1

INTRODUCTION: WHAT CAN AND CANNOT BE COMPUTED?

> There cannot be any [truths] that are so remote that they are not eventually reached nor so hidden that they are not discovered.
>
> —René Descartes, *Discourse on the Method for Conducting One's Reason Well and for Seeking the Truth in the Sciences* (1637)

The relentless march of computing power is a fundamental force in modern society. Every year, computer hardware gets better and faster. Every year, the software algorithms running on this hardware become smarter and more effective. So it's natural to wonder whether there are any limits to this progress. Is there anything that computers can't do? More specifically, is there anything that computers will *never* be able to do, no matter how fast the hardware or how smart the algorithms?

Remarkably, the answer to this question is a definite yes: contrary to the opinion of René Descartes in the quotation above, there *are* certain tasks that computers will never be able to perform. But that is not the whole story. In fact, computer scientists have an elegant way of classifying computational problems according to whether they can be solved effectively, ineffectively, or not at all. Figure 1.1 summarizes these three categories of computational problems, using more careful terminology: *tractable* for problems that can be solved efficiently; *intractable* for problems whose only methods of solution are hopelessly time consuming; and *uncomputable* for problems that cannot be solved by any computer program. The main purpose of this book is that we understand how and why different computational problems fall into these three different categories. The next three sections give an overview of each category in turn, and point out how later chapters will fill in our knowledge of these categories.

	Tractable problems	Intractable problems	Uncomputable problems
Description	can be solved efficiently	method for solving exists but is hopelessly time consuming	cannot be solved by any computer program
Computable in theory	✓	✓	×
Computable in practice	✓	× (?)	×
Example	shortest route on a map	decryption	finding all bugs in computer programs

Figure 1.1: Three major categories of computational problems: tractable, intractable, and uncomputable. The question mark in the middle column reminds us that certain problems that are believed to be intractable have not in fact been proved intractable—see page 5.

1.1 TRACTABLE PROBLEMS

As we can see from figure 1.1, a computational problem is *tractable* if we can solve it efficiently. Therefore, it's computable not only in theory, but also in practice. We might be tempted to call these "easy" problems, but that would be unfair. There are many tractable computational problems for which we have efficient methods of solution only because of decades of hard work by computer scientists, mathematicians, and engineers. Here are just a few of the problems that sound hard, but are in fact tractable:

- **Shortest path**. Given the details of a road network, find the shortest route between any two points. Computers can quickly find the optimal solution to this problem even if the input consists of every road on earth.
- **Web search**. Given the content of all pages on the World Wide Web, and a query string, produce a list of the pages most relevant to the query. Web search companies such as Google have built computer systems that can solve this problem in a fraction of a second.
- **Error correction**. Given some content (say, a large document or software package) to be transmitted over an unreliable network connection (say, a wireless network with a large amount of interference), encode the content so it can be transmitted with a negligible chance of any errors or omissions occurring. Computers, phones, and other devices are constantly using error correcting codes to solve this problem, which can in fact be achieved efficiently and with essentially perfect results.

In this book, we will discover that the notion of "tractable" has no precise scientific definition. But computer scientists have identified some important underlying properties that contribute to the tractability of a problem. Of these, the most important is that the problem can be solved in *polynomial time*.

Chapter 11 defines polynomial-time problems, and the related *complexity classes* Poly and P.

1.2 INTRACTABLE PROBLEMS

The middle column of figure 1.1 is devoted to problems that are *intractable*. This means that there is a program that can compute the answer, but the program is too slow to be useful—more precisely, it takes too long to solve the problem on large inputs. Hence, these problems can be solved in theory, but not in practice (except for small inputs). Intractable problems include the following examples:

- **Decryption**. Given a document encrypted with a modern encryption scheme, and *without* knowledge of the decryption key, decrypt the document. Here, the difficulty of decryption depends on the size of the decryption key, which is generally thousands of bits long in modern implementations. Of course, an encryption scheme would be useless if the decryption problem were tractable, so it should be no surprise that decryption is believed to be intractable for typical key sizes. For the schemes in common use today, it would require at least billions of years, even using the best known algorithms on the fastest existing supercomputer, to crack an encryption performed with a 4000-bit key.
- **Multiple sequence alignment**. Given a collection of DNA fragments, produce an alignment of the fragments that maximizes their similarity. An *alignment* is achieved by inserting spaces anywhere in the fragments, and we deliberately omit the precise definition of "optimal" alignment here. A simple example demonstrates the main idea instead. Given the inputs CGGATTA, CAGGGATA, and CGCTA, we can align them almost perfectly as follows:

```
C GG ATTA
CAGGGAT A
C G  CT A
```

  This is an important problem in genetics, but it turns out that when the input consists of a large number of modest-sized fragments, the best known algorithms require at least billions of years to compute an optimal solution, even using the fastest existing supercomputer.

Just as with "tractable," there is no precise scientific definition of "intractable." But again, computer scientists have uncovered certain properties that strongly suggest intractability. Chapters 10 and 11 discuss *superpolynomial* and *exponential* time. Problems that require superpolynomial time are almost always regarded as intractable. Chapter 14 introduces the profound notion of *NP-completeness*, another property that is associated with intractability. It's widely believed that NP-complete problems cannot be solved in polynomial time, and are therefore intractable. But this claim depends on the most notorious unsolved problem in computer science, known as "P versus NP." The unresolved nature of P versus NP explains the presence of a question mark ("?") in the middle column of figure 1.1: it represents the lack of complete certainty about whether certain problems are

intractable. Chapter 14 explains the background and consequences of the P versus NP question.

1.3 UNCOMPUTABLE PROBLEMS

The last column of figure 1.1 is devoted to problems that are *uncomputable*. These are problems that cannot be solved by any computer program. They cannot be solved in practice, and they cannot be solved in theory either. Examples include the following:

- **Bug finding**. Given a computer program, find all the bugs in the program. (If this problem seems too vague, we can make it more specific. For example, if the program is written in Java, the task could be to find all locations in the program that will throw an exception.) It's been proved that no algorithm can find all the bugs in all programs.
- **Solving integer polynomial equations**. Given a collection of polynomial equations, determine whether there are any integer solutions to the equations. (This may sound obscure, but it's actually an important and famous problem, known as Hilbert's 10th Problem. We won't be pursuing polynomial equations in this book, so there's no need to understand the details.) Again, it's been proved that no algorithm can solve this problem.

It's important to realize that the problems above—and many, many others—have been *proved* uncomputable. These problems aren't just hard. They are literally impossible. In chapters 3 and 7, we will see how to perform these impossibility proofs for ourselves.

1.4 A MORE DETAILED OVERVIEW OF THE BOOK

The goal of this book is that we to understand the three columns of figure 1.1: that is, we understand why certain kinds of problems are tractable, intractable, or uncomputable. The boundary between computable and uncomputable problems involves the field of *computability theory*, and is covered in part I of the book (chapters 2–9). The boundary between tractable and intractable problems is the subject of *complexity theory*; this is covered in part II of the book (chapters 10–14). Part III examines some of the origins and applications of computability and complexity theory. The sections below give a more detailed overview of each of these three parts.

This is a good time to mention a stylistic point: the book doesn't include explicit citations. However, the bibliography at the end of the book includes full descriptions of the relevant sources for every author or source mentioned in the text. For example, note the bibliographic entries for Descartes and Hilbert, both of whom have already been mentioned in this introductory chapter.

Overview of part I: Computability theory

Part I asks the fundamental question, which computational problems can be solved by writing computer programs? Of course, we can't get far without formal

definitions of the two key concepts: "problems" and "programs." Chapter 2 kicks this off by defining and discussing computer programs. This chapter also gives a basic introduction to the Python programming language—enough to follow the examples used throughout the book. Chapter 3 plunges directly into one of the book's most important results: we see our first examples of programs that are impossible to write, and learn the techniques needed to prove these programs can't exist. Up to this point in the book, mathematical formalism is mostly avoided. This is done so that we can build an intuitive understanding of the book's most fundamental concepts without any unnecessary abstraction. But to go further, we need some more formal concepts. So chapter 4 gives careful definitions of several concepts, including the notion of a "computational problem," and what it means to "solve" a computational problem. At this point, we'll be ready for some of the classical ideas of computability theory:

- Turing machines (chapter 5). These are the most widely studied formal models of computation, first proposed by Alan Turing in a 1936 paper that is generally considered to have founded the discipline of theoretical computer science. We will see that Turing machines are equivalent to Python programs in terms of what problems they can solve.
- Universal computer programs (chapter 6). Some programs, such as your own computer's operating system, are capable of running essentially any other program. Such "universal" programs turn out to have important applications and also some philosophical implications.
- Reductions (chapter 7). The technique of "reducing" problem X to problem Y (i.e., using a solution for Y to solve X) can be used to show that many interesting problems are in fact uncomputable.
- Nondeterminism (chapter 8). Some computers can perform several actions simultaneously or make certain arbitrary choices about what action to take next, thus acting "nondeterministically." This behavior can be modeled formally and has some interesting consequences.
- Finite automata (chapter 9). This is a model of computation even simpler than the Turing machine, which nevertheless has both theoretical and practical significance.

Overview of part II: Complexity theory

Part II addresses the issue of which computational problems are tractable—that is, which problems have efficient methods of solution. We start in chapter 10 with the basics of complexity theory: definitions of program running times, and discussions of which computational models are appropriate for measuring those running times. Chapter 11 introduces the two most fundamental complexity classes. These are Poly (consisting of problems that can be solved in polynomial time) and Expo (consisting of problems that can be solved in exponential time). Chapter 12 introduces PolyCheck, an extremely important complexity class with a somewhat strange definition. PolyCheck consists of problems that might themselves be extremely hard to solve, but whose solutions can be efficiently verified once they are found. Chapter 12 also examines two classes that are closely related to PolyCheck: NPoly and NP. A crucial tool in proving whether problems are "easy" or "hard" is the *polynomial-time mapping reduction*; this is the main topic

of chapter 13. The chapter also covers three classic problems that lie at the heart of complexity theory: CIRCUITSAT, SAT, and 3-SAT. Thus equipped, chapter 14 brings us to the crown jewel of complexity theory: NP-completeness. We'll discover a huge class of important and intensively studied problems that are all, in a certain sense, "equally hard." These NP-complete problems are believed—but not yet proved—to be intractable.

Overview of part III: Origins and applications

Part III takes a step back to examine some origins and applications of computability and complexity theory. In chapter 15, we examine Alan Turing's revolutionary 1936 paper, "On computable numbers." We'll understand the original definition of the now-famous Turing machine, and some of the philosophical ideas in the paper that still underpin the search for artificial intelligence. In chapter 16, we see how Turing's ideas can be used to prove important facts about the foundations of mathematics—including Gödel's famous incompleteness theorem, which states there are true mathematical statements that can't be proved. And in chapter 17, we look at the extraordinary 1972 paper by Richard Karp. This paper described 21 NP-complete problems, catalyzing the rampage of NP-completeness through computer science that still reverberates to this day.

1.5 PREREQUISITES FOR UNDERSTANDING THIS BOOK

To understand this book, you need two things:

- **Computer programming**. You should have a reasonable level of familiarity with writing computer programs. It doesn't matter which programming language(s) you have used in the past. For example, some knowledge of any one of the following languages would be good preparation: Java, C, C++, C#, Python, Lisp, Scheme, JavaScript, Visual Basic. Your level of programming experience should be roughly equivalent to one introductory college-level computer science course, or an advanced high-school computer science course. You need to understand the basics of calling methods or functions with parameters; the distinction between elementary data types like strings and integers; use of arrays and lists; control flow using if statements and for loops; and basic use of recursion. The practical examples in this book use the Python programming language, but you don't need any background in Python before reading the book. We will be using only a small set of Python's features, and each feature is explained when it is introduced. The online book materials also provide Java versions of the programs.
- **Some math**. Proficiency with high-school math is required. You will need familiarity with functions like x^3, 2^x, and $\log x$. Calculus is not required. Your level of experience should be roughly equivalent to a college-level pre-calculus course, or a moderately advanced high-school course.

The two areas above are the only *required* bodies of knowledge for understanding the book. But there are some other spheres of knowledge where some

previous experience will make it even easier to acquire a good understanding of the material:

- **Proof writing**. Computability theory and complexity theory form the core of theoretical computer science, so along the way we will learn the main tools used by theoretical computer scientists. Mostly, these tools are mathematical in nature, so we will be using some abstract, theoretical ideas. This will include stating formal theorems, and proving those theorems rigorously. Therefore, you may find it easier to read this book if you have first studied proof writing. Proof writing is often taught at the college level in a discrete mathematics course, or sometimes in a course dedicated to proof writing. Note that this book does not *assume* you have studied proof writing. All the necessary proof techniques are explained before they are used. For example, proof by contradiction is covered in detail in section 3.1. Proof by induction is not used in this book.
- **Algorithm analysis and big-*O* notation**. Part II of the book relies heavily on analyzing the running time of computer programs, often using big-*O* notation. Therefore, prior exposure to some basics of program analysis using big-*O* (which is usually taught in a first or second college-level computer science course) could be helpful. Again, note that the book does not *assume* any knowledge of big-*O* notation or algorithm analysis: sections 10.2 and 10.3 provide detailed explanations.

1.6 THE GOALS OF THE BOOK

The book has one fundamental goal, and two additional goals. Each of these goals is described separately below.

The fundamental goal: What can be computed?

The most fundamental goal of the book has been stated already: we want to understand why certain kinds of problems are tractable, intractable, or uncomputable. That explains the title of the book: *What Can Be Computed?* Quite literally, part I answers this question by investigating classes of problems that are computable and uncomputable. Part II answers the question in a more nuanced way, by addressing the question of what can be computed efficiently, in practice. We discover certain classes of problems that can be proved intractable, others that are widely believed to be intractable, and yet others that are tractable.

Secondary goal 1: A practical approach

In addition to the primary goal of understanding what can be computed, the book has two secondary goals. The first of these relates to *how* we will gain our understanding: it will be acquired in a *practical* way. That explains the book's subtitle, *A Practical Guide to the Theory of Computation*. Clearly, the object of our study is the theory of computation. But our understanding of the *theory* of computation is enhanced when it is linked to the *practice* of using computers.

Therefore, an important goal of this book is to be ruthlessly practical whenever it's possible to do so. The following examples demonstrate some of the ways that we emphasize practice in the theory of computation:

- Our main computational model is Python programs, rather than Turing machines (although we do study both models carefully, using Turing machines when mathematical rigor requires it).
- We focus on real computational problems, rather than the more abstract "decision problems," which are often the sole focus of computational theory. For more details, see the discussion on page 59.
- We start off with the most familiar computational model—computer programs—and later progress to more abstract models such as Turing machines and finite automata.

Secondary goal 2: Some historical insight

The other secondary goal of the book is to provide some historical insight into how and why the theory of computation developed. This is done in part III of the book. There are chapters devoted to Turing's original 1936 paper on computability, and to Karp's 1972 paper on NP-completeness. Sandwiched between these is a chapter linking Turing's work to the foundations of mathematics, and especially the incompleteness theorems of Gödel. This is important both in its own right, and because it was Turing's original motivation for studying computability. Of course, these chapters touch on only some small windows into the full history of computability theory. But within these small windows we are able to gain genuine historical insight. By reading excerpts of the original papers by Turing and Karp, we understand the chaotic intellectual landscape they faced—a landscape vastly different to today's smoothly manicured, neatly packaged theory of computation.

1.7 WHY STUDY THE THEORY OF COMPUTATION?

Finally, let's address the most important question: *Why* should we learn about the theory of computation? Why do we need to know "what can be computed"? There are two high-level answers to this: (a) it's useful and (b) the ideas are beautiful and important. Let's examine these answers separately.

Reason 1: The theory of computation is useful

Computer scientists frequently need to solve computational problems. But what is a good strategy for doing so? In school and college, your instructor usually assigns problems that can be solved using the tools you have just learned. But the real world isn't so friendly. When a new problem presents itself, some fundamental questions must be asked and answered. Is the problem computable? If not, is some suitable variant or approximation of the problem computable? Is the problem tractable? If not, is some suitable variant or approximation of the problem tractable? Once we have a tractable version of the problem, how can we

compare the efficiency of competing methods for solving it? To ask and answer each of these questions, you will need to know something about the theory of computation.

In addition to this high-level concept of usefulness, the theory of computation has more specific applications too, including the following:

- Some of the techniques for Turing reductions (chapter 7) and polynomial-time mapping reductions (chapter 13) are useful for transforming real-world problems into others that have been previously solved.
- Regular expressions (chapter 9) are often used for efficient and accurate text-processing operations.
- The theory of compilers and other language-processing tools depends heavily on the theory of automata.
- Some industrial applications (e.g., circuit layout) employ heuristic methods for solving NP-complete problems. Understanding and improving these heuristic methods (e.g., SAT-solvers) can be helped by a good understanding of NP-completeness.

Let's not overemphasize these arguments about the "usefulness" of the theory of computation. It is certainly possible to be successful in the technology industry (say, a senior software architect or database administrator) with little or no knowledge of computability and complexity theory. Nevertheless, it seems clear that a person who does have this knowledge will be better placed to grow, adapt, and succeed in any job related to computer science.

Reason 2: The theory of computation is beautiful and important

The second main reason for studying the theory of computation is that it contains profound ideas—ideas that deserve to be studied for their beauty alone, and for their important connections to other disciplines such as philosophy and mathematics. These connections are explicit even in Turing's 1936 "On computable numbers" paper, which was the very first publication about computability. Perhaps you will agree, after reading this book, that the ideas in it have the same qualities as great poetry, sculpture, music, and film: they are beautiful, and they are worth studying for their beauty.

It's worth noting, however, that our two reasons for studying the theory of computation (which could be summarized roughly as "usefulness" and "beauty") are by no means distinct. Indeed, the two justifications overlap and reinforce each other. The great Stanford computer scientist Donald Knuth once wrote, "We have some freedom in setting up our personal standards of beauty, but it is especially nice when the things we regard as beautiful are also regarded by other people as useful." So, I hope you will find the ideas in the rest of the book as useful and beautiful as I do.

EXERCISES

1.1 Give one example of each of the following types of problems: (i) tractable, (ii) intractable, and (iii) uncomputable. Describe each problem in 1 to 2

sentences. Don't use examples that have already been described in this chapter—do some research to find different examples.

1.2 In a few sentences of your own words, describe why you are interested in studying the theory of computation. Which, if any, of the reasons given in section 1.7 do you feel motivated by?

1.3 Part II of the book explores the notion of intractability carefully, but this exercise gives some informal insight into why certain computational problems can be computable, yet intractable.

(a) Suppose you write a computer program to decrypt an encrypted password using "brute force." That is, your program tries every possible encryption key until it finds the key that successfully decrypts the password. Suppose that your program can test one billion keys per second. On average, how long would the program need if the key is known to be 30 bits long? What about 200-bit keys, or 1000-bit keys? Compare your answers with comprehensible units, such as days, years, or the age of the universe. In general, each time we add a single bit to the length of the key, what happens to the expected running time of your program?

(b) Consider the following simplified version of the multiple sequence alignment problem defined on page 5. We are given a list of 5 genetic strings each of length 10, and we seek an alignment that inserts exactly 3 spaces in each of the strings. We use a computer program to find the best alignment using brute force: that is, we test every possible way of inserting 3 spaces into the 10-character strings. Approximately how many possibilities need to be tested? If we can test one billion possibilities per second, what is the running time of the program? What if there are 20 genetic strings instead of 5? If there are N genetic strings, what is the approximate running time in terms of N?

1.4 Consult a few different sources for the definition of "tractable" as it relates to computational problems. Compare and contrast the definitions, paying particular attention to the definition given in this chapter.

1.5 Interview someone who studied computer science at college and has now graduated. Ask them if they took a course on the theory of computation in college. If so, what do they remember about it, and do they recommend it to you? If not, do they regret not taking a computational theory course?