Computational Complexity: A Modern Approach

Sanjeev Arora and Boaz Barak
Princeton University

http://www.cs.princeton.edu/theory/complexity/

complexitybook@gmail.com

To our wives— Silvia and Ravit

About this book

Computational complexity theory has developed rapidly in the past three decades. The list of surprising and fundamental results proved since 1990 alone could fill a book: these include new probabilistic definitions of classical complexity classes (**IP** = **PSPACE** and the **PCP** Theorems) and their implications for the field of approximation algorithms; Shor's algorithm to factor integers using a quantum computer; an understanding of why current approaches to the famous **P** versus **NP** will not be successful; a theory of derandomization and pseudorandomness based upon computational hardness; and beautiful constructions of pseudorandom objects such as extractors and expanders.

This book aims to describe such recent achievements of complexity theory in the context of more classical results. It is intended to both serve as a textbook and as a reference for self-study. This means it must simultaneously cater to many audiences, and it is carefully designed with that goal. We assume essentially no computational background and very minimal mathematical background, which we review in Appendix A. We have also provided a *web site* for this book at `http://www.cs.princeton.edu/theory/complexity/` with related auxiliary material including detailed teaching plans for courses based on this book, a draft of all the book's chapters, and links to other online resources covering related topics. Throughout the book we explain the context in which a certain notion is useful, and *why* things are defined in a certain way. We also illustrate key definitions with examples. To keep the text flowing, we have tried to minimize bibliographic references, except when results have acquired standard names in the literature, or when we felt that providing some history on a particular result serves to illustrate its motivation or context. (Every chapter has a notes section that contains a fuller, though still brief, treatment of the relevant works.) When faced with a choice, we preferred to use simpler definitions and proofs over showing the most general or most optimized result.

The book is divided into three parts:

Part I: Basic complexity classes. This part provides a broad introduction to the field. Starting from the definition of Turing machines and the basic notions of computability theory, it covers the basic time and space complexity classes, and also includes a few more modern topics such as probabilistic algorithms, interactive proofs, cryptography, quantum computers, and the **PCP** Theorem and its applications.

Part II: Lower bounds on concrete computational models. This part describes lower bounds on resources required to solve algorithmic tasks on concrete models such as circuits, decision trees, etc. Such models may seem at first sight very different from Turing machines, but looking deeper one finds interesting interconnections.

Part III: Advanced topics. This part is largely devoted to developments since the late 1980s. It includes counting complexity, average case complexity, hardness amplification, derandomization and pseudorandomness, the proof of the **PCP** theorem, and natural proofs.

Almost every chapter in the book can be read in isolation (though Chapters 1, 2 and 7 must not be skipped). This is by design, because the book is aimed at many classes of readers:

- *Physicists, mathematicians, and other scientists.* This group has become increasingly interested in computational complexity theory, especially because of high-profile results such as Shor's algorithm and the recent deterministic test for primality. This

intellectually sophisticated group will be able to quickly read through Part I. Progressing on to Parts II and III they can read individual chapters and find almost everything they need to understand current research.

- *Computer scientists who do not work in complexity theory per se.* They may use the book for self-study, reference, or to teach an undergraduate or graduate course in theory of computation or complexity theory.
- *All those —professors or students— who do research in complexity theory or plan to do so.* The coverage of recent results and advanced topics is detailed enough to prepare readers for research in complexity and related areas.

This book can be used as a textbook for several types of courses:

- *Undergraduate Theory of Computation.* Many Computer Science departments offer an undergraduate *Theory of Computation* course, using say Sipser's book [Sip96]. Our text could be used to supplement Sipser's book with coverage of some more modern topics such as probabilistic algorithms, cryptography and quantum computing. Undergraduate students may find these more exciting than traditional topics such as automata theory and the finer distinctions of computability theory. The prerequisite mathematical background would be some comfort with mathematical proofs and discrete mathematics, as covered in the typical "discrete math"/"math for CS" courses currently offered in many CS departments.
- *Introduction to computational complexity for advanced undergrads/beginning grads.* The book can be used as a text for an introductory complexity course aimed at advanced undergraduate or graduate students in computer science (replacing books such as Papadimitriou's 1994 text [Pap94], that do not contain many recent results). Such a course would probably include many topics from Part I and then a sprinkling from Parts II and III, and assume some background in algorithms and/or the theory of computation.
- *Graduate Complexity course.* The book can serve as a text for a graduate complexity course that prepares graduate students for research in complexity theory or related areas like algorithms, machine learning, etc. Such a course can use Part I to review basic material, and then move on to the advanced topics of Parts II and III. The book contains far more material than can be taught in one term, and we provide on our website several alternative outlines for such a course.
- *Graduate seminars or advanced courses.* Individual chapters from Parts II and III can be used in seminars or advanced courses on various topics in complexity theory (e.g., derandomization, the **PCP** Theorem, lower bounds).

We provide several teaching plans and material for such courses on the book's web site. If you use the book in your course, we'd love to hear about it and get your feedback. We ask that you do not publish solutions for the book's exercises on the web though, so other people can use them as homework and exam questions as well.

As we finish this book, we are sorely aware of many more exciting results that we had to leave out. We hope the copious references to other texts will give the reader plenty of starting points for further explorations. We also plan to periodically update the book's website with pointers to newer results or expositions that may be of interest to you.

Above all, we hope that this book conveys our excitement about computational complexity and the insights it provides in a host of other disciplines.

Onward to **P** versus **NP**!

Acknowledgements

Our understanding of complexity theory was shaped through interactions with our colleagues, and we have learned a lot from far too many people to mention here. Boaz would like to especially thank two mentors— Oded Goldreich and Avi Wigderson— who introduced to him the world of theoretical computer science and still influence much of his thinking on this area.

We thank Luca Trevisan for coconceiving the book (7 years ago!) and helping write the first drafts of a couple of chapters. Several colleagues have graciously agreed to review for us early drafts of parts of this book. These include Scott Aaronson, Noga Alon, Paul Beame, Irit Dinur, Venkatesan Guruswami, Valentine Kavanets, Jonathan Katz, Subhash Khot, Jiří Matoušek , Klaus Meer, Or Meir, Moni Naor, Alexandre Pinto, Alexander Razborov, Oded Regev, Omer Reingold, Ronen Shaltiel, Madhu Sudan, Amnon Ta-Shma, Iannis Tourlakis, Chris Umans, Salil Vadhan, Dieter van Melkebeek, Umesh Vazirani, and Joachim von zur Gathen. Special thanks to Jiří, Or, Alexandre, Dieter and Iannis for giving us very detailed and useful comments on many chapters of this book.

We also thank many others who have sent notes on typos or bugs, comments that helped improve the presentations, or answered our questions on a particular proof or reference. These include Emre Akbas, Djangir Babayev, Miroslav Balaz, Arnold Beckmann, Ido Ben-Eliezer, Goutam Biswas, Shreeshankar Bodas, Josh Bronson, Arkadev Chattopadhyay, Bernard Chazelle, Maurice Cochand, Nathan Collins, Tim Crabtree, Morten Dahl, Ronald de Wolf, Scott Diehl, Dave Doty, Michael Fairbank, Joan Feigenbaum, Lance Fortnow, Ali Ghodsi, Parikshit Gopalan, Vipul Goyal, Venkat Guruswami, Stephen Harris, Johan Håstad, Andre Hernich, Yaron Hirsch, Thomas Holenstein, Moukarram Kabbash, Bart Kastermans, Joe Kilian, Tomer Kotek, Michal Koucy, Sebastian Kuhnert, Katrina LaCurts, Chang-Wook Lee, James Lee, John Lenz, Meena Mahajan, Mohammad Mahmoody-Ghidary, Shohei Matsuura, Mauro Mazzieri, John McCullough, Eric Miles, Shira Mitchell, Mohsen Momeni, Kamesh Munagala, Rolf Neidermeier, Robert Nowotniak, Taktin Oey, Emily Pitler, Toni Pitassi, Manoj Prabhakaran, Anup Rao, Saikiran Rapaka, Nicola Rebagliati, Johan Richter, Ron Rivest, Mohammad Sadeq Dousti, Rahul Santhanam, Cem Say, Robert Schweiker, Thomas Schwentick, Joel Seiferas, Jonah Sherman, Amir Shpilka, Yael Snir, Nikhil Srivastava, Thomas Starbird, Jukka Suomela, Elad Tsur, Leslie Valiant, Vijay Vazirani, Suresh Venkatasubramanisn, Justin Vincent-Foglesong, Jirka Vomlel, Daniel Wichs, Avi Wigderson, Maier Willard, Roger Wolff, Jureg Wullschleger, Rui Xue, Jon Yard, Henry Yuen, Wu Zhanbin, and Yi Zhang. Thank you!

Doubtless this is list is still missing some of the people that helped us with this project over the years— if you are one of them we are both grateful and sorry.

This book was typeset using LaTeX, for which we're grateful to Donald Knuth and Leslie Lamport. Stephen Boyd and Lieven Vandenberghe kindly shared with us the LaTeX macros of their book *Convex Optimization*.

Most of all, we'd like to thank our families— Silvia, Nia and Rohan Arora, and Ravit and Alma Barak.

Contents at a glance

Contents

Introduction

> *"As long as a branch of science offers an abundance of problems, so long it is alive; a lack of problems foreshadows extinction or the cessation of independent development."*
> David Hilbert, 1900

> *"The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?...I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block."*
> Alan Cobham, 1964

The notion of *computation* has existed in some form for thousands of years, in contexts as varied as routine account-keeping and astronomy. Here are three examples of tasks that we may wish to solve using computation:

- Given two integer numbers, compute their product.
- Given a set of n linear equations over n variables, find a solution if it exists.
- Given a list of acquaintances and a list of all pairs among them who do not get along, find the largest set of acquaintances you can invite to a dinner party such that every two invitees get along with one another.

Throughout history people had a notion of a process of producing an output from a set of inputs in a finite number of steps, and thought of "computation" as "a person writing numbers on a scratch pad following certain rules."

One of the important scientific advances in the first half of the 20th century was that the notion of "computation" received a much more precise definition. From this definition it quickly became clear that computation can happen in diverse physical and mathematical systems —Turing machines, lambda calculus, cellular automata, pointer machines, bouncing billiards balls, Conway's *Game of life*, etc.. Surprisingly, all these forms of computation are equivalent —in the sense that each model is capable of implementing all computations that we can conceive of on any other model (see Chapter 1). This realization quickly led to the invention of the standard *universal electronic computer*, a piece of hardware that is capable of executing all possible programs. The computer's rapid adoption in society in the subsequent decades brought computation into every aspect of modern life, and made computational issues important in design, planning, engineering, scientific discovery, and many other human endeavors. Computer *algorithms*, which are methods of solving computational problems, became ubiquitous.

But computation is not "'merely" a practical tool. It is also a major scientific concept. Generalizing from physical models such as cellular automata, scientists now view many natural phenomena as akin to computational processes. The understanding of reproduction in living things was triggered by the discovery of self-reproduction in computational machines. (In fact, a book by the physicist Schrödinger [Sch44] predicted the existence of a DNA-like substance in cells before Watson and Crick discovered it, and was credited by Crick as an inspiration for that research.) Today, computational models underlie many research areas in biology and neuroscience. Several physics theories such as QED give a description of

nature that is very reminiscent of computation, motivating some scientists to even suggest that the entire universe may be viewed as a giant computer (see Lloyd [Llo06]). In an interesting twist, such physical theories have been used in the past decade to design a model for *quantum computation*; see Chapter 10.

Computability versus complexity. After their success in defining computation, researchers focused on understanding what problems are *computable*. They showed that several interesting tasks are *inherently uncomputable*: no computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. Though a beautiful topic, computability will not be our focus in this book. We discuss it briefly in Chapter 1 and refer the reader to standard texts [Sip96, HMU01, Koz97, Rog87] for more details. Instead, we focus on *computational complexity theory*, which focuses on issues of *computational efficiency* —quantifying the amount of computational resources required to solve a given task. Below, we describe at an informal level how one can quantify "efficiency," and after that discuss some of the issues that arise in connection with its study.

Quantifying computational efficiency

To explain what we mean by *computational efficiency*, we use the three examples of computational tasks we mentioned above. We start with the task of multiplying two integers. Consider two different methods (or *algorithms*) to perform this task. The first is *repeated addition*: to compute $a \cdot b$, just add a to itself $b-1$ times. The other is the *grade-school algorithm* illustrated in Figure 1. Though the repeated addition algorithm is perhaps simpler than the grade-school algorithm, we somehow feel that the latter is *better*. Indeed, it is much more efficient. For example, multiplying 577 by 423 using repeated addition requires 422 additions, whereas doing it with the grade-school algorithm 3 multiplications of a number by a single digit and 3 additions.

$$\begin{array}{cccccc} & & & 5 & 7 & 7 \\ & & & 4 & 2 & 3 \\ \hline & & 1 & 7 & 3 & 1 \\ & 1 & 1 & 5 & 4 & \\ 2 & 3 & 0 & 8 & & \\ \hline 2 & 4 & 4 & 0 & 7 & 1 \end{array}$$

Figure 1 Grade-school algorithm for multiplication. Illustrated for computing $577 \cdot 423$.

We will quantify the efficiency of an algorithm by studying how its number of *basic operations* scales as we increase the *size* of the input. For this discussion, let the *basic operations* be addition and multiplication of single digits. (In other settings, we may wish to throw in division as a basic operation.) The *size* of the input is the number of digits in the numbers. The number of basic operations used to multiply two n-digit numbers (i.e., numbers between 10^{n-1} and 10^n) is at most $2n^2$ for the grade-school algorithm and at least $n10^{n-1}$ for repeated addition. Phrased this way, the huge difference between the two algorithms is apparent: even for 11-digit numbers, a pocket calculator running the grade-school algorithm would beat the best current supercomputer running the repeated addition algorithm. For slightly larger numbers even a fifth grader with pen and paper would outperform a supercomputer. We see that *the efficiency of an algorithm is to a considerable extent much more important than the technology used to execute it.*

Surprisingly enough, there is an even faster algorithm for multiplication that uses the *Fast Fourier Transform*. It was only discovered some 40 years ago and multiplies two n-digit numbers using $cn \log n \log\log n$ operations where c is some absolute constant independent of n; see Chapter 16. We call such an algorithm an $O(n \log n \log\log n)$-step algorithm: see

our notational conventions below. As n grows, this number of operations is significantly smaller than n^2.

For the task of solving linear equations, the classic *Gaussian elimination* algorithm (named after Gauss but already known in some form to Chinese mathematicians of the first century) uses $O(n^3)$ basic arithmetic operations to solve n equations over n variables. In the late 1960's, Strassen found a more efficient algorithm that uses roughly $O(n^{2.81})$ operations, and the best current algorithm takes $O(n^{2.376})$ operations (see Chapter 16).

The *dinner party* task also has an interesting story. As in the case of multiplication, there is an obvious and simple inefficient algorithm: try all possible subsets of the n people from the largest to the smallest, and stop when you find a subset that does not include any pair of guests who don't get along. This algorithm can take as much time as the number of subsets of a group of n people, which is 2^n. This is highly unpractical —an organizer of, say, a 70-person party, would need to plan it at least a thousand years in advance, even if she has a supercomputer at her disposal. Surprisingly, we still do not know of a significantly better algorithm for this task. In fact, as we will see in Chapter 2, we have reasons to suspect that no efficient algorithm *exists*, since this task turns out to be equivalent to the *independent set* computational problem, which, together with thousands of other important problems, is **NP**-complete. The famous "**P** versus **NP**" question (Chapter 2) asks whether or not any of these problems has an efficient algorithm.

Proving nonexistence of efficient algorithms

We have seen that sometimes computational tasks turn out to have nonintuitive algorithms that are more efficient than algorithms used for thousands of years. It would therefore be really interesting to prove for some computational tasks that the current algorithm is the *best* —in other words, no better algorithms exist. For instance, we could try to prove that the $O(n \log n \log \log n)$-step algorithm for multiplication cannot be improved upon (thus implying that multiplication is inherently more difficult than addition, which does have an $O(n)$-step algorithm). Or, we could try to prove that there is no algorithm for the dinner party task that takes fewer than $2^{n/10}$ steps. Trying to prove such results is a central goal of complexity theory.

How can we ever prove such a nonexistence result? There are infinitely many possible algorithms! So we have to *mathematically prove* that each one of them is less efficient that the known algorithm. This may be possible to do, because computation is a mathematically precise notion. In fact, this kind of result (if proved) would fit into a long tradition of *impossibility results* in mathematics, such as the independence of Euclid's parallel postulate from the other basic axioms of geometry, or the impossibility of trisecting an arbitrary angle using a compass and straightedge. Such results count among the most interesting, fruitful, and surprising results in mathematics.

In complexity theory, we are still only rarely able to prove such nonexistence of algorithms. We do have important nonexistence results in some concrete computational models that are not as powerful as general computers, which are described in Part II of the book. Since we are still missing good results for general computers, one important source of progress in complexity theory is our stunning success in *interrelating* different complexity questions, and the rest of the book is filled with examples of these.

Some interesting questions about computational efficiency

Now we give an overview of some important issues regarding computational complexity, all of which will be treated in greater detail in later chapters. An overview of mathematical background is given in Appendix A.

1. Computational tasks in a variety of disciplines such as the life sciences, social sciences and operations research involve searching for a solution across a vast space of possibilities (for example, the aforementioned tasks of solving linear equations and finding a maximal set of invitees to a dinner party). This is sometimes called *exhaustive search*, since the search *exhausts* all possibilities. Can this exhaustive search be replaced by a more *efficient* search algorithm?

   As we will see in Chapter 2, this is essentially the famous "**P** vs. **NP**" question, considered *the* central problem of complexity theory. Many interesting search problems are **NP**-complete, which means that if the famous conjecture $\mathbf{P} \neq \mathbf{NP}$ is true, then these problems do not have efficient algorithms; they are *inherently intractable*.

2. Can algorithms use randomness (i.e., coin tossing) to speed up computation?

   Chapter 7 introduces randomized computation and describes efficient *probabilistic algorithms* for certain tasks. But Chapters 19 and 20 show a surprising recent result giving strong evidence that randomness does *not* help speed up computation too much, in the sense that any probabilistic algorithm can be replaced with a *deterministic* algorithm (tossing no coins) that is almost as efficient.

3. Can hard problems become easier to solve if we allow the algorithms to err on a small number of inputs, or to only compute an *approximate* solution?

   Average-case complexity and *approximation algorithms* are studied in Chapters 11, 18, 19, and 22. These chapters also show fascinating connections between these questions, the power of randomness, different notions of mathematical proofs, and the theory of error correcting codes.

4. Can we derive any practical benefit from computationally hard problems? For example, can we use them to construct cryptographic protocols that are *unbreakable* (at least by any plausible adversary)?

   As described in Chapter 9, the security of digital cryptography is intimately related to the **P** vs. **NP** question (see Chapter 2) and average-case complexity (see Chapters 18).

5. Can we use the counterintuitive quantum mechanical properties of matter to build faster computers?

   Chapter 10 describes the fascinating notion of *quantum computers* that use quantum mechanics to speed up certain computations. Peter Shor has shown that, if ever built, quantum computers will be able to factor integers efficiently (thus breaking many current cryptosystems). However, currently there are many daunting obstacles to actually building such computers,

6. Do we need people to prove mathematical theorems, or can we generate mathematical proofs automatically? Can we check a mathematical proof without reading it completely? Do interactive proofs, involving a dialog between prover and verifier, have more power than standard "static" mathematical proofs?

   The notion of *proof*, central to mathematics, turns out to be central to computational complexity as well, and complexity has shed new light on the meaning of mathematical proofs. Whether mathematical proofs can be generated automatically turns out to depend on the **P** vs. **NP** question (see Chapter 2). Chapter 11 describes *probabilistically checkable proofs*. These are surprisingly robust mathematical proofs that can checked by only reading them in very few probabilistically chosen locations, in contrast to the traditional proofs that require line-by-line verification. Along similar lines we introduce the notion of *interactive proofs* in Chapter 8 and use them to derive some surprising results. Finally, *proof complexity*, a subfield of complexity studying the minimal proof length of various statements, is studied in Chapter 15.

At roughly 40 years of age, Complexity theory is still an infant science and many important results are less than 20 years old. We have few complete answers for any of these questions. In a surprising twist, computational complexity has also been used to prove some

metatmathematical theorems: they provide evidence of the difficulty of resolving some of the questions of ... computational complexity; see Chapter 23.

We conclude with another quote from Hilbert's 1900 lecture:

> *Proofs of impossibility were effected by the ancients ... [and] in later mathematics, the question as to the impossibility of certain solutions plays a preminent part. ...*
>
> *In other sciences also one meets old problems which have been settled in a manner most satisfactory and most useful to science by the proof of their impossibility. ... After seeking in vain for the construction of a perpetual motion machine, the relations were investigated which must subsist between the forces of nature if such a machine is to be impossible; and this inverted question led to the discovery of the law of the conservation of energy. ...*
>
> *It is probably this important fact along with other philosophical reasons that gives rise to conviction ... that every definite mathematical problem must necessarily be susceptible to an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts. ... This conviction... is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorance.*

Main Theorems and Definitions