Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to
complexitybook@gmail.com — **Thank You!!**

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 its everyday meaning, this term refers to the process of producing an output from a set of inputs in a finite number of steps. Here are three examples for computational tasks:

- 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 containing all pairs of individuals who are not on speaking terms with each other, find the largest set of acquaintances you can invite to a dinner party such that you do not invite any two who are not on speaking terms.

In the first half of the 20th century, the notion of "computation" was made much more precise than the hitherto informal notion of "a person writing numbers on a note pad following certain rules." Many different models of computation were discovered —Turing machines, lambda calculus, cellular automata, pointer machines, bouncing billiards balls, Conway's *Game of life*, etc.— and found to be equivalent. More importantly, they are all *universal*, which means that each is capable of implementing all computations that we can conceive of on any other model (see Chapter 1). The notion of universality motivated the invention of the standard *electronic computer*, which is capable of executing all possible programs. The computer's rapid adoption in society in the subsequent half decade brought computation into every aspect of modern life, and made computational issues important in design, planning, engineering, scientific discovery, and many other human endeavors.

However, computation is not just a practical tool, but also a major scientific concept. Generalizing from models such as cellular automata, scientists have come to 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 famous article by Pauli predicted the existence of a DNA-like substance in cells almost a decade before Watson and Crick discovered it.) 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 [**?**]). In an interesting twist, such physical theories have been used in the past decade to design a model for *quantum computation*; see Chapter 20.

From 1930s to the 1950s, researchers focused on the theory of *computability* and showed that several interesting computational tasks are *inherently uncomputable*: no computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. Though a beautiful theory, it will not be our focus here. (But, see the texts [**?**, **?**, **?**, **?**].) Instead, we focus on issues of *computational efficiency*. *Computational complexity theory* is concerned with how much computational resources are required to solve a given task. The questions it studies include the following:

1. Many computational tasks 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). Is there an *efficient* search algorithm for all such tasks, or do some tasks inherently require an exhaustive search?

   As we will see in Chapter 2, this is the famous "**P** vs. **NP**" question that is considered the central problem of complexity theory. Computational search tasks of the form above arise in a host of disciplines including the life sciences, social sciences and operations research, and computational complexity has provided strong evidence that many of these tasks are *inherently intractable*.

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

   Chapter 7 presents *probabilistic algorithms* and shows several algorithms and techniques that use probability to solve tasks more efficiently. But Chapters 16 and 17 show a surprising recent result giving strong evidence that randomness does *not* help speed up computation, 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 be solved quicker 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 15, 17, 18 and 19. 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. Is there any use for computationally hard problems? For example, can we use them to construct secret codes that are *unbreakable*? (at least in the universe's lifetime).

   Our society increasingly relies on digital cryptography for commerce and security. As described in Chapter 10, these secret codes are built using certain hard computational tasks

such as factoring integers. The security of digital cryptography is intimately related to the **P** vs. **NP** question (see Chapter 2) and average-case complexity (see Chapters 15).

5. Can we use the counterintuitive quantum mechanical properties of our universe to solve hard problems faster?

   Chapter 20 describes the fascinating notion of *quantum computers* that use such properties to speed up certain computations. Although there are many theoretical and practical obstacles to actually building such computers, they have generated tremendous interest in recent years. This is not least due to Shor's algorithm that showed that, if built, quantum computers will be able to factor integers efficiently. (Thus breaking many of the currently used cryptosystems.)

6. Can we generate mathematical proofs automatically? Can we check a mathematical proof by only reading 3 probabilistically chosen letters from it? 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 18 describes *probabilistically checkable proofs*. These are surprisingly robust mathematical proofs that can checked by only reading them in very few probabilistically chosen locations. *Interactive proofs* are studied in Chapter 8. Finally, *proof complexity*, a subfield of complexity studying the minimal proof length of various statements, is studied in Chapter 21.

At roughly 40 years of age, Complexity theory is still an infant science. Thus we still do not have complete answers for any of these questions. (In a surprising twist, computational complexity has also been used to provide evidence for the hardness to solve some of the questions of ... computational complexity; see Chapter 22.) Furthermore, many major insights on these questions were only found in recent years.

Meaning of efficiency

Now we explain the notion of *computational efficiency*, using the three examples for 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 requires only 3 additions and 3 multiplications of a number by a single digit.

We will quantify the efficiency of an algorithm by studying the number of *basic operations* it performs as the *size* of the input increases. Here, the *basic operations* are addition and multiplication of single digits. (In other settings, we may wish to throw in division as a basic operation.) The

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

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

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 $cn \log n$ operations where c is some absolute constant independent of n. (Using asymptotic notation, we call this an $O(n \log n)$-step algorithm; see Chapter 1.)

Similarly, 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.

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 are not on speaking terms. 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. In fact, as we will see in Chapter 2, we have reasons to suspect that no efficient algorithm *exists* for this task. We will see that it is equivalent to the *independent set* computational problem, which, together with thousands of other important problems, is **NP**-complete. The famous "**P** versus **NP**" question asks whether or not any of these problems has an efficient algorithm.

Proving nonexistence of efficient algorithms

We have seen that sometimes computational tasks have nonintuitive algorithms that are more efficient than algorithms that were known 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 $O(n \log n)$-step algorithm for multiplication can never be improved (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.

Since we cannot very well check every one of the infinitely many possible algorithms, the only way to verify that the current algorithm is the best is to *mathematically prove* that there is no better algorithm. This may indeed be possible to do, since computation can be given a mathematically precise model. There are several precedents for proving *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.

Given the above discussion, it is no surprise that mathematical proofs are the main tool of complexity theory, and that this book is filled with theorems, definitions and lemmas. However, we hardly use any fancy mathematics and so the main prerequisite for this book is the ability to read (and perhaps even enjoy!) mathematical proofs. The reader might want to take a quick look at Appendix A, that reviews mathematical proofs and other notions used, and come back to it as needed.

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 necessary be susceptible of 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 ignorabimus.*

Conventions: A *whole number* is a number in the set $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$. A number denoted by one of the letters i, j, k, ℓ, m, n is always assumed to be whole. If $n \geq 1$, then we denote by $[n]$ the set $\{1, \ldots, n\}$. For a real number x, we denote by $\lceil x \rceil$ the smallest $n \in \mathbb{Z}$ such that $n \geq x$ and by $\lfloor x \rfloor$ the largest $n \in \mathbb{Z}$ such that $n \leq x$. Whenever we use a real number in a context requiring a whole number, the operator $\lceil \ \rceil$ is implied. We denote by $\log x$ the logarithm of x to the base 2. We say that a condition holds for *sufficiently large* n if it holds for every $n \geq N$ for some number N (for example, $2^n > 100n^2$ for sufficiently large n). We use expressions such as $\sum_i f(i)$ (as opposed to, say, $\sum_{i=1}^{n} f(i)$) when the range of values i takes is obvious from the context. If u is a string or vector, then u_i denotes the value of the i^{th} symbol/coordinate of u.