

CONTENTS



<i>Acknowledgments</i>	xiii
<i>Preface for instructors</i>	xv
The inspiration of GEB	xv
Which “theory” course are we talking about?	xv
The features that might make this book appealing	xvi
What’s in and what’s out	xvii
Possible courses based on this book	xvii
Computer science as a liberal art	xviii

OVERVIEW 1

1 INTRODUCTION: WHAT CAN AND CANNOT BE COMPUTED? 3

1.1 Tractable problems	4
1.2 Intractable problems	5
1.3 Uncomputable problems	6
1.4 A more detailed overview of the book	6
Overview of part I: Computability theory	6
Overview of part II: Complexity theory	7
Overview of part III: Origins and applications	8
1.5 Prerequisites for understanding this book	8
1.6 The goals of the book	9
The fundamental goal: What can be computed?	9
Secondary goal 1: A practical approach	9
Secondary goal 2: Some historical insight	10
1.7 Why study the theory of computation?	10
Reason 1: The theory of computation is useful	10
Reason 2: The theory of computation is beautiful and important	11
Exercises	11

Part I: COMPUTABILITY THEORY 13

2 WHAT IS A COMPUTER PROGRAM? 15

2.1 Some Python program basics	15
Editing and rerunning a Python program	17
Running a Python program on input from a file	17
Running more complex experiments on Python programs	18
2.2 SISO Python programs	18
Programs that call other functions and programs	20
2.3 ASCII characters and multiline strings	21
2.4 Some problematic programs	22

2.5	Formal definition of Python program	23
2.6	Decision programs and equivalent programs	25
2.7	Real-world programs versus SISO Python programs	26
	Exercises	27
3	SOME IMPOSSIBLE PYTHON PROGRAMS	30
3.1	Proof by contradiction	30
3.2	Programs that analyze other programs	31
	Programs that analyze themselves	33
3.3	The program <code>yesOnString.py</code>	33
3.4	The program <code>yesOnSelf.py</code>	34
3.5	The program <code>notYesOnSelf.py</code>	36
3.6	<code>yesOnString.py</code> can't exist either	37
	A compact proof that <code>yesOnString.py</code> can't exist	37
3.7	Perfect bug-finding programs are impossible	39
3.8	We can still find bugs, but we can't do it perfectly	41
	Exercises	42
4	WHAT IS A COMPUTATIONAL PROBLEM?	45
4.1	Graphs, alphabets, strings, and languages	46
	Graphs	46
	Trees and rooted trees	49
	Alphabets	49
	Strings	50
	Languages	51
4.2	Defining computational problems	53
	Positive and negative instances	55
	Notation for computational problems	56
4.3	Categories of computational problems	57
	Search problems	57
	Optimization problems	57
	Threshold problems	57
	Function problems	58
	Decision problems	58
	Converting between general and decision problems	59
	Complement of a decision problem	60
	Computational problems with two input strings	61
4.4	The formal definition of "solving" a problem	62
	Computable functions	63
4.5	Recognizing and deciding languages	63
	Recognizable languages	65
	Recursive and recursively enumerable languages	66
	Exercises	66
5	TURING MACHINES: THE SIMPLEST COMPUTERS	71
5.1	Definition of a Turing machine	72
	Halting and looping	76
	Accepters and transducers	77

Abbreviated notation for state diagrams	78
Creating your own Turing machines	78
5.2 Some nontrivial Turing machines	79
The <code>moreCsThanGs</code> machine	80
The <code>countCs</code> machine	81
Important lessons from the <code>countCs</code> example	85
5.3 From single-tape Turing machines to multi-tape Turing machines	86
Two-tape, single-head Turing machines	87
Two-way infinite tapes	88
Multi-tape, single-head Turing machines	89
Two-tape, two-head Turing machines	90
5.4 From multi-tape Turing machines to Python programs and beyond	91
Multi-tape Turing machine \rightarrow random-access Turing machine	92
Random-access Turing machine \rightarrow real computer	92
Modern computer \rightarrow Python program	95
5.5 Going back the other way: Simulating a Turing machine with Python	95
A serious caveat: Memory limitations and other technicalities	97
5.6 Classical computers can simulate quantum computers	98
5.7 All known computers are Turing equivalent	98
Exercises	99
6 UNIVERSAL COMPUTER PROGRAMS: PROGRAMS THAT CAN DO ANYTHING	103
6.1 Universal Python programs	104
6.2 Universal Turing machines	105
6.3 Universal computation in the real world	107
6.4 Programs that alter other programs	110
Ignoring the input and performing a fixed calculation instead	112
6.5 Problems that are undecidable but recognizable	113
Exercises	114
7 REDUCTIONS: HOW TO PROVE A PROBLEM IS HARD	116
7.1 A reduction for easiness	116
7.2 A reduction for hardness	118
7.3 Formal definition of Turing reduction	120
Why “Turing” reduction?	120
Oracle programs	121
Why is \leq_T used to denote a Turing reduction?	121
Beware the true meaning of “reduction”	121
7.4 Properties of Turing reductions	122
7.5 An abundance of uncomputable problems	123
The variants of <code>YESONSTRING</code>	123
The halting problem and its variants	126
Uncomputable problems that aren’t decision problems	128
7.6 Even more uncomputable problems	130
The computational problem <code>COMPUTES_F</code>	132
Rice’s theorem	134
7.7 Uncomputable problems that aren’t about programs	134

7.8	Not every question about programs is uncomputable	135
7.9	Proof techniques for uncomputability	136
	Technique 1: The reduction recipe	137
	Technique 2: Reduction with explicit Python programs	138
	Technique 3: Apply Rice’s theorem	139
	Exercises	140
8	NONDETERMINISM: MAGIC OR REALITY?	143
8.1	Nondeterministic Python programs	144
8.2	Nondeterministic programs for nonddecision problems	148
8.3	Computation trees	149
8.4	Nondeterminism doesn’t change what is computable	153
8.5	Nondeterministic Turing machines	154
8.6	Formal definition of nondeterministic Turing machines	156
8.7	Models of nondeterminism	158
8.8	Unrecognizable problems	158
8.9	Why study nondeterminism?	159
	Exercises	160
9	FINITE AUTOMATA: COMPUTING WITH LIMITED RESOURCES	164
9.1	Deterministic finite automata	164
9.2	Nondeterministic finite automata	167
	State diagrams for nfas	168
	Formal definition of an nfa	169
	How does an nfa accept a string?	170
	Sometimes nfas make things easier	170
9.3	Equivalence of nfas and dfas	170
	Nondeterminism can affect computability: The example of pdas	173
	Practicality of converted nfas	174
	Minimizing the size of dfas	175
9.4	Regular expressions	175
	Pure regular expressions	176
	Standard regular expressions	177
	Converting between regexes and finite automata	178
9.5	Some languages aren’t regular	181
	The nonregular language G_nT_n	181
	The key difference between Turing machines and finite automata	183
9.6	Many more nonregular languages	183
	The pumping lemma	185
9.7	Combining regular languages	187
	Exercises	188
	Part II: COMPUTATIONAL COMPLEXITY THEORY	193
10	COMPLEXITY THEORY: WHEN EFFICIENCY DOES MATTER	195
10.1	Complexity theory uses asymptotic running times	195
10.2	Big- O notation	197
	Dominant terms of functions	199
	A practical definition of big- O notation	201

Superpolynomial and subexponential	202
Other asymptotic notation	202
Composition of polynomials is polynomial	203
Counting things with big- O	203
10.3 The running time of a program	204
Running time of a Turing machine	204
Running time of a Python program	206
The lack of rigor in Python running times	209
10.4 Fundamentals of determining time complexity	210
A crucial distinction: The length of the input versus the numerical value of the input	210
The complexity of arithmetic operations	212
Beware of constant-time arithmetic operations	214
The complexity of factoring	215
The importance of the hardness of factoring	216
The complexity of sorting	217
10.5 For complexity, the computational model <i>does</i> matter	217
Simulation costs for common computational models	217
Multi-tape simulation has quadratic cost	218
Random-access simulation has cubic cost	219
Universal simulation has logarithmic cost	219
Real computers cost only a constant factor	220
Python programs cost the same as real computers	220
Python programs can simulate random-access Turing machines efficiently	220
Quantum simulation may have exponential cost	220
All classical computational models differ by only polynomial factors	221
Our standard computational model: Python programs	221
10.6 Complexity classes	221
Exercises	224
11 Poly AND Expo: THE TWO MOST FUNDAMENTAL COMPLEXITY CLASSES	228
11.1 Definitions of Poly and Expo	228
Poly and Expo compared to P, Exp, and FP	229
11.2 Poly is a subset of Expo	230
11.3 A first look at the boundary between Poly and Expo	231
ALL3SETS and ALLSUBSETS	231
Traveling salespeople and shortest paths	232
Multiplying and factoring	235
Back to the boundary between Poly and Expo	235
Primality testing is in Poly	237
11.4 Poly and Expo don't care about the computational model	238
11.5 HALTEX: A decision problem in Expo but not Poly	238
11.6 Other problems that are outside Poly	243
11.7 Unreasonable encodings of the input affect complexity	244
11.8 Why study Poly, really?	245
Exercises	246

12	PolyCheck AND NPoly: HARD PROBLEMS THAT ARE EASY TO VERIFY	250
12.1	Verifiers	250
	Why “unsure”?	253
12.2	Polytime verifiers	254
	Bounding the length of proposed solutions and hints	255
	Verifying negative instances in exponential time	255
	Solving arbitrary instances in exponential time	255
12.3	The complexity class PolyCheck	256
	Some PolyCheck examples: PACKING, SUBSETSUM, and PARTITION	256
	The haystack analogy for PolyCheck	257
12.4	The complexity class NPoly	258
12.5	PolyCheck and NPoly are identical	259
	Every PolyCheck problem is in NPoly	259
	Every NPoly problem is in PolyCheck	260
12.6	The PolyCheck/NPoly sandwich	263
12.7	Nondeterminism <i>does</i> seem to change what is computable <i>efficiently</i>	264
12.8	The fine print about NPoly	265
	An alternative definition of NPoly	265
	NPoly compared to NP and FNP	266
	Exercises	268
13	POLYNOMIAL-TIME MAPPING REDUCTIONS: PROVING X IS AS EASY AS Y	272
13.1	Definition of polytime mapping reductions	272
	Polyreducing to nondcision problems	275
13.2	The meaning of polynomial-time mapping reductions	275
13.3	Proof techniques for polyreductions	276
13.4	Examples of polyreductions using Hamilton cycles	277
	A polyreduction from UHC to DHC	278
	A polyreduction from DHC to UHC	279
13.5	Three satisfiability problems: CIRCUITSAT, SAT, and 3-SAT	281
	Why do we study satisfiability problems?	281
	CIRCUITSAT	281
	SAT	282
	Conjunctive normal form	284
	ASCII representation of Boolean formulas	284
	3-SAT	285
13.6	Polyreductions between CIRCUITSAT, SAT, and 3-SAT	285
	The Tseytin transformation	285
13.7	Polyequivalence and its consequences	290
	Exercises	291
14	NP-COMPLETENESS: MOST HARD PROBLEMS ARE EQUALLY HARD	294
14.1	P versus NP	294
14.2	NP-completeness	296
	Reformulations of P versus NP using NP-completeness	298
14.3	NP-hardness	298

14.4	Consequences of $P=NP$	301
14.5	CIRCUITSAT is a “hardest” NP problem	302
14.6	NP-completeness is widespread	306
14.7	Proof techniques for NP-completeness	308
14.8	The good news and bad news about NP-completeness	309
	Problems in $NPoly$ but probably not NP-hard	309
	Some problems that are in P	309
	Some NP-hard problems can be approximated efficiently	310
	Some NP-hard problems can be solved efficiently for real-world inputs	310
	Some NP-hard problems can be solved in pseudo-polynomial time	310
	Exercises	311
Part III: ORIGINS AND APPLICATIONS		315
15	THE ORIGINAL TURING MACHINE	317
15.1	Turing’s definition of a “computing machine”	318
15.2	Machines can compute what humans can compute	324
15.3	The Church–Turing thesis: A law of nature?	327
	The equivalence of digital computers	327
	Church’s thesis: The equivalence of computer programs and algorithms	328
	Turing’s thesis: The equivalence of computer programs and human brains	329
	Church–Turing thesis: The equivalence of all computational processes	329
	Exercises	330
16	YOU CAN’T PROVE EVERYTHING THAT’S TRUE	332
	The history of computer proofs	332
16.1	Mechanical proofs	333
	Semantics and truth	336
	Consistency and completeness	338
	Decidability of logical systems	339
16.2	Arithmetic as a logical system	340
	Converting the halting problem to a statement about integers	341
	Recognizing provable statements about integers	343
	The consistency of Peano arithmetic	344
16.3	The undecidability of mathematics	345
16.4	The incompleteness of mathematics	346
16.5	What have we learned and why did we learn it?	349
	Exercises	350
17	KARP’S 21 PROBLEMS	353
17.1	Karp’s overview	353
17.2	Karp’s definition of NP-completeness	355
17.3	The list of 21 NP-complete problems	357

xii • Contents

17.4	Reductions between the 21 NP-complete problems	359
	Polyreducing SAT to CLIQUE	361
	Polyreducing CLIQUE to NODE COVER	363
	Polyreducing DHC to UHC	364
	Polyreducing SAT to 3-SAT	365
	Polyreducing KNAPSACK to PARTITION	365
17.5	The rest of the paper: NP-hardness and more	367
	Exercises	367
18	CONCLUSION: WHAT WILL BE COMPUTED?	370
18.1	The big ideas about what can be computed	370
	<i>Bibliography</i>	373
	<i>Index</i>	375

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?

INDEX



- 2-SAT, 285, 290
- 2TDCM, 321, 322
- 3-SAT, 8, 285, 287–290, 296, 358, 365
 - definition of, 284
- accept
 - for nfa, 170
 - for Python program, 25, 77
 - for Turing machine, 74
- accept state, 73
- accepter, 77
- addition, 213
- address, 92
- address tape, 92
- Adleman, Leonard, 216
- Agrawal, Agrawal, 237
- Aiken, Howard, 103, 107
- AKS algorithm, 237, 245, 367
- algorithm, 3, 9, 46, 81, 144, 172, 175, 178,
 - 197, 209, 210, 222, 246, 294, 310, 328,
 - 354, 371
- ALL3SETS, 232
- ALLSUBSETS, 232
- allSubsets.py, 233
- all3Sets.py, 233
- alphabet, 49, 74
- alterYesToComputesF.py, 133
- alterYesToGAGA.py, 119
- alterYesToHalt.py, 127
- AND gate, 282
- appendZero, 85
- Apple Inc., xviii
- approximation, 310
- architecture. *See* CPU architecture
- arithmetic operations, 212–215
- Arora, Sanjeev, 220
- artificial intelligence, 301, 329, 370
- ASCII, 21, 24, 26, 31, 48, 50, 51, 73, 93, 96,
 - 267, 283, 284, 341
- asymptotic notation, 202
- Austen, Jane, 332
- automata theory, xv
- automaton, 164
- average-case, 196
- axiom, 334
- Babai, László, 309
- Babbage, Charles, 71
- Barak, Boaz, 220
- base of logarithm, 197, 199, 201
- basic function, 197
- basic term. *See* term
- BEGINSWITHA, 54
- big-*O* notation, xvii, 9, 196–204, 218
 - formal definition, 198
 - practical definition, 201
- big-Omega, 202
- big-Theta, 202
- BinAd, 333
- binAd.py, 335
- BinAdLogic, 337
- binary strings, 51
- binaryIncrementer, 83, 205
- blank symbol, 72–74, 94, 164–166, 319
 - dfas without, 166
- Boole, George, 332, 345
- Boolean formula, 282, 284
- BOTHCONTAINGAGA, 61
- brain, 33, 108, 329, 371
- BrokenBinAdLogic, 337
- brokenSort.py, 47
- bubble sort, 45
- bubbleSort.py, 47
- bug finding, 6, 39–41, 45
- building block
 - for Turing machine, 82, 84, 94
- calculus, xvii, 8, 198, 201, 203, 332
- cell, 72
- cellular automaton, 109, 164
- certificate, 266
- CHECKMULTIPLY, 60

- chess, generalized, 244
- Child, Julia, 116, 122
- Church's thesis, 328
- Church, Alonzo, 327–329, 346, 349
- Church–Turing thesis, 98, 327–330, 371
- circle-free machine, 321–323
- CIRCUITSAT, 8, 281–282, 285–288, 290, 296, 297, 302–307
 - definition of, 283
- circular machine, 321–322
- citations, 6
- claim, xvii
- classical computer, 98, 99, 220, 221
- clause, 284
 - splitting, 288
- CLIQUE, 361–364
- clique, 361
- clone. *See* Turing machine
- closed, 337, 341, 343
- CNF, 284
- code window, 17
- codomain, 53
- combinatorics, 203
- compiler, xvi, 11, 51, 68
- complement
 - of decision problem, 60
 - of language, 52
- complete, 338
 - defined by Karp, 354, 357
- complexity
 - circuit, xviii
 - space, xviii
 - time. *See* time complexity
- complexity class, xviii, 5, 7, 221–224, 265, 300
- complexity theory. *See* computational complexity
- composite, 237
- computability theory, 6–7, 195, 333
- computable
 - function, 63, 323
 - number, 318, 322–324
 - problem, 62
 - sequence, 322
- computation tree, 149–153, 261
 - growth of, 152
 - negative leaf, 150
 - nonterminating leaf, 150
 - positive leaf, 150
- computational complexity, xvi, 6–8, 195–227
- computational problem, 3, 7, 45–70
- compute, 62
- computer
 - as used by Turing, 324
- computer program, 3, 6, 10, 15–29, 32, 33, 45, 62, 134, 324, 328, 329. *See also* universal computation
- COMPUTES_F, 132–134
 - uncomputability of, 132
- COMPUTISEVEN, 131
- computing machine, 318–320
- concatenation, 50, 52, 176, 178
- configuration, 73, 319
 - complete, 319, 321
- conjunctive normal form. *See* CNF
- connected, 48
- consistent, 338
- Const, 222
- containsGAGA.py, 15, 16, 25, 33, 103, 105, 207, 239
- containsGAGAandCACAandTATA.py, 20
- containsGAGA, 77
- CONTAINSNANA, 145
- containsNANA.py, 145
- context-free language. *See* language
- control unit, 72
- convertPartitionToPacking.py, 274
- Conway, John, 110
- Cook, Stephen, 297, 302, 353, 355, 356
- Cook–Levin theorem, xviii, 356
- core. *See* CPU core
- countCs, 81–84, 209
- countLines.py, 22, 32
- CPU architecture, 23, 93–95
 - 16-bit, 32-bit, 64-bit, 89
- CPU core, 95, 143, 144, 153, 196, 264
- CPU instruction, 93, 206, 207, 209, 220
- crash, 39, 41
- crashOnSelf.py, 40
- CRASHONSTRING, 63
- crashOnString.py, 39
- cryptography, 159, 216, 217, 238, 245, 309, 371
- cycle, 47
- decidable
 - language, 64, 66, 113, 181
 - logical system, 339
 - problem, 62
- decide, 62
- decision problem, xvi, 10, 53, 56, 128, 230, 238, 265, 266, 272, 294, 300, 301, 356
 - definition of, 58
- decision program, 25

- decryption, 5
- def, 16
- deleteToInteger, 85
- Descartes, René, 3, 6, 164
- description
 - of dfa, 166
 - of Turing machine, 96, 106
- DESS, 61, 106, 111
- deterministic, 24, 144, 153, 295, 320, 354.
 - See also* dfa
- dfa, xviii, 164–167, 171, 174, 179–181
 - definition of, 164
 - minimization of, 175
- DHC, 277–281, 296, 358, 364–365
 - definition of, 278
- Diffie–Hellman key exchange, 309
- Dijkstra’s algorithm, 234
- Diophantine equations, 134. *See also* Hilbert’s 10th Problem
- DIRECTEDHAMILTONCYCLE.
 - See* DHC
- direction function, 73
- DISCRETELOG, 309
- division, 213
- DNA, 5, 16, 110, 329
- domain, 53
- dominant
 - function, 199
 - term, 200
- double exponential. *See* exponential
- DT, 200
- dtm. *See* Turing machine, deterministic

- ϵ -transition, 169
- Edmonds, Jack, 294
- Eliot, T. S., 21
- emulate, 86
- encryption, 5, 216, 309
- engineering, xix, 250, 371
- Entscheidungsproblem, 317, 318, 329, 346, 349
- equivalence
 - of dfa and regex, 180
 - of dfas and nfas, 167, 170
 - of Python programs, 26
 - of transducers, 77
- error correcting codes, 4
- ESS, 61, 65, 106, 111
- EULERCYCLE, 236
- exception, 6, 22, 24, 29, 39, 63, 126
- exec(), 104

- Exp, xviii, 229, 265
 - definition of, 294
- Exp0, 7, 223, 228–249, 263, 265
 - definition of, 228
- exponential
 - basic function, 197, 198
 - double, 200, 229, 246
 - time, 5, 223, 246, 354

- FACTOR, 215, 229, 235, 258, 309
- factor.py, 216
- factorial, 197
- factoring, 159, 212, 235, 309
 - complexity of, 215–216
 - decision version of, 237
 - importance of, 216
- FACTORINRANGE, 237
- FACTORUNARY, 244
- feasible packing, 257
- Feynman, Richard, 331
- file-system, 23
- final state, 78
- FINDNANA, 148
- FINDPATH, 57, 60
- FINDSHORTPATH, 57
- findstr, 176
- finite automaton, xvii, 7, 98, 164–191, 371.
 - See also* dfa, nfa
- first incompleteness theorem. *See* incompleteness theorem
- FixedBinAdLogic, 338
- FNP, 265–268
- formal language. *See* language
- Fortnow, Lance, 301
- FP, 230, 265
- from, 18
- function
 - mathematical, 53
 - partial, 323
 - total, 323
- function polynomial-time, 230
- function problem, 58

- Game of Life, 110
- Garey, Michael, 307
- gate, 282
- GEB. *See* Gödel, Escher, Bach
- general computational problem,
 - see* computational problem
- genetic string, 16, 17
- genetics, 5, 16, 223, 281, 307
- Glover, Denis, xiii
- GnTn, 181
- Gödel, Escher, Bach, xv

- Gödel, Kurt, 8, 10, 318, 333, 346, 349, 350. *See also* incompleteness theorem
- `godel.py`, 347
- Goldreich, Oded, 299
- Gowers, Timothy, 370
- grammar, xviii
- graph, 46
 - directed, 48
 - undirected, 46
 - weighted, 48
- graph isomorphism, 367
- GRAPHISOMORPHISM, 268, 309
- `grep`, 176
- GTHENONET, 154
- `GthenOneT`, 155

- HALFUHC, 308
- halt, 78, 322
 - for Python program, 126
 - for Turing machine, 76, 126
- halt state, 73
- HALTEX, 239
- `haltExTuring.py`, 240
- halting problem, xviii, 99, 126–128, 231, 241, 317, 323, 341
- halting states, 73
- HALTSBEFORE100, 136
- HALTSINEXPTIME,
see HALTEX
- HALTSINSOMEPOLY,
248
- HALTSONALL, 127
- HALTSONEMPTY, 127
- HALTSONSOME, 127
- HALTSONSTRING, 127
- `haltsViaPeano.py`, 345
- Hamilton cycle, 47, 204, 233
- Hamilton path, 47, 204
- Hamilton, William, 233
- HAMILTONCYCLE. *See* UHC
- hardware, 3, 25, 41, 71, 196
 - equivalence with software, 95
 - verification, 281
- Hartmanis, Juris, 228
- HASPATH, 60
- HASSHORTPATH, 60
- haystack analogy, 257
- heap sort, 217
- Hilbert’s 10th Problem, 6, 134
- Hilbert, David, 6, 45, 134, 318, 329, 332, 345, 346, 349
- hint, 251
- history of computational theory, xvii, 10, 126, 281, 309, 317–369
- Hobbes, Thomas, 317
- Hofstadter, Douglas, xv
- Hopper, Grace, 370

- IDLE, 16, 18, 103
- `ignoreInput.py`, 112
- imitation game, 324
 - 2014 movie of, 324
- `import`, 18
- incomplete, 338
- incompleteness theorem, 8, 10, 318, 333, 346, 349
- inconsistent, 338
- `incrementWithOverflow`, 83
- inference rule, 334
- infinite loop. *See* loop
- `infiniteLoop.py`, 23
- input
 - for Turing machine, 74
 - length of, 210
 - numerical value of, 210
- input/output tape, 89, 92, 94
- instance, 55
 - negative, 56
 - positive, 56
- `inString`, 16
- instruction. *See* CPU instruction
- instruction set, 93, 206, 220
- `int()`, 20
- integer polynomial equations. *See* Hilbert’s 10th Problem
- Intel, 94, 197, 214
- interactive proof, xviii
- intersection, 52
- intractable, 3–6, 245, 250, 354
- ISCOMPOSITE, 237
- ISEVEN, 131
- ISMEMBER, 63
- ISPRIME, 237, 310

- Java, xvi, 6, 8, 15, 16, 51, 52, 64, 68, 86, 104, 107, 176, 188, 191, 371
- Java virtual machine. *See* JVM
- JFLAP, xviii, 78–79, 88, 166, 169
- Jobs, Steve, xviii
- Johnson, David, 307
- `join()`
 - string method, 27, 45, 208
 - thread method, 144
- JVM, 86, 107

- Karp reduction. *See* polyreduction
Karp, Richard, 8, 10, 306, 307, 310, 353–369
Kayal, Neeraj, 237
keyword arguments, 147
Kleene star, 52, 176, 178, 179
Kleene, Stephen, 52, 327
KNAPSACK, 355, 359, 365–367
Knuth, Donald, xix, 11

Ladner, Richard, 272
lambda calculus, xviii, 327–329
language, 51, 354
 context-free, xvii, xviii
 empty, 51
 nonregular, 181–187
 recursive, xviii, 66
 recursively enumerable, xviii, 66
 regular, xviii, 181, 187
lastTtoA, 76
leaf, 49. *See also* computation tree
Leibniz, Gottfried, 332, 345
length of input. *See* input
level, 49
Levin, Leonid, 250, 297, 302, 353, 355, 356
liberal arts, xviii–xix
Lin, 222
linear programming, 367
linear speedup theorem, 218
LINEARPROGRAMMING, 310
Linux, 86
Linz, Peter, xvi
list comprehension, 212
literal, 284
 complementary, 362
little-*o*, 202
logarithmic basic function, 197, 198
logical system, 337
LogLin, 222
longerThan1K.py, 33
LONGESTPATH, 236
longestWord.py, 22
loop
 in Turing machine, 76
 infinite, 23, 33, 46, 65, 113, 126, 151, 322
Lovelace, Ada, 195
Lynch, Nancy, 272

m-configuration, 319
main function, 19, 24
many–one reduction. *See* polyreduction
MATCHINGCHARINDICES, 208
 matchingCharIndices.py, 209
mathematics, xix, 8, 11, 250, 307, 340, 357, 370, 371
 foundations of, 332
 incompleteness of, 346–349
 undecidability of, 345–346
max bisection, 245
MAXCUT, 236
maximal *N*-match, 334
maybeLoop.py, 33
MCOPIESOFc, 229
MCOPIESOFc.py, 211
mechanical proof, 336
memory, xviii, 24, 25, 92, 93, 97–98, 183
merge sort, 217
Mertens, Stephan, xvi, 135, 246, 307
Microsoft, 41, 103
MINCUT, 236, 310
Minsky, Marvin, 108
Moore, Cristopher, xvi, 135, 246, 307
moreCsThanGs, 80–82, 206
multi-core. *See* CPU core
multiple sequence alignment,
 see MULTSEQALIGN
multiplication, 213
 grade-school algorithm, 212
MULTIPLY, 58, 60, 212, 235
multiply.py, 212
multiplyAll.py, 19, 25, 31
multitasking, 143, 153
MULTSEQALIGN, 5, 12, 102

ndContainsNANA.py, 146
ndFindNANA.py, 149
ndFindNANADivConq.py, 151
neighbor, 48
new state function, 73
new symbol function, 73
newline character, 21
nfa, xviii, 167–171, 174, 180, 181
 definition of, 169
 strict, 169
Nisan, Noam, 95
NO, 64
node, 46
NODE COVER, 358
noMainFunction.py, 23
nonconstructive, 346
nondeterminism, xviii, 7, 143–163, 167, 170, 173, 264, 317, 320, 371
 compared with parallelism, 144

- nondeterministic, 144, 320, 371. *See also*
 - Turing machine, nfa, running time
 - computation, 151, 152
 - program, 24
 - Python program, 144, 153, 157
 - transducers, 158
- NonDetSolution, 145, 148
- NOT gate, 282
- NOTHASPETH, 61
- notYesOnSelf.py, 36, 37, 39
- NP, xviii, 7, 265, 266, 300, 301, 354, 356
 - definition of, 294
- NP-complete, xviii, 5, 8, 11, 159, 246, 250, 272, 281, 294–314, 353, 355–357
 - definitions of, 297–298
- NP-hard, xviii, 275, 298–300, 308, 367, 371
- NPComplete, 296
- NPoly, 7, 258–271, 309, 371
 - definition of, 258
- ntm, *see* Turing machine, nondeterministic
- NUMCHARSONSTRING, 129
- numerical function, 57
- NUMSTEPSONSTRING, 130

- objective function, 57
- operating system, 23–25, 41, 71, 95, 104, 107, 143, 144, 153, 158
- optimization problem, 57
- OR gate, 282
- oracle. *See also* Turing machine
- oracle function, 121
- oracle program, 121
- output
 - of nondeterministic computation, 151, 152
 - of Python program, 24
 - of Turing machine, 74

- P, xviii, 5, 229, 265, 266, 300, 301, 309, 354, 356
 - definition of, 294
- P versus NP, 5, 263, 294–296, 298, 301–302, 355, 356, 371
- PACKING, 257
- parallelism, 144
- PARTITION, 257, 359, 365–367
- path, 47
- pda. *See* pushdown automaton
- Peano arithmetic, 340, 341
 - consistency of, 344
 - incompleteness of, 346–349
- Petzold, Charles, 317

- philosophy, xix, 11, 324, 329
- physically realistic, 98, 144, 329, 371
- Poly, 7, 223, 228–249, 265
 - definition of, 228
- PolyCheck, 7, 223, 256–271, 371
- PolyCheck/NPoly, 262
- polyequivalent, 290
- polylogarithmic, 203, 246
- polynomial, 198
 - basic, 198
 - basic function, 197, 198
 - composition of, 203
- polynomial time. *See* polytime
- polynomial-time mapping reduction.
 - See* polyreduction
- polyreduction, xviii, 7, 11, 120, 272–293, 355
 - definition of, 273
- polytime, 4, 228, 246, 354, 371
 - definition of, 254
- Post correspondence problem, 134
- predicate, 57
- prerequisite, xvii, 8
- Priestley, Joseph, 164
- primality testing, 237, 367
- prime numbers, 51, 237
- private key, 216
- problem. *See* computational problem
- program, 7. *See also* computer program,
 - Python program
- programming, xvii, 8
- programming language, 8, 52, 91, 93, 104, 176, 188, 195
- proof by contradiction, 30–31
- proof system, 333, 335
- proof-writing, 9
- protein-folding, 301
- provable. *See* statement
- PROVABLEINPEANO, 343
- pseudo-code, 139, 210
- pseudo-polynomial time, 212, 311
- public key, 216
- pumping, 184
 - cutoff, 184
 - lemma, 185–187
- pushdown automaton, xvii, xviii, 173.
 - See also* the online supplement
- Python function, 16
- Python program, 10, 15–21, 23–26, 95, 97, 98, 221, 371
 - definition of, 24
 - halting of, 126

- impossible, 30–44
- language of, 51
- nondeterministic. *See* nondeterministic
- output, 24
- reductions via, 138
- running time of, 206–210, 220
- SISO, 18–21, 23, 26, 41, 45
- universal, 104–105
- Python programming language, 7, 8, 15
 - version of, 16, 23, 25, 97, 209
- `pythonSort.py`, 47

- Quad, 222
- quantum algorithm, 101
- quantum computer, 98, 220, 221, 238, 371
- quasipolynomial, 202, 246, 248, 309

- RAM, 92, 93
- random-access tape, 92
- randomness, xviii, 24, 223
- read-write head, 72
- recognize, 52, 65, 113, 134, 343
- `recognizeEvenLength.py`, 66
- recursive language. *See* language
- recursively enumerable. *See* language
- recursiveness, 327
- reduction, xviii, 7, 11, 116–142, 323, 371.
 - See also* polyreduction, Turing reduction
 - for easiness, 116–118
 - for hardness, 118–120
- reference computer system, 23, 24, 130, 206
- regex. *See* regular expression
- register, 93
- regular expression, xviii, 11, 175–181, 333, 339
 - primitive, 176
 - pure, 176–177
 - standard, 177–178
- regular language. *See* language
- reject
 - for nfa, 170
 - for Python program, 25, 78
 - for Turing machine, 74
 - implicit, 82, 172
- reject state, 73
- repeated N -match, 334
- repetition, 52
- `RestrictedBinAdLogic`, 337
- `returnsNumber.py`, 23
- reverse, 187

- `rf()`, 17
- Rice's theorem, xviii, 123, 133, 134, 139, 371
- Rivest, Ron, 216
- ROM, 93
- root, 49
- RSA, 216, 311
- rule 110 automaton, 109
- `Run Module`, 17
- running time. *See also* Turing machine
 - absolute, 197, 371
 - asymptotic, 197, 371
 - nondeterministic, 258, 266
- Russell, Bertrand, 143

- SAT, 8, 281–284, 286, 287, 289, 290, 296–298, 302, 353, 356, 361
 - definition of, 284
 - real-world inputs for, 310
- SAT-solver, 11
- satisfiability, 281, 282
- satisfy
 - Boolean formula, 283
 - circuit, 281
- Saxena, Nitin, 237
- scanned symbol, 73, 319
- Schocken, Shimon, 95
- search engine. *See* web search
- search problem, 57
- Searle, John, 15, 27
- self-reflection, 33
- Selman, Alan, 272
- Shamir, Adi, 216
- shell window, 16
- `shiftInteger`, 83
- shortest path. *See* SHORTESTPATH
- SHORTESTPATH, 4, 55, 60, 235, 236, 310
- SHORTESTPATHLENGTH, 58
- `simulateDfa.py`, 166
- `simulateTM.py`, 97
- simulation, 86
 - chain of, 86
 - costs, 217–221
- Sipser, Michael, xvi, 89, 346
- SISO, 18. *See also* Python program
- soft- O , 202
- software verification, 41, 281
- solution, 54
 - correct, 251
- solution set, 54
- solve, 62
- `sorted`, 45, 217

- sorting algorithms, 45–46, 227
 - complexity of, 217
- `SortWords`, 46, 62, 217, 228
- `split()`, 19
- splitting a clause. *See* clause
- start state, 73
- state diagram, 75, 165
 - abbreviated notation for, 78
 - for `nfas`, 168–169
- state set, 73
- statement, 333, 335
 - provable, 334, 336
- Stearns, Richard, 228
- `str()`, 20
- strict nfa. *See* nfa
- string, 50
- subexponential, 202
- `SUBSETSUM`, 257
- subtraction, 213
- superpolynomial, 5, 202, 228, 237
- symbol, 49, 72, 319
 - first kind, 320, 321
 - second kind, 320, 321
- `syntaxError.py`, 23

- tape, 72, 319
- `TASKASSIGNMENT`, 307, 310, 311
- term, 198. *See also* dominant
 - basic, 199
- terminate. *See* halt
- theory course, xv
- thread, 24, 144, 158, 258
 - child, 149
 - Python, 144
 - root, 149
- threading module, 144, 145
- threshold problem, 57
- `threshToOpt.py`, 69
- `throwsException.py`, 23
- time complexity, 205, 207, 210–217, 244
- time constructible, 222
- $\text{Time}(f(n))$, 222
- tractable, 3–5, 7, 245
- transducer, 77, 158
- transition function, 74, 75, 157, 165, 169
- transitivity
 - of dominant functions, 199
 - of polyequivalence, 290
 - of Turing reductions, 122
- traveling salesperson problem. *See* TSP
- tree, 49
 - rooted, 49

- `TRUEINPEANO`, 343
- truth assignment, 337
- truth problem, 339
- Tseytin transformation, 285–287, 290
- TSP, 232, 235, 310
- TSPD, 253
- TSPPATH, 235
- Turing equivalent, 98
- Turing machine, xvii, xviii, 7, 8, 10, 25, 26, 71–102, 143, 164, 173, 181, 195, 209, 302, 328, 354, 371
 - as defined by Turing, 317–323
 - clone, 154
 - definition of, 74
 - deterministic, 167
 - multi-tape, 86–91, 98, 217, 218, 327
 - nondeterministic, 152, 154–158, 164, 167, 354
 - oracle, 121
 - random-access, 92–95, 101
 - read-only, 191
 - running time of, 204–206
 - universal, 105–107
- Turing reduction, 11, 136, 139, 273, 276, 317
 - definition of, 120
 - polytime, 274, 299
- Turing test, 324, 327, 328
- Turing’s thesis, 329
- Turing, Alan, 7, 8, 10, 11, 30, 33, 39, 71, 73, 86, 106, 126, 317, 346, 349, 352, 370
- two-way infinite tape, 78, 88–89

- UHC, 236, 277–281, 296, 358, 364–365
 - definition of, 277
- unary, 244
- uncomputable problem, 3, 4, 6, 118, 123–140, 299, 370, 371
 - definition of, 62
- undecidable, xviii, 63
 - language, 64
 - logical system, 343, 345–346
 - problem, 62, 113, 134, 317, 323, 349
- UNDIRECTEDHAMILTONCYCLE, *see* UHC
- Unicode, 21, 51
- union, 52
- universal computation, xviii, 7, 103–115, 123, 135, 158, 219, 317, 370. *See also* Turing machine, Python program
 - real-world, 107–110
- `universal.py`, 105
- UNIX, 176

- unrecognizable, 158
- unsure, 253
- utils.py, 17
- utils.readfile(), 17

- verifier, 250–256, 371
 - definition of, 251
 - polytime, 254–256
- verifyFactor.py, 251
- verifyFactorPolytime.py, 254
- verifyTspD.py, 253
- vertex, 47

- waitForOnePosOrAllNeg(), 147
- WCBC, xv
- weaker computational model, 98, 181
- weather forecasting, 301
- web search, 4, 159
- weirdCrashOnSelf.py, 40
- weirdH.py, 241
- weirdYesOnString.py, 38, 241

- well-formed, 333, 335
- whitespace, 19, 21, 54
- wire, 282
- witness, 266
- Wolfram, Stephen, 330
- worst-case, 196, 205

- YES, 64
- yes.py, 33, 71
- YESONALL, 124
- YESONEMPTY, 124
- yesOnSelf.py, 35, 37, 39
- YESONSOME, 124
- YESONSTRING, 63, 124
- yesOnString.py, 33, 37–39
- yesViaComputesF.py, 133
- yesViaGAGA.py, 119
- yesViaHalts.py, 127

- ZeroDivisionError, 22