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 some examples of 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 notepad 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 (the "modern slide rule"), but also a major scientific concept. Generalizing from models such as cellular automata, many 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 21.

From 1930s to the 1950s, researchers focused on computation in the abstract and tried to understand its power. They developed a theory of which algorithmic problems are *computable*. Many interesting algorithmic tasks have been found to be *uncomputable* or *undecidable*: 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 Sipser [**?**] or Rogers [**?**].) Instead, we focus on issues of *computational efficiency*. *Computational complexity theory* asks the following simple question: how much computational resources are required to solve a given computational task? Below, we discuss the meaning of this question.

Though complexity theory is usually studied as part of Computer Science, the above discussion suggests that it will be of interest in many other disciplines. Since computational processes also arise in nature, understanding the resource requirements for computational tasks is a very natural scientific question. The notion of *proof* and *good characterization* are basic to mathematics, and many aspects of the famous **P** versus **NP** question have a bearing on such issues, as will be pointed out in several places in the book (see Chapters 2, 9 and 19). Optimization problems arise in a host of disciplines including the life sciences, social sciences and operations research. Complexity theory provides strong evidence that, like the independent set problem, many other optimization problems are likely to be *intractable* and have no efficient algorithm (see Chapter 2). Our society increasingly relies every day on digital cryptography, which is based upon the (presumed) computational difficulty of certain problems (see Chapter 10). Randomness and statistics, which revolutionized several sciences including social sciences, acquire an entirely new meaning once one throws in the notion of computation (see Chapters 7 and 17). In physics, questions about intractability and quantum computation may help to shed light on the fundamental properties of matter (see Chapter 21).

Meaning of efficiency

Now we explain the notion of *computational efficiency* and give examples.

A simple example, hinted at in Cobham's quote at the start of the chapter, concerns multiplying two integers. Consider two different methods (or *algorithms*) for this task. The first is *repeated addition*: to compute $a \cdot b$, just add a to itself b times. The other is the *gradeschool algorithm* illustrated in Figure 1. Though the repeated addition algorithm is perhaps simpler than the gradeschool algorithm, we somehow feel that the latter is *better*. Indeed, it is much more efficient. For example, multiplying 577 and 423 by repeated addition requires 577 additions, whereas doing it with the gradeschool 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 single-digit addition and multiplication. (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 gradeschool 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 gradeschool algorithm would beat the best current supercomputers 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*

$$\begin{array}{cccccc}
 & & & 4 & 2 & 3 \\
 & & & 5 & 7 & 7 \\
\hline
 & & 2 & 9 & 6 & 1 \\
 & 2 & 9 & 6 & 1 & \\
2 & 1 & 1 & 5 & & \\
\hline
2 & 4 & 3 & 0 & 7 & 1
\end{array}$$

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

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 on n. Using the familiar asymptotic notation, we call this an $O(n \log n)$-step algorithm.

Similarly, for the problem of solving linear equations, the classic *Gaussian elimination* algorithm (named after Gauss but 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* problem 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 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 problem. We will see that this problem is equivalent to the *independent set* problem, which, together with thousands of other famous 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 problems have nonintuitive algorithms, which are quantifiably better (i.e., more efficient) than algorithms that were known for thousands of years. It would therefore be really interesting to prove for interesting 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 problem that takes fewer than $2^{n/10}$ steps.

It may be possible to *mathematically prove* such statements, since computation is a mathematically precise notion. 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. Impossibility proofs are among the most interesting, fruitful, and surprising results in mathematics.

Subsequent chapters of this book identify many interesting questions about the inherent computational complexity of tasks, usually with respect to the *Turing Machine* model. Most such questions are still unanswered, but tremendous progress has been made in the past few decades in showing that many of the questions are *interrelated*, sometimes in unexpected ways. This interrelationship is usually exhibited using a *reduction*. For an intriguing example of this, see the last chapter (Chapter 23), which uses computational complexity to explain why we are stuck in resolving the central open questions concerning computational complexity.

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.