. 22, pages 407–420, 1981.

[81] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[82] J. von zur Gathen. Algebraic Complexity Theory. *Ann. Rev. Comput. Sci.*, Vol. 3, pages 317–347, 1988.

[83] O. Goldreich. *Foundation of Cryptography – Class Notes.* Computer Science Dept., Technion, Israel, Spring 1989. Superseded by [87, 88].

[84] O. Goldreich. A Note on Computational Indistinguishability. *Information Processing Letters*, Vol. 34, pages 277–281, May 1990.

[85] O. Goldreich. Notes on Levin's Theory of Average-Case Complexity. *ECCC*, TR97-058, Dec. 1997.

[86] O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), Springer, 1999.

[87] O. Goldreich. *Foundation of Cryptography: Basic Tools.* Cambridge University Press, 2001.

[88] O. Goldreich. *Foundation of Cryptography: Basic Applications.* Cambridge University Press, 2004.

[89] O. Goldreich. Short Locally Testable Codes and Proofs (Survey). *ECCC*, TR05-014, 2005.

[90] O. Goldreich. On Promise Problems (a survey in memory of Shimon Even [1935-2004]). *ECCC*, TR05-018, 2005.

[91] O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

[92] O. Goldreich, S. Goldwasser, and A. Nussboim. On the Implementation of Huge Random Objects. In *44th IEEE Symposium on Foundations of Computer Science*, pages 68–79, 2002.

[93] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, pages 653–750, July 1998.

[94] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, Vol. 25, No. 1, February 1996, pages 169–192. Preliminary version in *17th ICALP*, 1990.

[95] O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st ACM Symposium on the Theory of Computing*, pages 25–32, 1989.

[96] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM*, Vol. 38, No. 3, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

[97] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th ACM Symposium on the Theory of Computing,* pages 218–229, 1987.

[98] O. Goldreich, N. Nisan and A. Wigderson. On Yao's XOR-Lemma. *ECCC*, TR95-050, 1995.

[99] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Algorithmica*, pages 302–343, 2002.

[100] O. Goldreich and D. Ron. A sublinear bipartite tester for bounded degree graphs. *Combinatorica*, Vol. 19 (3), pages 335–373, 1999.

[101] O. Goldreich, R. Rubinfeld and M. Sudan. Learning polynomials with queries: the highly noisy case. *SIAM J. Discrete Math.*, Vol. 13 (4), pages 535–570, 2000.

[102] O. Goldreich, S. Vadhan and A. Wigderson. On interactive proofs with a laconic provers. *Computational Complexity*, Vol. 11, pages 1–53, 2002.

[103] O. Goldreich and A. Wigderson. Computational Complexity. In *The Princeton Companion to Mathematics*, to appear.

[104] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[105] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985. Earlier versions date to 1982.

[106] S. Goldwasser, S. Micali, and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, April 1988, pages 281–308.

[107] S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 73–90, 1989. Extended abstract in *18th STOC*, 1986.

[108] S.W. Golomb. *Shift Register Sequences*. Holden-Day, 1967. (Aegean Park Press, revised edition, 1982.)

[109] V. Guruswami, C. Umans, and S. Vadhan. Extractors and condensers from univariate polynomials. *ECCC*, TR06-134, 2006.

[110] J. Hartmanis and R.E. Stearns. On the Computational Complexity of of Algorithms. *Transactions of the AMS*, Vol. 117, pages 285–306, 1965.

[111] J. Håstad. Almost Optimal Lower Bounds for Small Depth Circuits. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 143–170, 1989. Extended abstract in *18th STOC*, 1986.

[112] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, Vol. 182, pages 105–142, 1999. Preliminary versions in *28th STOC* (1996) and *37th FOCS* (1996).

[113] J. Håstad. Getting optimal in-approximability results. *Journal of the ACM*, Vol. 48, pages 798–859, 2001. Extended abstract in *29th STOC*, 1997.

[114] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SIAM Journal on Computing*, Volume 28, Number 4, pages 1364–1396, 1999. Preliminary versions by Impagliazzo *et. al.* in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

[115] J. Håstad and S. Khot. Query efficient PCPs with pefect completeness. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 610–619, 2001.

[116] A. Healy. Randomness-Efficient Sampling within NC1. *Journal of Computational Complexity*, to appear. Preliminary version in *10th RANDOM*, 2006.

[117] A. Healy, S. Vadhan and E. Viola. Using nondeterminism to amplify hardness. In *36th ACM Symposium on the Theory of Computing*, pages 192–201, 2004.

[118] D. Hochbaum (ed.). *Approximation Algorithms for NP-Hard Problems.* PWS, 1996.

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

[120] S. Hoory, N. Linial, and A. Wigderson. *Expander Graphs and their Applications. Bull. AMS*, Vol. 43 (4), pages 439–561, 2006.

[121] N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Journal on Computing*, Vol. 17, pages 760–778, 1988.

[122] R. Impagliazzo. Hard-core Distributions for Somewhat Hard Problems. In *36th IEEE Symposium on Foundations of Computer Science*, pages 538–545, 1995.

[123] R. Impagliazzo and L.A. Levin. No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random. In *31st IEEE Symposium on Foundations of Computer Science*, pages 812–821, 1990.

[124] R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th ACM Symposium on the Theory of Computing*, pages 220–229, 1997.

[125] R. Impagliazzo and A. Wigderson. Randomness vs Time: Derandomization under a Uniform Assumption. *Journal of Computer and System Science*, Vol. 63 (4), pages 672-688, 2001.

[126] R. Impagliazzo and M. Yung. Direct Zero-Knowledge Computations. In *Crypto87*, Springer-Verlag Lecture Notes in Computer Science (Vol. 293), pages 40–51, 1987.

[127] M. Jerrum, A. Sinclair, and E. Vigoda. A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries. *Journal of the ACM*, Vol. 51 (4), pages 671–697, 2004.

[128] M. Jerrum, L. Valiant, and V.V. Vazirani. Random Generation of Combinatorial Structures from a Uniform Distribution. *Theoretical Computer Science*, Vol. 43, pages 169–188, 1986.

[129] N. Kahale, Eigenvalues and Expansion of Regular Graphs. *Journal of the ACM*, Vol. 42 (5), pages 1091–1106, September 1995.

[130] R. Kannan, H. Venkateswaran, V. Vinay, and A.C. Yao. A Circuit-based Proof of Toda's Theorem. *Information and Computation*, Vol. 104 (2), pages 271–276, 1993.

[131] R.M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pages 85–103, 1972.

[132] R.M. Karp and R.J. Lipton. Some connections between nonuniform and uniform complexity classes. In *12th ACM Symposium on the Theory of Computing*, pages 302-309, 1980.

[133] R.M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *24th IEEE Symposium on Foundations of Computer Science*, pages 56-64, 1983.

[134] R.M. Karp and V. Ramachandran: Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science, Vol A: Algorithms and Complexity*, 1990.

[135] M. Karchmer and A. Wigderson. Monotone Circuits for Connectivity Require Super-logarithmic Depth. *SIAM J. Discrete Math.*, Vol. 3 (2), pages 255–265, 1990. Preliminary version in *20th STOC*, 1988.

[136] M.J. Kearns and U.V. Vazirani. *An introduction to Computational Learning Theory.* MIT Press, 1994.

[137] S. Khot and O. Regev. Vertex Cover Might be Hard to Approximate to within $2-\varepsilon$. In *18th IEEE Conference on Computational Complexity*, pages 379–386, 2003.

[138] V.M. Khrapchenko. A method of determining lower bounds for the complexity of Pi-schemes. In *Matematicheskie Zametki* 10 (1),pages 83–92, 1971 (in Russian). English translation in *Mathematical Notes of the Academy of Sciences of the USSR* 10 (1) 1971, pages 474–479.

[139] J. Kilian. A Note on Efficient Zero-Knowledge Proofs and Arguments. In *24th ACM Symposium on the Theory of Computing*, pages 723–732, 1992.

[140] D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

[141] A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

[142] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1996.

[143] R.E. Ladner. On the Structure of Polynomial Time Reducibility. *Journal of the ACM*, Vol. 22, 1975, pages 155–171.

[144] C. Lautemann. BPP and the Polynomial Hierarchy. *Information Processing Letters*, 17, pages 215–217, 1983.

[145] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[146] L.A. Levin. Universal Search Problems. *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in *problems of Information Transmission 9*, pages 265–266.

[147] L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Information and Control*, Vol. 61, pages 15–37, 1984.

[148] L.A. Levin. Average Case Complete Problems. *SIAM Journal on Computing*, Vol. 15, pages 285–286, 1986.

[149] L.A. Levin. Fundamentals of Computing. *SIGACT News*, Education Forum, special 100-th issue, Vol. 27 (3), pages 89–110, 1996.

[150] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

[151] N. Livne. All Natural NPC Problems Have Average-Case Complete Versions. *ECCC*, TR06-122, 2006.

[152] C.-J. Lu, O. Reingold, S. Vadhan, and A. Wigderson. Extractors: optimal up to constant factors. In *35th ACM Symposium on the Theory of Computing*, pages 602–611, 2003.

[153] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan Graphs. *Combinatorica*, Vol. 8, pages 261–277, 1988.

[154] M. Luby and A. Wigderson. Pairwise Independence and Derandomization. TR-95-035, International Computer Science Institute (ICSI), Berkeley, 1995. ISSN 1075-4946.

[155] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *Journal of the ACM*, Vol. 39, No. 4, pages 859–868, 1992. Preliminary version in *31st FOCS*, 1990.

[156] F. MacWilliams and N. Sloane. *The theory of error-correcting codes.* North-Holland, 1981.

[157] G.A. Margulis. Explicit Construction of Concentrators. (In Russian.) *Prob. Per. Infor.*, Vol. 9 (4), pages 71–80, 1973. English translation in *Problems of Infor. Trans.*, pages 325–332, 1975.

[158] S. Micali. Computationally Sound Proofs. *SIAM Journal on Computing*, Vol. 30 (4), pages 1253–1298, 2000. Preliminary version in *35th FOCS*, 1994.

[159] G.L. Miller. Riemann's Hypothesis and Tests for Primality. *Journal of Computer and System Science*, Vol. 13, pages 300–317, 1976.

[160] P.B. Miltersen and N.V. Vinodchandran. Derandomizing Arthur-Merlin Games using Hitting Sets. *Journal of Computational Complexity*, Vol. 14 (3), pages 256–279, 2005. Preliminary version in *40th FOCS*, 1999.

[161] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[162] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, Vol. 4, pages 151–158, 1991.

[163] J. Naor and M. Naor. Small-bias Probability Spaces: Efficient Constructions and Applications. *SIAM Journal on Computing*, Vol 22, 1993, pages 838–856. Preliminary version in *22nd STOC*, 1990.

[164] M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Application. In *21st ACM Symposium on the Theory of Computing*, 1989, pages 33–43.

[165] M. Nguyen, S.J. Ong, S. Vadhan. Statistical Zero-Knowledge Arguments for NP from Any One-Way Function. In *47th IEEE Symposium on Foundations of Computer Science*, pages 3-14, 2006.

[166] N. Nisan. Pseudorandom bits for constant depth circuits. *Combinatorica*, Vol. 11 (1), pages 63–70, 1991.

[167] N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992.

[168] N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Journal of Computational Complexity*, Vol. 4, pages 1-11, 1994.

[169] N. Nisan and A. Wigderson. Hardness vs Randomness. *Journal of Computer and System Science*, Vol. 49, No. 2, pages 149–167, 1994.

[170] N. Nisan and D. Zuckerman. Randomness is Linear in Space. *Journal of Computer and System Science*, Vol. 52 (1), pages 43–52, 1996.

[171] C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[172] C.H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. In *20th ACM Symposium on the Theory of Computing*, pages 229–234, 1988.

[173] N. Pippenger and M.J. Fischer. Relations among complexity measures. *Journal of the ACM*, Vol. 26 (2), pages 361–381, 1979.

[174] E. Post. A Variant of a Recursively Unsolvable Problem. *Bull. AMS*, Vol. 52, pages 264–268, 1946.

[175] M.O. Rabin. Digitalized Signatures. In *Foundations of Secure Computation* (R.A. DeMillo et. al. eds.), Academic Press, 1977.

[176] M.O. Rabin. Digitalized Signatures and Public Key Functions as Intractable as Factoring. MIT/LCS/TR-212, 1979.

[177] M.O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, Vol. 12, pages 128–138, 1980.

[178] R. Raz. A Parallel Repetition Theorem. *SIAM Journal on Computing*, Vol. 27 (3), pages 763–803, 1998. Extended abstract in *27th STOC*, 1995.

[179] R. Raz and A. Wigderson. Monotone Circuits for Matching Require Linear Depth. *Journal of the ACM*, Vol. 39 (3), pages 736–744, 1992. Preliminary version in *22nd STOC*, 1990.

[180] A. Razborov. Lower bounds for the monotone complexity of some Boolean functions. In *Doklady Akademii Nauk SSSR*, Vol. 281, No. 4, 1985, pages 798–801. English translation in *Soviet Math. Doklady*, 31, pages 354–357, 1985.

[181] A. Razborov. Lower bounds on the size of bounded-depth networks over a complete basis with logical addition. In *Matematicheskie Zametki*, Vol. 41, No. 4, pages 598–607, 1987. English translation in *Mathematical Notes of the Academy of Sci. of the USSR*, Vol. 41 (4), pages 333–338, 1987.

[182] A.R. Razborov and S. Rudich. Natural Proofs. *Journal of Computer and System Science*, Vol. 55 (1), pages 24–35, 1997.

[183] O. Reingold. Undirected ST-Connectivity in Log-Space. In *37th ACM Symposium on the Theory of Computing*, pages 376–385, 2005.

[184] O. Reingold, S. Vadhan, and A. Wigderson. Entropy Waves, the Zig-Zag Graph Product, and New Constant-Degree Expanders and Extractors. *Annals of Mathematics*, Vol. 155 (1), pages 157–187, 2001. Preliminary version in *41st FOCS*, pages 3–13, 2000.

[185] H.G. Rice. Classes of Recursively Enumerable Sets and their Decision Problems. *Trans. AMS*, Vol. 89, pages 25–59, 1953.

[186] R.L. Rivest, A. Shamir and L.M. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *CACM*, Vol. 21, Feb. 1978, pages 120–126.

[187] D. Ron. Property testing. In *Handbook on Randomization, Volume II*, pages 597–649, 2001. (Editors: S. Rajasekaran, P.M. Pardalos, J.H. Reif and J.D.P. Rolim.)

[188] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM Journal on Computing*, Vol. 25 (2), pages 252–271, 1996.

[189] M. Saks and S. Zhou. $\mathrm{RSPACE}(S) \subseteq \mathrm{DSPACE}(S^{3/2})$. In 36th *IEEE Symposium on Foundations of Computer Science*, pages 344–353, 1995.

[190] W.J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *JCSS*, Vol. 4 (2), pages 177-192, 1970.

[191] A. Selman. On the structure of NP. *Notices Amer. Math. Soc.*, Vol. 21 (6), page 310, 1974.

[192] R. Shaltiel. Recent Developments in Explicit Constructions of Extractors. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century, Vol 1: Algorithms and Complexity*, World scietific, 2004. (Editors: G. Paun, G. Rozenberg and A. Salomaa.) Preliminary version in *Bulletin of the EATCS 77*, pages 67–95, 2002.

[193] R. Shaltiel and C. Umans. Simple Extractors for All Min-Entropies and a New Pseudo-Random Generator. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 648–657, 2001.

[194] C.E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. American Institute of Electrical Engineers*, Vol. 57, pages 713–723, 1938.

[195] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

[196] C.E. Shannon. Communication Theory of Secrecy Systems. *Bell Sys. Tech. Jour.*, Vol. 28, pages 656–715, 1949.

[197] A. Shamir. IP = PSPACE. *Journal of the ACM*, Vol. 39, No. 4, pages 869–877, 1992. Preliminary version in *31st FOCS*, 1990.

[198] A. Shpilka. Lower Bounds for Matrix Product. *SIAM Journal on Computing*, pages 1185-1200, 2003.

[199] M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.

[200] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[201] R. Smolensky. Algebraic Methods in the Theory of Lower Bounds for Boolean Circuit Complexity. In *19th ACM Symposium on the Theory of Computing* pages 77–82, 1987.

[202] R.J. Solomonoff. A Formal Theory of Inductive Inference. *Information and Control*, Vol. 7/1, pages 1–22, 1964.

[203] R. Solovay and V. Strassen. A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing*, Vol. 6, pages 84–85, 1977. Addendum in *SIAM Journal on Computing*, Vol. 7, page 118, 1978.

[204] D.A. Spielman. *Advanced Complexity Theory*, Lectures 10 and 11. Notes (by D. Lewin and S. Vadhan), March 1997. Available from `http://www.cs.yale.edu/homes/spielman/AdvComplexity/1998/` as `lect10.ps` and `lect11.ps`.

[205] L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, Vol. 3, pages 1–22, 1977.

[206] L. Stockmeyer. The Complexity of Approximate Counting. In *15th ACM Symposium on the Theory of Computing*, pages 118–126, 1983.

[207] V. Strassen. Algebraic Complexity Theory. In *Handbook of Theoretical Computer Science: Volume A – Algorithms and Complexity*, J. van Leeuwen editor, MIT Press/Elsevier, 1990, pages 633–672.

[208] M. Sudan. Decoding of Reed Solomon codes beyond the error-correction bound. *Journal of Complexity*, Vol. 13 (1), pages 180–193, 1997.

[209] M. Sudan. Algorithmic introduction to coding theory. Lecture notes, Available from `http://theory.csail.mit.edu/~madhu/FT01/`, 2001.

[210] M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR Lemma. *Journal of Computer and System Science*, Vol. 62, No. 2, pages 236–266, 2001.

[211] R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.

[212] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, Vol. 20 (5), pages 865–877, 1991.

[213] B.A. Trakhtenbrot. A Survey of Russian Approaches to *Perebor* (Brute Force Search) Algorithms. *Annals of the History of Computing*, Vol. 6 (4), pages 384–398, 1984.

[214] L. Trevisan. Constructions of Near-Optimal Extractors Using Pseudo-Random Generators. In *31st ACM Symposium on the Theory of Computing*, pages 141–148, 1998.

[215] V. Trifonov. An $O(\log n \log\log n)$ Space Algorithm for Undirected st-Connectivity. In *37th ACM Symposium on the Theory of Computing*, pages 623–633, 2005.

[216] C.E. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. Londom Mathematical Soceity*, Ser. 2, Vol. 42, pages 230–265, 1936. A Correction, *ibid.*, Vol. 43, pages 544–546.

[217] C. Umans. Pseudo-random generators for all hardness. *Journal of Computer and System Science*, Vol. 67 (2), pages 419–440, 2003.

[218] S. Vadhan. A Study of Statistical Zero-Knowledge Proofs. PhD Thesis, Department of Mathematics, MIT, 1999. Available from `http://www.eecs.harvard.edu/~salil/papers/phdthesis-abs.html`.

[219] S. Vadhan. An Unconditional Study of Computational Zero Knowledge. In *45th IEEE Symposium on Foundations of Computer Science*, pages 176–185, 2004.

[220] L.G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, Vol. 8, pages 189–201, 1979.

[221] L.G. Valiant. A theory of the learnable. *CACM*, Vol. 27/11, pages 1134–1142, 1984.

[222] L.G. Valiant and V.V. Vazirani. NP Is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47 (1), pages 85–93, 1986.

[223] J. von Neumann, First Draft of a Report on the EDVAC, 1945. Contract No. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia. Reprinted (in part) in *Origins of Digital Computers: Selected Papers*, Springer-Verlag, Berlin Heidelberg, pages 383–392, 1982.

[224] J. von Neumann, Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100, pages 295–320, 1928.

[225] I. Wegener. *The Complexity of Boolean Functions.* Wiley-Teubner, 1987.

[226] I. Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[227] A. Wigderson. The amazing power of pairwise independence. In *26th ACM Symposium on the Theory of Computing*, pages 645–647, 1994.

[228] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[229] A.C. Yao. Separating the Polynomial-Time Hierarchy by Oracles. In *26th IEEE Symposium on Foundations of Computer Science*, pages 1-10, 1985.

[230] A.C. Yao. How to Generate and Exchange Secrets. In *27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[231] S. Yekhanin. New Locally Decodable Codes and Private Information Retrieval Schemes. *ECCC*, TR06-127, 2006.