

# WHAT CAN BE COMPUTED?

A PRACTICAL GUIDE TO THE THEORY OF COMPUTATION



JOHN MACCORMICK

WHAT CAN BE COMPUTED?





# WHAT CAN BE COMPUTED?



A PRACTICAL GUIDE TO THE  
THEORY OF COMPUTATION



JOHN MACCORMICK

PRINCETON UNIVERSITY PRESS • PRINCETON AND OXFORD



Copyright © 2018 by Princeton University Press

Published by Princeton University Press,  
41 William Street, Princeton, New Jersey 08540

In the United Kingdom: Princeton University Press,  
6 Oxford Street, Woodstock, Oxfordshire OX20 1TR

[press.princeton.edu](http://press.princeton.edu)

Curved crease sculpture by Erik Demaine and Martin Demaine.  
Cover photo courtesy of the artists.

All Rights Reserved

ISBN 978-0-691-17066-4

Library of Congress Control Number 2018935138

British Library Cataloging-in-Publication Data is available

This book has been composed in Sabon and Avenir

Printed on acid-free paper ∞

Typeset by Nova Techset Pvt Ltd, Bangalore, India

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

# 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

<b>1 INTRODUCTION: WHAT CAN AND CANNOT BE COMPUTED?</b>	<b>1</b>
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

<b>2 WHAT IS A COMPUTER PROGRAM?</b>	<b>13</b>
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
<b>3 SOME IMPOSSIBLE PYTHON PROGRAMS</b>	<b>30</b>
3.1 Proof by contradiction	30
3.2 Programs that analyze other programs	31
Programs that analyze themselves	33
3.3 The program yesOnString.py	33
3.4 The program yesOnSelf.py	34
3.5 The program notYesOnSelf.py	36
3.6 yesOnString.py can't exist either	37
A compact proof that yesOnString.py 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
<b>4 WHAT IS A COMPUTATIONAL PROBLEM?</b>	<b>45</b>
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
<b>5 TURING MACHINES: THE SIMPLEST COMPUTERS</b>	<b>71</b>
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 moreCsThanGs machine	80
The countCs machine	81
Important lessons from the countCs 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 → random-access Turing machine	92
Random-access Turing machine → real computer	92
Modern computer → 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
<b>6 UNIVERSAL COMPUTER PROGRAMS: PROGRAMS THAT CAN DO ANYTHING</b>	<b>103</b>
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
<b>7 REDUCTIONS: HOW TO PROVE A PROBLEM IS HARD</b>	<b>116</b>
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 YESONSTRING	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 COMPUTES <sub>F</sub>	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
<b>8</b>	<b>NONDETERMINISM: MAGIC OR REALITY?</b>	<b>143</b>
8.1	Nondeterministic Python programs	144
8.2	Nondeterministic programs for nondecision 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
<b>9</b>	<b>FINITE AUTOMATA: COMPUTING WITH LIMITED RESOURCES</b>	<b>164</b>
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 GnTn	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
<b>Part II: COMPUTATIONAL COMPLEXITY THEORY</b>		<b>193</b>
<b>10</b>	<b>COMPLEXITY THEORY: WHEN EFFICIENCY DOES MATTER</b>	<b>195</b>
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
<b>11 Poly AND Expo: THE TWO MOST FUNDAMENTAL COMPLEXITY CLASSES</b>	<b>228</b>
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

<b>12</b>	<b>PolyCheck AND NPoly: HARD PROBLEMS THAT ARE EASY TO VERIFY</b>	<b>250</b>
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
<b>13</b>	<b>POLYNOMIAL-TIME MAPPING REDUCTIONS: PROVING X IS AS EASY AS Y</b>	<b>272</b>
13.1	Definition of polytime mapping reductions	272
	Polyreducing to nondecision 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
<b>14</b>	<b>NP-COMPLETENESS: MOST HARD PROBLEMS ARE EQUALLY HARD</b>	<b>294</b>
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
<b>Part III: ORIGINS AND APPLICATIONS</b>	<b>315</b>
<b>15 THE ORIGINAL TURING MACHINE</b>	<b>317</b>
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
<b>16 YOU CAN’T PROVE EVERYTHING THAT’S TRUE</b>	<b>332</b>
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
<b>17 KARP’S 21 PROBLEMS</b>	<b>353</b>
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

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
<b>18</b>	<b>CONCLUSION: WHAT WILL BE COMPUTED?</b>	<b>370</b>
18.1	The big ideas about what can be computed	370
	<i>Bibliography</i>	373
	<i>Index</i>	375

## ACKNOWLEDGMENTS



You should have been told  
Only in you was the gold

— Denis Glover, *Arawata Bill*

Several friends and colleagues offered substantial feedback on the manuscript; for this, I'm especially grateful to Chauncey Maher, Frank McSherry, Michael Skalak, Tim Wahls, and Barry Wittman.

Numerous students offered suggestions and found bugs. Among them are Leonard Bacon-Shone, Sid Batra, Jake Beley, Woo Byun, Ian Doyle, Steven Fitzpatrick, Abby Hsieh, James Midkiff, Lam Nguyen, Wode Ni, Khanh Pham, Yutong Shang, Mackenzie Stricklin, Peixin Sun, Jiacheng Wang, and Xiang Wei. Thank you all!

The anonymous reviewers offered many insightful comments that resulted in significant improvements. Vickie Kearn, Executive Editor at Princeton University Press, was instrumental in guiding the book from its earliest days to its final form.

I thank the London Mathematical Society for permission to quote Alan Turing's work in chapter 15 and Springer for permission to quote Richard Karp's work in chapter 17.

I am particularly grateful to my students in COMP314 (Theoretical Foundations of Computer Science) at Dickinson College, who suffered with surprising enthusiasm through several courses taught with fragmentary versions of the manuscript and often guided me towards the light. I also thank Dickinson College itself, including its staff, administrators, faculty, and students, for providing an environment that nurtures projects such as this one.

Without the support and joy of my family, this book would not have been possible. Kristine, Alastair, and Jillian: thank you! *Only in you was the gold*. The book is dedicated with great love to Alastair and Jillian.



## PREFACE FOR INSTRUCTORS



This preface is intended for instructors and other experienced computer science practitioners who are trying to understand the motivation for this book. The preface focuses especially on the similarities and differences compared to other treatments of the same material. If you are a student or general reader, you can skip the preface and start reading at chapter 1.

### THE INSPIRATION OF GEB

Ever since I read Douglas Hofstadter's classic *Gödel, Escher, Bach* (GEB) as a teenager, I've known that the field of computer science has, at its foundations, a collection of extraordinarily beautiful and profound ideas. And because of GEB's surprising accessibility, I've also known that these ideas can be presented to anyone interested in computer science. Decades later, when I consider the undergraduate computer science curriculum, I view it as an opportunity to energize and excite students with a similar collection of beautiful, profound ideas—taught in a way that is accessible to any computer science undergraduate.

This book—*What Can Be Computed?*, or WCBC for short—is an attempt to do just that. It would, of course, be outrageous hubris to claim some kind of equivalence with GEB; I have no intention of making such a claim. My only hope is to aspire towards the same type of excitement, around a somewhat similar collection of ideas, with a similar level of accessibility. If the book achieves its goals, students will gain insight into the power and limitations of computation: that some important and easily stated problems cannot be solved by computer; that computation is a universal and pervasive concept; that some problems can be solved efficiently in polynomial time, and others cannot; and that many important problems have not been proved intractable, but are nevertheless believed to have no efficient method of solution.

### WHICH "THEORY" COURSE ARE WE TALKING ABOUT?

In addition to trying to capture some GEB-like excitement, a more prosaic goal for the book is to serve as a textbook for an undergraduate course on the theory of computation. It's not uncommon to speak of “the” theory course in the computer science curriculum, but the undergraduate theory course has several different guises, including the following possibilities:

- **Automata theory.** This approach covers deterministic and nondeterministic finite automata, pushdown automata, languages, grammars, Turing

machines, and computability. This provides an excellent prelude to a compilers course, and also gives good coverage of computability. Complexity theory, including NP-completeness, is probably covered, but it is not a central theme of the course. Peter Linz's *Introduction to Formal Languages and Automata* is a leading example of the automata theory approach.

- **Complexity theory and algorithms.** This approach emphasizes complexity theory right from the start, and ties it in with computability as a subtopic. Taking this approach allows extra time to cover real algorithms in detail, producing something of a hybrid between traditional complexity theory courses and algorithms courses. *The Nature of Computation*, by Christopher Moore and Stephan Mertens, is perhaps too monumental to be considered as a strictly undergraduate textbook—but it is a wonderful example of this fusion between complexity theory and algorithms. It also includes a single magisterial chapter (chapter 7, “The Grand Unified Theory of Computation”) linking complexity theory with the three key approaches to computability.
- **Computability and complexity theory.** This approach sacrifices some of the knowledge needed for leaping directly into an advanced compilers course. Automata theory is still covered, but in less detail. Instead, more time is spent studying complexity classes. Advanced topics such as interactive proofs and randomness can also be covered. Michael Sipser's *Introduction to the Theory of Computation* is an excellent (perhaps even canonical) instance of the computability-and-complexity-theory approach.

WCBC falls squarely in the third category above: it covers the basics of computability and complexity with roughly equal emphasis. In my view, this approach has the best chance of capturing the GEB-style excitement discussed above.

## THE FEATURES THAT MIGHT MAKE THIS BOOK APPEALING

What is the difference between WCBC and highly regarded texts such as Sipser's? There are several features of WCBC that I hope instructors will find appealing:

1. **A practical approach.** It may seem oxymoronic for a “theory” course to take a “practical” approach, but there is no contradiction here. In fact, students can gain a visceral understanding of theoretical ideas by manipulating real computer programs that exploit those ideas. Therefore, this book uses Python programs wherever possible, both for presenting examples of ideas and for proving some important results. (The online materials also include Java versions of the programs.) Another aspect of practicality is that we mostly consider general computational problems (i.e., not just decision problems). Decision problems are an important special case, but computers are usually used to solve more general problems. By working primarily with general problems, students feel a more direct connection between the theory and practice of computation.
2. **Undergraduate focus.** Most CS-theory textbooks present a wealth of detail and technical material, making them suitable for graduate courses as well

as undergraduate courses. In contrast, WCBC focuses solely on presenting material for undergraduates. By eliminating advanced topics, we can spend more time discussing the details and nuances of the central ideas. Rigorous proofs are given for a strong majority of the results, but sometimes we veer towards providing the intuition behind important ideas rather than insisting on complete technical detail. Mathematical results are presented as “claims” rather than lemmas, theorems, propositions, and corollaries. This is a purely cosmetic feature but the intention is to maintain rigor while avoiding any unnecessary formality.

3. **Minimal prerequisites.** This book aims to make the “theory” course accessible to any student moving through the computer science major. The prerequisites are essentially high-school math and an introductory programming course. More specifically, the book requires elementary knowledge of exponential, logarithmic, and polynomial functions, but does not use calculus. It makes heavy use of proof by contradiction and big- $O$  notation, but both are explained from first principles. Proof by induction is not used at all.
4. **Historical perspective.** The book is in three parts. Part I covers computability theory and part II covers complexity theory. Part III of the book provides some historical perspective on the theory of computer science, by leading students through some of the original materials that have inspired the discipline. As discussed below, these can be interleaved with the technical material as the semester progresses, or taught as a special topic at the end of the semester.

## WHAT'S IN AND WHAT'S OUT

We've seen that the book focuses on undergraduates, in part by jettisoning advanced topics. So, what has been left in, and what has been left out? Obviously, the table of contents provides all the details, but an informal summary is given in figure 1. Note that pushdown automata and context-free languages are covered in an online appendix.

Some readers may be surprised that WCBC covers Turing machines (chapter 5) before finite automata (chapter 9). There is no need for alarm: finite automata are a special case of Turing machines, and students appear to have no trouble with the inversion of the conventional ordering. Because WCBC's practical approach to the theory of computation emphasizes computer programs, it makes sense for us first to study abstract models of real computer programs (i.e., Turing machines), and later to focus on the special case of finite automata.

## POSSIBLE COURSES BASED ON THIS BOOK

The primary application I have in mind for the book is a one-semester undergraduate course on the theory of computation. The content has been carefully selected so that the entire book can be covered in a single semester, even with an undergraduate class that has minimal prerequisites (programming experience and high-school math). Naturally, the chapters are arranged in my preferred

	<b>Computability</b>	<b>Complexity</b>
<b>Included</b>	Turing machines; undecidable problems including classics such as the halting problem; Rice's theorem; equivalence of Turing machines and real computers with infinite memory; universal computers; reductions for proving undecidability; nondeterminism; dfas, nfas, regular expressions, and regular languages. Pushdown automata and context-free languages are included in an online appendix.	complexity classes P, Exp, NP; problems in Exp but not P; NP-completeness, including an informal proof of the Cook–Levin theorem; P = NP; polynomial-time mapping reductions for proving NP-hardness.
<b>Excluded</b>	Unrestricted grammars; recursive function theory; lambda calculus; many details about recursively enumerable, recursive, and nonrecursive languages.	space complexity; hierarchy theorems; circuit complexity; randomness; interactive proofs.

**Figure 1:** Traditional theory topics that are included and excluded from this book.

teaching order. One of my goals is that the most advanced major topic, NP-completeness, is not eclipsed by being covered in the last one or two weeks of the semester. Hence, I try to finish NP-completeness with at least two weeks to spare and then cover the “historical perspective” topics of part III in a low-pressure way at the end of the course. This gives students time to complete homework assignments on NP-completeness and absorb the mysteries of P versus NP. I’ve also sometimes scheduled a creative final project in the last week or two. Examples include building an interesting Turing machine using JFLAP, implementing a challenging reduction in Python, and implementing simulation of nondeterministic automata in Python. Another approach is to interleave the historical and background topics of part III with the technical topics of parts I and II. Chapters 15 and 16 can be covered any time after chapter 6.

Chapter 9, on finite automata, could be omitted without causing much difficulty. Any or all of the chapters in part III can be skipped. Other sections that could be skipped or lightly skimmed include 4.5, 6.5, 7.6–7.7, 8.7–8.9, 9.4–9.6, 11.6–11.8, 12.6, 12.8, and 14.5.

## COMPUTER SCIENCE AS A LIBERAL ART

Steve Jobs, the visionary former CEO of Apple, was well known for his insistence that success in the high-tech industry requires appreciation of both technology and the liberal arts. When studying the theory of computation, we have an opportunity to blend these two apparently competing areas. The discipline of computer science, as a whole, has some aspects that are engineering focused and other aspects that are more reminiscent of the liberal arts. Indeed, the

computer scientist Donald Knuth has observed that of the seven ancient liberal arts (grammar, rhetoric, logic, arithmetic, geometry, music, astronomy), at least three play important roles in computer science. I believe that studying the theory of computation allows students to apply engineering skill while also contemplating connections between philosophy, mathematics, and the use of computers in our society.



## OVERVIEW





# 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 CGCTTA, 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 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?

Part I  
COMPUTABILITY THEORY





# 2



## WHAT IS A COMPUTER PROGRAM?

Programs are independent of their realization in machines; ...the same program could be realized by an electronic machine, a Cartesian mental substance, or a Hegelian world spirit.

— John Searle, *Minds, Brains, and Programs* (1980)

The main point of this book is to analyze what can be computed by computer programs. Therefore, it might seem important to agree on a clear definition of “computer program” before we start. Surprisingly, the precise definition doesn’t matter much. As we will see in chapter 5, it turns out that different types of computers, and different types of programs written in different programming languages, are all equivalent in terms of which problems they can solve.

Most of the time, however, it will be useful to focus our attention on a much more specific class of programs. This will make it easier to understand and prove concrete facts about programs, and to give meaningful examples. Throughout the book, therefore, we will focus on the class of *Python programs*. In principle, the ideas in the book apply equally well to other programming languages. But we need to choose one specific language for our examples, and in this text we use Python for that. The online materials also give Java versions of all the programs.

For our purposes, a *Python program* is a program written in the Python programming language and saved in a single file. A more formal definition of Python programs will be given in section 2.5. But at this point, we need to pause for a detour through the basics of running and editing Python programs.

### 2.1 SOME PYTHON PROGRAM BASICS

Typically, the name of a Python program file ends in “.py”. Our first example is `containsGAGA.py`, which you can download with the online materials for the book. The contents of this file are given in figure 2.1. The purpose of `containsGAGA.py` is to examine an input string, returning “yes” if the input contains “GAGA” as a substring, and “no” otherwise.

```

1 import utils; from utils import rf
2 def containsGAGA(inString):
3     if 'GAGA' in inString:
4         return 'yes'
5     else:
6         return 'no'

```

**Figure 2.1:** The Python program `containsGAGA.py`.

Like many of the examples in this book, `containsGAGA.py` is intended to operate on *genetic strings*—that is, strings containing the four letters C, G, A, T that represent the possible bases present in a fragment of DNA. Usually, these examples have no real biomedical significance. In the particular case of `containsGAGA.py`, for example, it’s unlikely there is any real application in genetics or medicine that requires the detection of the substring GAGA. Nevertheless, we will frequently use genetic strings in our examples, purely to add the flavor of practicality. Because real-life genetic strings can be billions of characters long, this will also remind us that we want our programs to work on inputs of any finite length, not just the tiny inputs that can easily be discussed on the pages of this book.

Let’s get back to some Python basics. The next few paragraphs give the minimal background and instructions needed to understand and run Python programs for this book. If you don’t already know some Python, it would be a good idea at this point to stop reading, and spend an hour or two working through an online tutorial. Excellent introductions are available at the official Python site, [www.python.org](http://www.python.org).

Here’s a quick explanation of the code in figure 2.1. Line 1 imports some utilities that we’ll use later. Line 2 defines a Python *function*, using the Python keyword `def`. The name of the function is `containsGAGA`, and the function receives a single parameter, called `inString` (an abbreviation of “input string”). The function uses an `if` statement to check whether the string `GAGA` occurs as a substring anywhere in `inString`, returning `yes` or `no` as appropriate. Searching for substrings is particularly easy in Python, using the `in` keyword (see line 3). Note the absence of curly braces, `{}`, which are used to surround blocks of code in other languages like Java and C++. In Python, blocks of code are identified purely by indentation. Also note the use of single quotes to surround strings, as in `'GAGA'`. In fact, Python lets you use single or double quotes (e.g., `"GAGA"`), but single quotes are more common.

To run this program, install a recent version of Python (preferably, version 3 or higher), which is freely available for all common operating systems at [www.python.org](http://www.python.org). The installation comes with a program called IDLE, which can be used to edit and run Python programs. Launch IDLE; this will open a window and display an introductory message, followed by a prompt where you can type input. The prompt looks like this:

>>>

Let’s call this window the *shell window*. From the shell window, use the `File|Open` menu to open the file `containsGAGA.py`. The file appears in a

new window that we'll call the *code window*. In the code window, hit F5 (or equivalently, choose Run Module from the Run menu). This does two things: (i) it restarts the Python interpreter, wiping out anything that may have been placed in memory by previously entered commands, and (ii) executes the code in the code window. In particular, the Python function `containsGAGA()` is now loaded into Python's table of available functions, and is ready for use. Try it out now:

```
>>> containsGAGA('CTGAGAC')
```

Once you have got this working (you should see the output `yes`), try it with a few different input strings. Convince yourself that the outputs (`yes` or `no`) are correct for every input. And note this important time-saving tip: use Ctrl+P or Alt+P (depending on your operating system) to cycle through your history of commands. You can edit previous commands too. This saves a great deal of typing.

## Editing and rerunning a Python program

Next, make a small change to `containsGAGA.py`. For example, change the outputs to be `yep` and `nope`. Do this by editing the code directly in the code window. Save the file, then reload it (hit F5). Go back to the shell window and verify that your changes worked.

## Running a Python program on input from a file

Suppose we want to find out if a genetic string that is thousands, millions, or billions of characters long contains the substring GAGA. It's obviously not going to be practical to paste our input string into the shell window. A better way to do this is to read the string directly from a file. The materials for this book include a file called `utils.py`, which contains many useful utilities, including a function that lets us read the contents of a file into a string. Every program provided with this book imports these utilities before doing anything else (see line 1 of `containsGAGA.py`). So the utilities will always be available, provided you have first loaded one of the book's program files.

In particular, there is a function called `utils.readFile()` that reads the entire contents of a file and returns it as a string. In the online book materials, there is a file called `geneticString.txt`. This file contains a genetic string that is 2093 characters long. We can store the file's contents in a string variable using

```
>>> longString = utils.readFile('geneticString.txt')
```

Because `utils.readFile()` is needed so often, we provide “`rf()`” as a convenient abbreviation for it—that's the purpose of the code fragment “`from utils import rf`” in line 1 of `containsGAGA.py`. So the previous command can be rewritten as

```
>>> longString = rf('geneticString.txt')
```

Try it now, and also use `print(longString)` to see the genetic string itself.

## 18 • 2 What is a computer program?

Next, we can easily find out whether this genetic string contains “GAGA”:

```
>>> containsGAGA(longString)
```

Of course, this could have been done more compactly in a single line:

```
>>> containsGAGA(rf('geneticString.txt'))
```

## Running more complex experiments on Python programs

Ideally, readers will have fun experimenting with the provided Python programs and with creating new Python programs too. As you become more experienced with Python, you may choose to switch to a more advanced development environment. IDLE is designed to be easy for beginners, but lacks some features preferred by professional software developers. So, feel free to adopt whatever technology you wish. But at a minimum, you need a way of easily experimenting with Python. Here’s one way to do that. From IDLE’s shell window, create a new file by choosing *File|New*. Save the file as `temp.py`, in the same folder as the online book materials. To make the utilities and `containsGAGA()` available, enter the following lines:

```
import utils
from utils import rf
from containsGAGA import *
```

These variants of the `import` command have slightly different effects. The `import` variant makes all functions in `utils.py` available, but their names must be prefixed by “`utils.`”, as in `utils.writefile()`. The `from` variant lets you use imported functions, without requiring any prefix. With `from`, you can import specific functions (e.g., `rf`) or use `*` to import all functions from a file.

Next, add some more commands to `temp.py`, so that you can conduct an experiment with `containsGAGA()`. For example, you could add the following lines:

```
shortString = 'CTGAGAC'
print('short string result:', containsGAGA(shortString))
longString = rf('geneticString.txt')
print('long string result:', containsGAGA(longString))
```

If you then save the file and hit F5, the two results of the `containsGAGA()` calls are printed in the shell window. By editing and adding to `temp.py`, you can expand and improve your experiment without having to retype commands in the shell window.

## 2.2 SISO PYTHON PROGRAMS

Notice that the `containsGAGA` function receives a string parameter as input, and returns a string (“yes” or “no”) as output. Any Python function that receives string parameters as input and returns a string value as output is called a *SISO function*. (SISO is an acronym of String In, String Out.) Usually, we will deal

```

1   def multiplyAll(inString):
2       # split on whitespace
3       numbers = inString.split()
4
5       # convert strings to integers
6       for i in range(len(numbers)):
7           numbers[i] = int(numbers[i])
8
9       # compute the product of the numbers array
10      product = 1
11      for num in numbers:
12          product = product * num
13
14      # convert the product to a string, and return it
15      productString = str(product)
16      return productString

```

**Figure 2.2:** The Python program `multiplyAll.py`.

with SISO functions that take exactly one string parameter, but we'll sometimes also consider SISO functions with multiple string parameters.

The formal definition of a “Python program” will be given in section 2.5. Until then, let's think of a Python program as just a single `.py` file containing one or more Python functions. The first function in the program is called the *main* function. Programs with a SISO main function are called *SISO Python programs*. In this book, the main function of every Python program will be a SISO function. In fact, because every Python program in the book is a SISO program, the word SISO is unnecessary, and we will usually omit it. So from now on, we use “Python program” and “SISO Python program” interchangeably.

This SISO requirement may seem strange or unnecessarily restrictive. After all, many computer programs deal with numbers, not strings. The reason for restricting to SISO programs is that it simplifies our analysis of what can be computed. But in fact, the restriction is only a superficial one. The next example shows how to convert between numbers and strings in Python. And much more complicated objects (e.g., graphs) can also be described by strings—look ahead to figure 4.3 (page 48) for an example of this.

Let's now examine how SISO programs can compute numerical results. Figure 2.2 shows the listing for `multiplyAll.py`, a SISO program that computes the product of a list of numbers. (Note that this listing omits the first line, “`import utils; from utils import rf`”. From this point on, listings will usually omit `import` statements and certain other kinds of “plumbing” code.) Here's an explanation of `multiplyAll.py`, assuming we provide “5 8 3” as the input parameter:

- Line 1: The parameter `inString` is “5 8 3”.
- Line 3: `inString` gets split into an array of strings, using the `split()` method for Python strings. By default, `split()` splits on whitespace, which means that any combinations of space, newline, and tab characters are used

```

1  from containsGAGA import *
2  def containsGAGAandCACAandTATA(inString):
3      if containsGAGA(inString)=='yes' and \
4          containsCACA(inString) and \
5              containsTATA(inString):
6          return 'yes'
7      else:
8          return 'no'
9
10 def containsCACA(searchString):
11     return containsSubstring(searchString, 'CACA')
12
13 def containsTATA(searchString):
14     return containsSubstring(searchString, 'TATA')
15
16 def containsSubstring(searchString, subString):
17     if subString in searchString:
18         return True
19     else:
20         return False

```

**Figure 2.3:** The Python program `containsGAGAandCACAandTATA.py`.

as separators. As a result, after line 3 has been executed, `numbers` is a three-element array containing the three strings “5”, “8”, and “3”.

- Lines 6–7: Each element of the `numbers` array is converted from a string to an integer, using the built-in Python function `int()`.
- Lines 10–12: Compute the product of all values in the `numbers` array.
- Line 15: Convert the integer product into a string, using the built-in Python function `str()`.
- Line 16: Return the answer as a string of digits, “120”.

As you can see, it’s trivial to convert between strings and integers using the built-in functions `int()` and `str()`. So this SISO function works with strings on the surface, but deals with numbers under the covers.

## Programs that call other functions and programs

One of the most powerful ideas of modern programming languages is that functions or methods can call other functions or methods, which in turn call other functions or methods, and so on. Figure 2.3 illustrates these possibilities with a slightly silly example: the Python program `containsGAGAandCACA-andTATA.py`. This program imports `containsGAGA()` from another file, and provides additional functions `containsCACA()`, `containsTATA()`, and `containsSubstring()` below the main function. These last three functions return Boolean values, demonstrating that non-main functions do not have to be SISO.

Figure 2.3 also introduces one new feature of Python: excessively long lines can be continued onto a new line using the “\” character. This can be seen at

line 5: this line and the two previous ones constitute a single “virtual” line, as far as the Python interpreter is concerned.

## 2.3 ASCII CHARACTERS AND MULTILINE STRINGS

In this book, we will often use a set of characters known as *ASCII*, which is an acronym for American Standard Code for Information Interchange. ASCII is a standard set of 128 characters often used in storing and transmitting text files. For example, in some word-processing programs, when you choose the “Save As” menu option and select the .txt file format, your document is saved as a sequence of ASCII characters. The ASCII character set includes all the letters, digits, and punctuation characters that appear on English keyboards (including !@#\$%^&\*() -\ / " ' \_~| + , . <>?=), as well as the space character, the tab character, and two variants of the newline character. These last-mentioned characters (space, tab, newlines) are collectively known as *whitespace*. Sometimes we’ll use “ $\downarrow$ ” to represent a newline character within a string. For example, the string “a $\downarrow$ b $\downarrow$ c” represents the letters a, b, and c printed on separate lines. Within a Python program itself, a newline character can be represented by the special character combination “\n”. Look ahead to countLines.py (figure 2.4, page 22) for an example of this. One other technicality needs to be mentioned: strictly speaking, Python uses a character set called *Unicode*, which includes ASCII as a subset but incorporates many thousands of other characters too. This book, however, avoids Unicode. Our programs and strings will be restricted to the 128 ASCII characters, with some minor exceptions that will be clearly noted.

The fact that ASCII includes space and newline characters is extremely useful. It means that we can think of large, multiline blocks of text as being single strings. Thus, the following excerpt from T. S. Eliot’s poem *The Waste Land* is a single ASCII string:

```
Here is no water but only rock
Rock and no water and the sandy road
The road winding above among the mountains
Which are mountains of rock without water
```

For us, the main importance of strings is that they can be used as the inputs and outputs of computer programs. For example, the excerpt above could be used as the input to a computer program that counts the number of lines (output: 4), or finds the longest word (output: mountains). See figure 2.4 for implementations of these two examples. Test them out now using the file wasteland.txt provided with the online book materials:

```
>>> countLines(rf('wasteland.txt'))
>>> longestWord(rf('wasteland.txt'))
```

The programs in figure 2.4 demonstrate another feature of the Python examples used in this book. Generally speaking, the code presented here does *not* strive for efficiency, and does *not* attempt to use typical, elegant Python idioms. For example, countLines.py is rather inefficient, because it creates a new array containing all the lines of the input, rather than simply counting

```

1  def countLines(inString):
2      # split on newlines
3      lines = inString.split('\n')
4      # return the number of lines, as a string
5      return str(len(lines))

```

```

1  def longestWord(inString):
2      words = inString.split()
3      longest = ''
4      lengthOfLongest = 0
5      for word in words:
6          if len(word) > lengthOfLongest:
7              longest = word
8              lengthOfLongest = len(word)
9      return longest

```

**Figure 2.4:** The Python programs `countLines.py` and `longestWord.py`.

the newline characters in the input. As for `longestWord.py`, experienced Python programmers will frown at the long-winded use of a `for` loop with two local variables. In fact, this program can be written in only one or two lines of elegant Python. But the “elegant” approach requires some more advanced Python constructs, and we will avoid those advanced features when it makes sense to do so. At all times, the emphasis will be on making the programs easy to understand, especially for readers with little or no Python background. Elegance and efficiency will often be sacrificed to achieve this goal.

## 2.4 SOME PROBLEMATIC PROGRAMS

Figure 2.5 presents five Python programs that are problematic in different ways—either they don’t meet our definition of SISO Python programs, or they exhibit some kind of behavior that we want to avoid. It’s important to understand what makes each of these programs problematic. So, run each of the programs yourself, in IDLE. You should observe the following behaviors:

- `syntaxError.py`: This program is not syntactically correct Python because the parentheses are not balanced at line 2. When you hit F5 to evaluate the code, you get a message similar to “`SyntaxError: unexpected EOF while parsing.`”
- `noMainFunction.py`: This program has no main function. Although it is syntactically correct Python, it does not meet this book’s definition of a Python program.
- `throwsException.py`: When you run this program on any input, the result is a Python *exception*, because the program tries to divide by zero. The precise output in this case is “`ZeroDivisionError: division by zero.`” Python programs can throw many different kinds of exceptions. Real-world Python software often uses exceptions in a beneficial way. But in this book,

```

1 def syntaxError(inString):
2     return 2 * (3 + 5)

3 import utils; from utils import rf
4 x = 5
5 y = 7
6 z = x + y

7 def throwsException(inString):
8     return 5 / 0

9 def infiniteLoop(inString):
10    x = 1
11    while x > 0:
12        x = x + 1
13    return str(x)

14 def returnsNumber(inString):
15    return 32

```

**Figure 2.5:** Five problematic Python programs: `syntaxError.py`, `noMainFunction.py`, `throwsException.py`, `infiniteLoop.py`, and `returnsNumber.py`.

throwing an exception is considered bad behavior and makes the output of the program undefined.

- `infiniteLoop.py`: The main function enters an infinite loop and never returns, so its output is undefined.
- `returnsNumber.py`: Our definition of Python programs requires a SISO main function. Here, the main function returns an integer. This would be common in real-world Python software, but in this book the program output is undefined unless the main function returns a string.

## 2.5 FORMAL DEFINITION OF PYTHON PROGRAM

It's now time to give a careful definition of SISO Python programs and their outputs. These definitions can be skipped on first reading; they are targeted at readers who are interested in making the “practical” Python program approach as rigorous as possible. If that goal doesn't interest you, jump ahead to section 2.6 on page 25.

Our definitions will make use of a *reference computer system*, denoted  $C$  for Computer. The reference computer  $C$  will be discussed in more detail below. For now, you can imagine that  $C$  is a complete description of a real computer, including the computer architecture, operating system, and file-system contents. It might be easiest to imagine that  $C$  is your own laptop or desktop computer, complete with the particular version of Python that you have chosen to install.

In general, it's possible for real computer programs to produce different outputs on different runs (e.g., by using a source of randomness, or accessing some frequently changing data on the Internet, or using multiple threads that get interleaved arbitrarily by the operating system). Programs that can produce different outputs are called *nondeterministic* and will be studied in chapter 8. In contrast, we call a program *deterministic* if its execution path is uniquely determined by the state of  $C$  when the program begins executing. Programs in this book are assumed to be deterministic unless stated otherwise.

With those technical details taken care of, we can launch into our formal definitions:

**Definition of a Python program.** With respect to a reference computer system  $C$ , a *Python program* is a string of ASCII characters  $P$ , such that

- $P$  is syntactically correct Python;
- $P$  defines at least one Python function (in which case the first defined function is the *main function*).

We also need to define the *output* of a Python program. Our initial definition assumes the main function has a single parameter; we will later describe how to relax that requirement.

**Definition of  $P(I)$ , the output of a Python program.** Let  $P$  be a Python program with respect to a reference computer system  $C$ . Suppose  $P$  has main function  $M$ , and let  $I$  be a string of ASCII characters that will be used as input to  $P$ . The *output of  $P$  on input  $I$* , written  $P(I)$ , is produced by using computer  $C$  to execute  $M$  with input  $I$ , then defining  $P(I)$  as follows:

- If  $M$  returns an ASCII string  $S$ , then  $P(I) = S$ .
- If  $M$  returns a Python object that is not an ASCII string, then  $P(I)$  is undefined.
- If  $M$  throws an exception on input  $I$ , then  $P(I)$  is undefined.
- If  $M$  doesn't terminate on input  $I$ , then  $P(I)$  is undefined.

In plain English,  $P(I)$  is the string output by  $P$ 's main function on input  $I$ , whenever this makes sense, and undefined otherwise. The detailed definition above gives the precise meaning of “whenever this makes sense.”

Note that the definition can easily be generalized for main functions that receive multiple string parameters. We replace  $I$  with multiple strings  $I_1, I_2, \dots$ , and obtain a corresponding definition for  $P(I_1, I_2, \dots)$ .

The above definitions of Python programs and their outputs work perfectly well in practice, but from a mathematical point of view they might appear to have some technical problems. What exactly do we mean by “syntactically correct Python”? Does the definition depend on which computer and operating system are used? Do different versions of Python produce different results? How do we define the output if the computer crashes during execution? What if someone has edited the Python library files, so that a statement such as “`import sys`” does not import the expected functionality from the `sys` module? What if the Python program runs out of memory?

Ultimately, the correct way to answer all these questions is to define a mathematical model of a completely abstract computer with fixed, known behavior. In chapter 5, we will meet the *Turing machine*, which is precisely the kind of mathematical model needed. We will see that it's always possible to think of a Python program as a convenient representation of a Turing machine. So any lack of rigor in real Python programs can be eliminated by considering the underlying Turing machine instead.

But we would like to use Python programs as our computational model whenever possible. What kind of computer systems should we imagine our Python programs are running on? The short answer is, choose whatever hardware and software configuration you want, and then add as much memory as needed to execute the program. *This* is the reference computer system *C* to which the above definitions apply. In other words, whenever we prove a theoretical result about a Python program, feel free to consider that we are really proving the result for programs running on your computer, with your operating system, your files, and your version of Python. When you examine the proof, you'll find that none of our assumptions depend on the hardware architecture, Python version, or any other specific details of the computing configuration. It's true that there may be minor differences in output between different architectures, operating systems, and Python versions. But our results will always hold, despite these differences.

For example, the precise definition of “syntactically correct Python” is determined by the actual computer system *C* you have chosen to use. If `prog.py` runs on your computer with no syntax errors, then it's syntactically correct with respect to your system. In some cases, `prog.py` might be syntactically correct on your machine but incorrect on mine. However, any theorem that we prove about Python programs in general will be true on your computer and my computer, and for any other reasonable computer system *C*.

The problem of running out of memory is more fundamental, and we will return to it in chapter 5 (see page 97). For now, just imagine that we can add as much memory as needed to the computer system *C*.

## 2.6 DECISION PROGRAMS AND EQUIVALENT PROGRAMS

This section gives two important technical definitions about Python programs. We will often need to analyze an important class of programs with particularly simple outputs, known as decision programs.

**Definition of a decision program.** A *decision program* is a Python program that always returns one of the two strings “yes” or “no”, whenever the output is defined.

For example, `containsGAGA.py` is a decision program, but `multiplyAll.py`, `countLines.py`, and `longestWord.py` are not decision programs.

Decision programs are central to computational theory, so we have special terminology for the outputs. If  $P(I) = \text{yes}$ , then  $P$  *accepts*  $I$ . If  $P(I) = \text{no}$ , then  $P$  *rejects*  $I$ . For example, `containsGAGA.py` accepts “CTGAGAA” but rejects “CCGA”.

We also need a notion of equivalence between Python programs. Two programs are *equivalent* if for any given input, both programs produce the same output. Formally,

**Definition of equivalence of Python programs.** Python programs  $P_1$  and  $P_2$  are *equivalent* if  $P_1(I) = P_2(I)$  for all strings  $I$ . For multiparameter programs, we of course require  $P_1(I_1, I_2, \dots) = P_2(I_1, I_2, \dots)$ .

Note that the definition of equivalence includes undefined outputs: if  $P_1$  and  $P_2$  are equivalent, then  $P_1(I)$  is undefined if and only if  $P_2(I)$  is undefined.

## 2.7 REAL-WORLD PROGRAMS VERSUS SISO PYTHON PROGRAMS

Let us now return to the main topic of this chapter: What is a computer program? The answer depends on your point of view. In real life, computer programs do many wonderful things. They display graphics on the screen; they receive input from keyboards, mice, and touchscreens; they send information to, and receive information from, other computers on the Internet. However, the particular class of programs analyzed in this book is much more restricted. Our programs will always receive as input one or more strings of ASCII characters. And they will always produce as output a single string of ASCII characters. These are the SISO Python programs defined in the previous section.

It may seem surprising, but we don't lose much generality in restricting to SISO programs. The reason is that ASCII strings can easily be converted to and from binary strings. (Look ahead to page 50 if you're curious about this.) And all the input and output methods mentioned above (including keyboards, disks, touchscreens, monitors, wireless networks, and wired networks) provide input and output as binary strings. So it's always possible to think of any computer program as receiving exclusively ASCII string input, and providing exclusively ASCII string output. Hence, in the remainder of this book, the phrase "computer program" will refer only to SISO programs—but keep in mind that all our results will apply equally well to programs with more general types of input and output.

Even if we restrict to SISO programs, however, there are at least three different—but equivalent—answers to the question, what is a computer program?:

1. **The “Turing machine” definition.** As already discussed on page 25, Turing machines are the abstract models of computation that we will meet in chapter 5. Because they are completely abstract, Turing machines are well suited to rigorous mathematical proofs.
2. **The “Python program” definition.** On page 24, we gave a reasonably formal definition of SISO Python programs, but it wasn't completely rigorous from a mathematical point of view. Python programs provide a practical approach to discussing and analyzing computation, so we will use Python programs as the main computational model in this book. The equivalence of Python programs and Turing machines is discussed in chapter 5.
3. **The “anything reasonable” definition.** It turns out that any other reasonable definition of “computer program” is equivalent to the previous two! That is, any piece of SISO computer code written in any programming

language, and run on any existing computer (with as much memory as necessary), is equivalent to some Python program and some Turing machine.

When reading this book, it will help to keep all three of the above definitions of *computer program* in mind. When proving theorems, we will use Python programs or Turing machines, choosing whichever definition leads to the easiest proof. But we should acknowledge that our definition of Python programs doesn't quite achieve the level of rigor required for formal mathematical reasoning. If there is ever any doubt about the rigor of a mathematical proof that uses Python programs, we should think of the Python program as simply a convenient representation of an equivalent Turing machine.

One of the beautiful properties of computational theory is that it applies to all computer programs, on all computers. Indeed, as John Searle observed in the quotation opening this chapter, we can imagine programs being instantiated in biological entities, or even spiritual ones—but for our purposes in this book, it will be easiest to restrict ourselves to more standard computer hardware.

## EXERCISES

**2.1** Spend a few hours working through the basics of Python using an online tutorial. At a minimum, make sure you understand

- `if` statements, `for` loops, and `while` loops;
- basic operations on the three main data structures in Python: lists, tuples, and dictionaries;
- using the string `join()` method to convert a list of strings into a single string with a given separator between the original items in the list;
- array slicing, for example, `s[3 : 7]` is a list of `s[3]` to `s[6]` inclusive;
- negative indices, for example, `s[-4]` is `s[len(s) - 4]`.

**2.2** Implement each of the following programs as a SISO Python program. In each case you may assume the input is valid (i.e., consists of a correctly formatted list of integers).

- (a) Write a program that takes as input a list of integers separated by whitespace. The output is a string representing the sum of every second integer in the list. For example, if the input is “58 41 78 3 25 9” then the output is “53”, because  $41 + 3 + 9 = 53$ .
- (b) Write a program, similar to the program in (a), but adding up every *third* element of the input instead of every second element.
- (c) Write a decision program that accepts a list of integers if the sum of every third element is greater than the sum of every second element, and rejects otherwise. Your program must import and use the programs from (a) and (b).

**2.3** Write a SISO Python program that takes two parameters, and returns a string representing the number of times the character G occurs in each parameter, separated by a newline character. For example, if the two input parameters are (“CAGGT”, “GTGTGTGT”), the output should be “2 4”.

**2.4** Which of the following strings  $P$  is a Python program, according to the formal definition given in the section 2.5? (You may base your definition on any reasonable reference computer  $C$ .)

- (a)  $P = \text{"def f(x): return x"}$
- (b)  $P = \text{"def f(x): x = 'am I a Python program?'”}$
- (c)  $P = \text{"x = str(7+2*5)"}$
- (d)  $P = \text{"def f(x,y,z): return y"}$
- (e)  $P = \text{"def f(x): return y"}$
- (f)  $P = \text{"def f(x): ret x"}$

**2.5** For each of the following Python programs  $P$  and input strings  $I$ , give the output  $P(I)$ , using the formal definition of  $P(I)$  given in section 2.5, and employing any reasonable reference computer  $C$ :

- (a)  $P = \text{"def f(x): return x[-2]"}$ ,  
 $I = \text{"abcdefg"}$
- (b)  $P = \text{"def f(x): return x+5"}$ ,  
 $I = \text{"43"}$
- (c)  $P = \text{"def f(x): while True: x=7 return x"}$   
 (note that “ $x=7$ ” is indented here but “ $return$ ” is not),  
 $I = \text{"CAGGT"}$
- (d)  $P = \text{"def f(x): return x[3:7]"}$ ,  
 $I = \text{"abcdefghijklm"}$
- (e)  $P = \text{"def f(x): return x[3:7]"}$ ,  
 $I = \text{"ab"}$
- (f)  $P = \text{"def f(x): return str(len(x+x+'x'))"}$ ,  
 $I = \text{"GAGAT"}$
- (g)  $P = \text{"def f(x): return str(len(x))"}$ ,  
 $I = P$
- (h)  $P = \text{"def f(x): return str(1/int(x))"}$ ,  
 $I = \text{"0"}$
- (i)  $P = \text{"def f(x,y): return x[-1]+y[-1]"}$ ,  
 $I_1 = \text{"abc"}, I_2 = \text{"xyz"}$  (give  $P(I_1, I_2)$  for this question)

**2.6** Which pairs of the following Python programs are equivalent?

- $P = \text{"def f(x): return str(len(x)+1)"}$
- $Q = \text{"def f(x): return str(len(x+'a'))"}$
- $R = \text{"def f(x): L=1 while L!=len(x): L=L+1 return str(L+1)"}$  (note that all lines except the first are indented here)

**2.7** Consider the following Python program:

```

def hmmm(inString):
    m = int(inString)
    if m>0 and m%10==0:
        return 'yes'
    else:
        return 'no'

```

```

1  def oooops (inString) :
2      try:
3          val = int(inString)
4      except ValueError:
5          val = len(inString)
6      s = 'A'
7      i = 0
8      while i != val:
9          s += inString[2]
10         i += 1
11     return s

```

**Figure 2.6:** The Python program `oooops.py`. See exercise 2.9.

- (a) Describe the set of strings accepted by this program.
- (b) Describe the set of strings rejected by this program.
- (c) Describe the set of strings that are neither accepted nor rejected by this program. (Hint: It is a nonempty set.)
- (d) Is the program a decision program?

**2.8** Write a Python decision program that is as short as possible, as measured by the number of characters in the source code file. Your program must satisfy the formal definition of a Python program (page 24).

**2.9** Read a tutorial about using `try...except` blocks with Python exceptions. Then consider the program `oooops.py` in figure 2.6, and answer the following questions:

- (a) What is `oooops("abc")`?
- (b) What is `oooops("abcdefghijkl")`?
- (c) What is `oooops("a")`?
- (d) What is `oooops("008")`?
- (e) What is `oooops("8")`?
- (f) What is `oooops("-11")`?
- (g) Describe the set of all strings  $I$  for which `oooops(I)` is undefined, according to the formal definition of Python program outputs on page 24.

# 3



## SOME IMPOSSIBLE PYTHON PROGRAMS

There are certain things that such a machine cannot do.

— Alan Turing, *Computing Machinery and Intelligence* (1950)

From the overview in chapter 1, we already know that there are certain problems that can't be solved by computer programs. The existence of these uncomputable problems is, arguably, the most profound and important result in theoretical computer science. In this chapter, we plunge directly into the depths of uncomputability. Our goal will be to give a rigorous proof of a result that was stated in chapter 1: it is impossible to write a computer program that examines other computer programs and successfully finds all the bugs in those programs. But we will approach that goal in gradual steps. Our first step on that path is to understand a technique called “proof by contradiction.”

### 3.1 PROOF BY CONTRADICTION

This book will make heavy use of a mathematical technique called *proof by contradiction*. Although mathematicians like to lay claim to this technique, it's actually something that people use all the time in everyday life, often without even thinking about it. For example, suppose you see some milk in your fridge. You remember that you bought the milk a few days ago at a local grocery store, but you'd like to remember exactly which day it was purchased. An idea pops into your head, and you ask yourself, did I buy the milk on Monday? A split second later, you remember that you were out of town all day Monday, so you couldn't possibly have bought the milk at a local store that day.

This is a typical human thought process, but if we examine it more closely, it turns out to be a proof by contradiction. It starts with a statement  $S$  that we would like to prove is false (“I bought the milk on Monday”). We then *assume* that  $S$  is true. Based on this assumption, we derive some other statement  $C$  (for Consequence) that follows from  $S$ . In this case, the consequence  $C$  is “I was in town on Monday.” We then observe that  $C$  contradicts some other statement  $T$  that is known to be true (“I was out of town on Monday”). Therefore the

assumption  $S$  that produced  $C$  must be wrong, and we conclude that  $S$  is false. In words, “If I bought the milk on Monday, I was in town Monday; but I was out of town on Monday, so I didn’t buy the milk on Monday.” More abstractly, “ $S$  implies  $C$ , but  $C$  is false, so  $S$  is false.”

This argument was presented in great detail, because it’s essential to understand that proof by contradiction is a natural, obvious technique that we use all the time. Throughout the book, whenever you see a proof by contradiction that seems mysterious, think back to this simple example. That is, instead of asking, can the program  $P$  exist?, ask yourself, did I buy the milk on Monday?

One slight wrinkle is that proof by contradiction is always used to prove that a statement is *false*. If you want to prove that something is *true*, you need to first assume its opposite. That’s why mathematical proofs often begin with arguments like the following:

We wish to prove that all zurgles are yellow. Suppose not, and argue for a contradiction. Thus, we may assume the existence of a non-yellow zurgle  $Z$ . [And now continue, showing that  $Z$ ’s existence contradicts a known fact.]

Here is a real example, which relies on the well-known fact that the square of an integer cannot be negative.

**Claim 3.1.** The equation  $x^2 + 9 = 0$  has no integer solutions.

*Proof of the claim.* Assume not, and argue for a contradiction. Thus, we may assume the existence of an integer  $x$  such that  $x^2 + 9 = 0$ . Rearranging this equation, we see that  $x^2 = -9$ . Thus,  $x$  is an integer whose square is negative. This contradicts the fact that the square of an integer cannot be negative. Hence, our initial assumption that  $x$  exists must be false.  $\square$

## 3.2 PROGRAMS THAT ANALYZE OTHER PROGRAMS

According to the definition on page 24, Python programs are just ASCII strings. And program inputs are ASCII strings too. This raises the interesting possibility that the input to a Python program could be another Python program.

A simple example of this is provided by the `countLines.py` program, which we have seen already but is repeated here in figure 3.1. This program counts the number of lines in its input. For example, on input “zyx↵3 2 1↵cbabc”, `countLines.py` outputs “3”. On the other hand, what happens if we execute the following shell command?

```
>>> countLines(rf('multiplyAll.py'))
```

(Here, `multiplyAll.py` is the program shown in figure 2.2, page 19. And recall that `rf` is an abbreviation for `utils.readFile`.) The program `multiplyAll.py` is a 19-line string, so the output of the above command is “19”.

To anyone familiar with computers, this trivial experiment is completely unsurprising. Programs are stored as files, and of course programs can read and

```

1 def countLines(inString):
2     # split on newlines
3     lines = inString.split('\n')
4     # return the number of lines, as a string
5     return str(len(lines))

```

Figure 3.1: The Python program countLines.py.

```

1 import utils; from utils import rf
2 def containsGAGA(inString):
3     if 'GAGA' in inString:
4         return 'yes'
5     else:
6         return 'no'

```

```

def yes(inString):
1 return 'yes'

```

```

def longerThan1K(inString):
1 if len(inString) > 1000:
2     return 'yes'
3 else:
4     return 'no'

```

```

def maybeLoop(inString):
1 if not 'secret sauce' in inString:
2     # enter an infinite loop
3     i = 0
4     while i>=0:
5         i = i + 1
6 else:
7     # output "yes" if input length is even and "no" otherwise
8     if len(inString) % 2 == 0:
9         return 'yes'
10    else:
11        return 'no'
12

```

Figure 3.2: Four examples of decision programs.

analyze other files. So we should certainly expect that one program (countLines.py) can analyze another program (multiplyAll.py)—in this case, by counting the number of lines in the input program. On the other hand, it's also possible to take a more profound, philosophical view of this experiment: in the universe as we know it, the ability of one system to analyze other similar systems appears to be rather rare. In fact, apart from computer programs, the only obvious

examples are the brains of animals here on Earth. So, we might suspect that our simple `countLines.py` program represents the tip of a deep philosophical iceberg.

## Programs that analyze themselves

And we can quickly push this capacity for analysis to an even deeper level. What happens when we execute the following command?

```
>>> countLines(rf('countLines.py'))
```

This asks `countLines.py` to analyze itself, and report how many lines of code it has. Try this experiment now; you should get the output “7”. (The listing of `countLines.py` in figure 3.1 shows only 5 lines, but there are actually two additional lines not shown in the listing.)

Again, the success of this experiment is unsurprising to anyone who has worked with computers. But there is again a hint of something remarkable here: this time, we have an example of *self-reflection*—the capability of computer programs, as with the brains of animals, to perform certain types of analysis on *themselves*.

However, as Alan Turing discovered in 1936, the great power of self-reflection brings with it certain inevitable limitations: the very fact that computer programs can analyze themselves enables us to prove that there are certain things computer programs cannot do at all. The next section explains the details behind this peculiar trade-off.

## 3.3 THE PROGRAM `yesOnString.py`

The rest of this chapter will focus on decision programs, which output one of only two possible strings “yes” or “no”. (See page 25 for a formal definition.) Four examples of decision programs are shown in figure 3.2. We’ve seen `containsGAGA.py` before, and the other three are also easy to understand:

- `yes.py` ignores its input, and always returns “yes”
- `longerThan1K.py` returns “yes” if its input is longer than 1000 characters, and “no” otherwise
- `maybeLoop.py` enters an infinite loop unless its input contains the substring “secret sauce”. For inputs that do contain “secret sauce”, the program returns “yes” on inputs of even length, and “no” otherwise.

Before moving on, we need an excellent understanding of what happens when decision programs analyze other programs and/or themselves. Figure 3.3 lists 16 different examples of decision programs in action. It may seem tedious, but it’s essential that you work through all 16 examples before reading on. Cover the output column with your hand, and verify that you can reproduce it effortlessly. It would also be a good idea to execute these commands on your own computer, and play around with some similar examples.

We’re now in a position to think about some much more interesting decision programs. The first one we’ll investigate is called `yesOnString.py`. This

Shell command (with <code>rf = utils.readFile</code> )	Output
<code>containsGAGA('CTGAGAT')</code>	yes
<code>containsGAGA(rf('geneticString.txt'))</code>	yes
<code>containsGAGA(rf('longerThan1K.py'))</code>	no
<code>containsGAGA(rf('containsGAGA.py'))</code>	yes
<code>yes('CTGAGAT')</code>	yes
<code>yes(rf('geneticString.txt'))</code>	yes
<code>yes(rf('containsGAGA.py'))</code>	yes
<code>yes(rf('yes.py'))</code>	yes
<code>longerThan1K('CTGAGAT')</code>	no
<code>longerThan1K(rf('geneticString.txt'))</code>	yes
<code>longerThan1K(rf('containsGAGA.py'))</code>	no
<code>longerThan1K(rf('longerThan1K.py'))</code>	no
<code>maybeLoop('CTGAGAT')</code>	undefined
<code>maybeLoop('some secret sauce')</code>	no
<code>maybeLoop(rf('containsGAGA.py'))</code>	undefined
<code>maybeLoop(rf('maybeLoop.py'))</code>	yes

Figure 3.3: The outputs of some simple decision programs.

program takes two string parameters,  $P$  and  $I$ . Its output is defined by

$$\text{yesOnString.py}(P, I) = \begin{cases} \text{"yes"} & \text{if } P \text{ is a Python program, } P(I) \text{ is} \\ & \text{defined, and } P(I) = \text{"yes"}, \\ \text{"no"} & \text{otherwise.} \end{cases}$$

In plain English, `yesOnString.py` answers the following question: Does  $P$  return “yes” on string input  $I$ ? If  $P$  isn’t a valid Python program, the answer is “no”. If  $P$  is a valid Python program, but  $P(I)$  is undefined, the answer is “no”. If  $P(I)$  is defined, but equals anything other than “yes”, the answer is “no”. In fact, the answer is “yes” only if  $P(I)$  is defined and equals “yes”.

In contrast to all the Python programs mentioned so far, the source code for `yesOnString.py` isn’t provided. Indeed, we will soon prove that it’s impossible to write such a program. But let’s ignore that inconvenient fact for now.

Figure 3.4 shows several examples of the outputs that `yesOnString.py` would produce, if it existed. For example, the first line outputs “no” because the  $P$ -parameter (“not a program”) isn’t a valid Python program. The second line outputs “no” because `containsGAGA('CAGT')` returns “no”. Check for yourself that the remaining lines are also correct.

### 3.4 THE PROGRAM `yesOnSelf.py`

We saw earlier that the ability of programs to analyze themselves might be the tip of a philosophical iceberg. Let’s follow this hint, and specialize `yesOnString.py`—which analyzes the output of any program on any input—to the particular case of programs that analyze themselves. This gives us

Shell command (with <code>rf = utils.readFile</code> )	Output
<code>yesOnString('not a program', 'CAGT')</code>	no
<code>yesOnString(rf('containsGAGA.py'), 'CAGT')</code>	no
<code>yesOnString(rf('containsGAGA.py'), rf('containsGAGA.py'))</code>	yes
<code>yesOnString(rf('yes.py'), 'CAGT')</code>	yes
<code>yesOnString(rf('yes.py'), rf('yes.py'))</code>	yes
<code>yesOnString(rf('longerThan1K.py'), rf('geneticString.txt'))</code>	yes
<code>yesOnString(rf('longerThan1K.py'), rf('longerThan1K.py'))</code>	no

**Figure 3.4:** The outputs of **yesOnString.py**. All except the first one can be checked using `yesOnStringApprox.py` (see exercise 3.4).

Shell command (with <code>rf = utils.readFile</code> )	Output
<code>yesOnSelf('not a program')</code>	no
<code>yesOnSelf(rf('containsGAGA.py'))</code>	yes
<code>yesOnSelf(rf('yes.py'))</code>	yes
<code>yesOnSelf(rf('longerThan1K.py'))</code>	no
<code>yesOnSelf(rf('yesOnSelf.py'))</code>	yes or no (!?)

**Figure 3.5:** The outputs of **yesOnSelf.py**. The last line is mysterious, since either answer seems correct.

a single-parameter Python program `yesOnSelf.py`, defined formally for string input  $P$  by

$$\text{yesOnSelf.py}(P) = \begin{cases} \text{"yes"} & \text{if } P \text{ is a Python program, } P(P) \text{ is} \\ & \text{defined, and } P(P) = \text{"yes"}, \\ \text{"no"} & \text{otherwise.} \end{cases}$$

In plain English, `yesOnSelf.py` answers the following question: Does  $P$  return “yes” on string input  $P$ ? Rephrased even more simply, the question is, does  $P$  return “yes” when given itself as input? As before, the answer is “no” if  $P$  isn’t a valid Python program, or  $P(P)$  isn’t defined. The answer is “yes” only if  $P(P)$  is defined and equals “yes”.

We will soon see that a correct version of `yesOnSelf.py` is impossible. But if we assume for now that `yesOnSelf.py` could be written, figure 3.5 shows some typical outputs that this program would produce. In the first line, the input “not a program” produces “no” because the input isn’t a valid Python program. In the second line, inputting the source code for `containsGAGA.py` results in a “yes”, because the program `containsGAGA.py` does contain the string GAGA, and therefore outputs “yes” when run on itself. As usual, you should carefully check each line of the table to make sure you understand it.

However, the last line of the table presents a mystery and requires some serious thought. This line asks `yesOnSelf.py` to predict its own output. Strangely, it seems that “yes” and “no” are both correct answers. The output of `yesOnSelf.py` is like a self-fulfilling prophecy: if `yesOnSelf.py` outputs “yes” when

Shell command (with <code>rf = utils.readFile</code> )	Output
<code>notYesOnSelf('not a program')</code>	<code>yes</code>
<code>notYesOnSelf(rf('containsGAGA.py'))</code>	<code>no</code>
<code>notYesOnSelf(rf('yes.py'))</code>	<code>no</code>
<code>notYesOnSelf(rf('longerThan1K.py'))</code>	<code>yes</code>
<code>notYesOnSelf(rf('notYesOnSelf.py'))</code>	<code>!?!?!</code>

**Figure 3.6: The outputs of `notYesOnSelf.py`.** By definition, `notYesOnSelf.py` produces the opposite answer to `yesOnSelf.py`, so this table—except for its last row—is identical to figure 3.5, but with the outputs switched from “yes” to “no”, and vice versa. The last line produces a contradiction, since there is no correct output.

run on itself, then it outputs “yes” when run on itself. But if `yesOnSelf.py` outputs “no” when run on itself, then it outputs “no” when run on itself. This is definitely mysterious, but not actually contradictory. Nevertheless, it is a sign of serious trouble ahead, as we will soon see.

### 3.5 THE PROGRAM `notYesOnSelf.py`

Our next program, called `notYesOnSelf.py`, is a minor tweak on the previous one: it simply reverses the output of `yesOnSelf.py`, so that “yes” becomes “no”, and vice versa. So `notYesOnSelf.py` answers the following question: Is it true that  $P$  does *not* output “yes” when run on itself? Formally, the definition of `notYesOnSelf.py` is

$$\text{notYesOnSelf.py}(P) = \begin{cases} \text{“no”} & \text{if } P \text{ is a Python program, } P(P) \text{ is defined, and } P(P) = \text{“yes”}, \\ \text{“yes”} & \text{otherwise.} \end{cases}$$

As with the previous two programs, we’ll soon prove that `notYesOnSelf.py` can’t exist. But let’s temporarily assume it does exist, and proceed with computing some of the outputs that `notYesOnSelf.py` would produce. For the most part, this is an extremely simple exercise, since we just invert the outputs in figure 3.5. The result is shown in figure 3.6. As usual, check each line in this table to make sure you understand it. For a specific example, consider the third line: `notYesOnSelf(rf('yes.py'))`. Because `yes.py` does produce “yes” when run on itself, the output is “no”.

However, the last line in figure 3.6 deals a devastating blow to our strange collection of Python programs. This line is

```
notYesOnSelf(rf('notYesOnSelf.py'))
```

The above Python command asks the following question:

Is it true that `notYesOnSelf.py` does *not* output “yes” when run on itself?

There are only two possible answers to this question: yes or no. But both possibilities lead to contradictions. If the answer to the question is yes, then

```

1 from yesOnString import yesOnString
2 def yesOnSelf(progString):
3     return yesOnString(progString, progString)

5
6 def notYesOnSelf(progString):
7     val = yesOnSelf(progString)
8     if val == 'yes':
9         return 'no'
10    else:
11        return 'yes'

```

**Figure 3.7: The programs yesOnSelf.py and notYesOnSelf.py.** These programs are trivial to write, assuming that yesOnString.py exists. But as we have shown, this leads to the conclusion that yesOnString.py cannot exist.

we know notYesOnSelf.py doesn't output "yes" when run on itself, so the answer should in fact have been no—a contradiction. So the first possibility is ruled out; let's consider the second possibility. If the answer to the question is no, then we know notYesOnSelf.py *does* output "yes" when run on itself, so the answer should in fact have been yes—again a contradiction. Hence, the mere existence of the program notYesOnSelf.py creates a contradiction, and we must conclude that notYesOnSelf.py cannot exist. Hence we have proved

**Claim 3.2.** The Python program notYesOnSelf.py cannot exist.

### 3.6 yesOnString.py CAN'T EXIST EITHER

We've achieved one of the objectives of this chapter: we've proved that one particular Python program (notYesOnSelf.py) is impossible. But notYesOnSelf.py is a strange and obscure program—indeed, it was constructed in a deliberately contrived fashion, just so that it would produce a contradiction. Can we prove the impossibility of a more sensible program? Yes we can:

**Claim 3.3.** The Python program yesOnString.py cannot exist.

*Proof of the claim.* This is a proof by contradiction. Assume yesOnString.py does exist. Then we can easily use it to write the two programs yesOnSelf.py and notYesOnSelf.py shown in figure 3.7. But this contradicts our previous result that notYesOnSelf.py cannot exist. Hence our initial assumption was incorrect, and we conclude yesOnString.py cannot exist either. □

The flow of ideas in this proof is shown more concretely in figure 3.8.

### A compact proof that yesOnString.py can't exist

We have just devoted several pages to proving that yesOnString.py can't exist. In fact, this proof can be done much more compactly—it's usually presented in

Program name	Program behavior
<code>yesOnString(<math>P, I</math>)</code>	<ul style="list-style-type: none"> <li>- return “yes”, if <math>P(I) = \text{“yes”}</math></li> <li>- return “no”, otherwise</li> </ul>
$\downarrow$	
<code>yesOnSelf(<math>P</math>)</code>	<ul style="list-style-type: none"> <li>- return “yes”, if <math>P(P) = \text{“yes”}</math></li> <li>- return “no”, otherwise</li> </ul>
$\downarrow$	
<code>notYesOnSelf(<math>P</math>)</code>	<ul style="list-style-type: none"> <li>- return “no”, if <math>P(P) = \text{“yes”}</math></li> <li>- return “yes”, otherwise</li> </ul>

**Figure 3.8:** A sequence of three decision programs that cannot exist. The last program, `notYesOnSelf.py`, is obviously impossible since it produces a contradiction when run on itself. However, each of the programs can be produced easily by invoking the one above it (shown by the arrows). Therefore, none of the three programs can exist.

```

1 from yesOnString import yesOnString
2 def weirdYesOnString(progString):
3     if yesOnString(progString, progString)=='yes':
4         return 'no'
5     else:
6         return 'yes'
```

**Figure 3.9:** The program `weirdYesOnString.py`.

only a few lines. Because it will be important for us to understand the compact version of the proof, we will now restate the previous claim, and give the compact proof.

**Claim 3.4.** The Python program `yesOnString.py` cannot exist.

*Proof of the claim.* Suppose `yesOnString.py` does exist, and argue for a contradiction. Then we can write the program `weirdYesOnString.py`, shown in figure 3.9. This program has the same behavior as `notYesOnSelf.py`: when it is given itself as input, `weirdYesOnString.py` returns “yes” if and only if it returns “no”. So all possible behaviors result in a contradiction, and the claim is proved.  $\square$

So why did we devote several pages to proving this claim earlier, when it’s possible to do it with a 6-line Python program and a 4-sentence proof? The reason is that the Python program `weirdYesOnString.py`, and the 4-sentence proof that goes with it, contain several subtle ideas. When presented in its compact form, the proof seems almost magical, and it’s hard to avoid the feeling that some kind of cheap mathematical trick has been used. If you feel that way, you’re in good company: the very first person to publish a proof of this kind was

Alan Turing, and he had similar concerns. In his 1936 paper, “On computable numbers,” Turing writes,

This proof, although perfectly sound, has the disadvantage that it may leave the reader with a feeling that “there must be something wrong”.

Turing then goes on to give a more detailed and constructive proof of his result. For the same reason, the presentation in this chapter attempts to unpack the key subtle ideas, breaking them out into the separate programs `yesOnString.py`, `yesOnSelf.py`, and `notYesOnSelf.py`. Each of these programs introduces just one new idea, and demonstrates that the idea is a genuinely plausible programming technique. Specifically,

- `yesOnString.py` introduces the idea that programs can analyze other programs and report what they might output for various inputs;
- `yesOnSelf.py` introduces the idea that we can analyze how a program will behave when given itself as input—this idea is often known as “diagonalization,” in reference to a proof technique first invented by the mathematician Georg Cantor;
- `notYesOnSelf.py` introduces the idea of inverting the output of a decision program (i.e., switching the yes/no outputs), specifically for the purpose of producing a self-contradictory program.

Combining all three ideas into the 6-line program of figure 3.9, we get our slightly mysterious—but perfectly correct—proof that `yesOnString.py` can’t exist.

## 3.7 PERFECT BUG-FINDING PROGRAMS ARE IMPOSSIBLE

The impossibility of writing the program `yesOnString.py` is interesting, but perhaps it’s not immediately clear why this program might be useful. In contrast, our next example would be incredibly useful if it were actually possible: we will discuss the program `crashOnString.py`, and prove that it too cannot exist. The program `crashOnString(P, I)` takes as input another program *P*, and a string *I*. It returns “yes” if *P(I)* crashes, and “no” otherwise. (For our purposes, the word “crash” here means the same thing as “throws a Python exception.”) In other words, `crashOnString.py` tells us if there’s a bug that causes a crash in program *P*, when run on input *I*—an extremely useful piece of information. In the rest of this chapter, we will use “bug” to mean “bug that causes a crash.” In fact, it can be shown that it’s equally impossible for an algorithm to find all instances of other types of bugs too, but we restrict to crashes for simplicity.

Figure 3.10 summarizes the argument proving that `crashOnString.py` is impossible. The argument in figure 3.10 is rather similar to the argument for `yesOnString.py`. First, we create a sequence of carefully crafted programs that will produce a contradiction. In this case, the two programs are `crashOnSelf.py` and `weirdCrashOnSelf.py`, shown in figure 3.11. Then we show the last program in the sequence produces contradictory behavior, and is therefore

Program name	Program behavior
<code>crashOnString(<math>P, I</math>)</code>	<ul style="list-style-type: none"> <li>– return “yes”, if <math>P</math> crashes on input <math>I</math></li> <li>– return “no”, otherwise</li> </ul>
 <code>crashOnSelf(<math>P</math>)</code>	<ul style="list-style-type: none"> <li>– return “yes”, if <math>P</math> crashes on input <math>P</math></li> <li>– return “no”, otherwise</li> </ul>
 <code>weirdCrashOnSelf(<math>P</math>)</code>	<ul style="list-style-type: none"> <li>– return without crashing, if <math>P</math> crashes on input <math>P</math></li> <li>– crash, otherwise</li> </ul>

**Figure 3.10:** A sequence of three programs that cannot exist. The last program, `weirdCrashOnSelf.py`, is obviously impossible, since it produces a contradiction when run on itself. However, each of the programs can be produced easily by invoking the one above it (shown by the arrows). Therefore, none of the three programs can exist.

```

1 from crashOnString import crashOnString
2 def crashOnSelf(progString):
    return crashOnString(progString, progString)

```

```

1 from crashOnSelf import crashOnSelf
2 def weirdCrashOnSelf(progString):
    val = crashOnSelf(progString)
    if (val == 'yes'):
        return 'did not crash!'
    else:
        # deliberately crash (divide by zero)
        x = 1 / 0

```

**Figure 3.11:** The programs `crashOnSelf.py` and `weirdCrashOnSelf.py`. These programs are trivial to write, assuming that `crashOnString.py` exists. But as we will show, this leads to the conclusion that `crashOnString.py` cannot exist.

impossible. Finally, we conclude the first program in the sequence is also impossible. The argument is formalized in the following two claims.

**Claim 3.5.** The Python program `weirdCrashOnSelf.py` cannot exist.

*Proof of the claim.* This is a proof by contradiction. Assume `weirdCrashOnSelf.py` exists. Note carefully the strange behavior of `weirdCrashOnSelf.py`: if its input crashes on itself, this program returns without crashing (line 5); if its input doesn't crash on itself, this program deliberately crashes (it performs a divide-by-zero at line 8). So consider what happens when we execute the command

```
>>> weirdCrashOnSelf(rf('weirdCrashOnSelf.py'))
```

Let us refer to this Python shell command as  $W$ . The execution of  $W$  either crashes, or doesn't crash. We will now see that both possibilities lead to a contradiction:

1.  **$W$  crashes.** If  $W$  crashes, the calculation of `val` at line 3 returns “yes”, but that means the program goes on to terminate successfully without crashing. (In fact, it returns the string “did not crash!”, but that detail is irrelevant.) In summary, if  $W$  crashes, then it doesn't crash, so this possibility leads to a contradiction.
2.  **$W$  doesn't crash.** If  $W$  doesn't crash, the calculation of `val` at line 3 returns “no”, but that means the program goes on to crash with a divide-by-zero. In summary, if  $W$  doesn't crash, then it does crash, so this possibility also leads to a contradiction.

All possibilities lead to contradictions, so we conclude that `weirdCrashOnSelf.py` can't exist.  $\square$

Finally, we can also prove the following claim:

**Claim 3.6.** The Python program `crashOnString.py` cannot exist.

*Proof of the claim.* This is a proof by contradiction. Assume `crashOnString.py` does exist. Then we can easily use it to write the two programs `crashOnSelf.py` and `weirdCrashOnSelf.py` shown in figure 3.11. This contradicts the previous claim. Hence our initial assumption was incorrect, and we conclude `crashOnString.py` cannot exist either.  $\square$

## 3.8 WE CAN STILL FIND BUGS, BUT WE CAN'T DO IT PERFECTLY

We now agree that `crashOnString.py` can't exist, so it's impossible to determine automatically whether an arbitrary SISO Python program will crash on a given input. But recall the discussion on page 26: we restrict our definition of computer program to “SISO Python program” purely for convenience. It can be shown these results apply to any kind of computer hardware and software. Therefore, it's equally impossible to find all the bugs in Xbox video games, operating systems like Windows and OS X, or code that launches nuclear missiles. So this is an important result with deep implications for a society that relies on technology.

On the other hand, it's important to realize what we have *not* proved. We have not proved that it's impossible to find bugs in computer programs automatically. We have merely proved you can't write a program that will find all the bugs in all programs. In fact, software companies like Microsoft employ thousands of people whose job it is to write automated bug-detecting software. Academic computer scientists have studied the problem of bug detection intensively for decades, and continue to make breakthroughs in this field, which is known as *software verification*. So, there are many effective approaches for detecting certain types of bugs in many types of programs, but we know these approaches can never work perfectly.

## EXERCISES

**3.1** Use proof by contradiction to prove the following statements:

- (a) There are infinitely many positive integers. (Many young children appear to understand and accept this proof, which is extremely useful when asked, what is the largest number?)
- (b) There are infinitely many negative integers.
- (c) There are infinitely many even numbers.
- (d) There is no smallest positive real number.

**3.2** The next two exercises will deepen your understanding of how programs can analyze themselves.

- (a) Write a program `countChars.py` that returns the number of characters in its input. For example, on input “CATTG” the output is “5”.
- (b) What is the output of `countChars.py` when given itself as input?
- (c) Add a few space characters to the end of `countChars.py`, and save this as the new program `countCharsB.py`. What is the output of `countCharsB.py` when given itself as input?
- (d) It’s clear that `countChars.py` and `countCharsB.py` are equivalent, since the added space characters make no difference to the execution of the program. Explain why these equivalent programs gave different answers, in parts (b) and (c) above, to the following question: What is the output of this program when given itself as input?

**3.3** As stated in figure 3.3, the output of the command

```
>>> containsGAGA(rf('containsGAGA.py'))
```

is “yes”. Write a new version of `containsGAGA.py`, called `containsGA_GA.py`. The new version should be equivalent to the old one (i.e., it produces exactly the same outputs given the same inputs). In addition, the result of the commands

```
>>> containsGA_GA(rf('containsGA_GA.py'))
```

and

```
>>> containsGAGA(rf('containsGA_GA.py'))
```

should both be “no”.

**3.4** Study the source code of the program `yesOnStringApprox.py`, which is provided with the book resources. This program is a (very!) approximate version of `yesOnString.py`: it produces correct results on only four particular inputs. Also study the related programs `yesOnSelfApprox.py` and `notYesOnSelfApprox.py`. By analyzing the source code and/or direct experimentation in IDLE, determine the output of the following Python commands:

- (a) `yesOnStringApprox(rf('longerThan1K.py'), rf('longerThan1K.py'))`
- (b) `yesOnStringApprox(rf('maybeLoop.py'), rf('maybeLoop.py'))`
- (c) `yesOnSelfApprox(rf('longerThan1K.py'))`
- (d) `notYesOnSelfApprox(rf('containsGAGA.py'))`

**3.5** Consider a hypothetical Python decision program called `noOnString.py`. This program is similar to `yesOnString.py`, but detects “no” outputs instead. Specifically, `noOnString.py` takes two parameters ( $P, I$ ) and outputs “yes” if and only if  $P(I) = \text{``no''}$ . As you might expect, `noOnString.py` cannot exist, but we can approximate it for a few particular values of  $P$ , using the same technique as `yesOnStringApprox.py` (see previous exercise). Write a program called `noOnStringApprox.py` that has this desired behavior. Your approximation should work correctly for the same inputs as `yesOnStringApprox.py`.

**3.6** Similarly to the previous question, consider a hypothetical Python decision program called `noOnSelf.py` that takes a parameter  $P$  and outputs “yes” if and only if  $P(P) = \text{``no''}$ . Write an approximation `noOnSelfApprox.py` that approximates `noOnSelf.py` for the same values of  $P$  as in the previous question.

**3.7** Define `noOnSelf.py` as in the previous question. Assuming that `noOnSelf.py` exists, what should be the output of the following commands?

- (a) `noOnSelf(rf('noOnSelf.py'))`
- (b) `noOnSelf(rf('yesOnSelf.py'))`
- (c) `yesOnSelf(rf('noOnSelf.py'))`

**3.8** Define `noOnSelf.py` as in the previous questions. Although `noOnSelf.py` is similar to `notYesOnSelf.py`, these programs are not equivalent. Give an example of an input for which these two programs (if they existed!) would give different outputs. Briefly explain your answer.

**3.9** Alter `yesOnStringApprox.py` so that it works correctly on at least three additional interesting inputs.

**3.10** Consider a Python program `definedOnString.py`. This program takes two parameters ( $P, I$ ). It returns “yes” if  $P(I)$  is defined, and “no” otherwise. Using an approach similar to figure 3.9 and claim 3.4 (page 38), prove that `definedOnString.py` cannot exist.

**3.11** Consider a Python program `longerThan10.py`. This program takes two parameters ( $P, I$ ). It returns “yes” if  $P(I)$  is a string of more than 10 characters, and “no” otherwise. Using an approach similar to figure 3.9 and claim 3.4 (page 38), prove that `longerThan10.py` cannot exist.

**3.12** Consider a Python program `startsWithZ.py`. This program takes two parameters ( $P, I$ ). It returns “yes” if  $P(I)$  is a string that starts with the character `z`, and “no” otherwise. Using an approach similar to figure 3.9 and claim 3.4 (page 38), prove that `startsWithZ.py` cannot exist.

**3.13** Using the previous three questions as inspiration, describe another impossible Python program. Try to come up with an interesting program: for example, `longerThan5.py` and `startsWithABC.py` are indeed impossible, but they are too similar to the earlier questions to be interesting.

**3.14** Explain the flaw in the following “proof” that it is impossible to write a program that counts the number of lines in its input:

Suppose that `countLines.py` exists as described and argue for a contradiction. By editing the return statement, we can create a new

```
1 from countLines import countLines
2 from countLinesPlus1 import countLinesPlus1
3 def weirdCountLines(inString):
4     if inString == rf('weirdCountLines.py'):
5         return countLinesPlus1(inString)
6     else:
7         return countLines(inString)
```

Figure 3.12: The Python program `weirdCountLines.py`.

program `countLinesPlus1.py`; this new program returns one more than the number of lines in the input. We can then create `weirdCountLines.py`, as shown in figure 3.12. This program operates as follows: if given itself as input, the output is one more than the number of lines in the input; otherwise the output is the number of lines in the input. This produces a contradiction when `weirdCountLines.py` is run with itself as input, since the output is not equal to the number of lines in the input.

**3.15** Suppose a new video game called *Best Game Ever* is released. You play the game with a friend and are disappointed to discover the game is rather buggy. Your friend, who has taken a course on the theory of computation, says, “Actually, it’s impossible for a computer program to automatically find all the bugs in the *Best Game Ever* software.” Do you agree with your friend? Explain your answer.

# 4



## WHAT IS A COMPUTATIONAL PROBLEM?

We know that every age has its own problems, which the following age either solves or casts aside as profitless and replaces by new ones.

— David Hilbert, Lecture delivered before the International Congress of Mathematicians in Paris (1900)

Chapter 1 gave an overview of tractable, intractable, and uncomputable computational problems. But the concept of a *computational problem* was never defined formally. In this chapter, we remedy that: computational problems will be given a rigorous mathematical definition.

But first, let's step back and understand how computational problems fit into what we have learned so far. Chapter 2 gave three equivalent definitions of *computer program* (page 26), and we settled on SISO Python programs as our working definition. Chapter 3 demonstrated that certain types of programs—such as perfect bug-finding programs—cannot exist. So what's the connection between the computer *programs* of the previous two chapters, and the computational *problems* covered in this chapter? The answer is

*programs solve problems*

That is, a computational problem describes what we want to compute. A computer program describes one particular way of computing the solution to a problem. As a concrete example, figure 4.1 describes a classic problem in computer science: sorting a list into lexicographical (i.e., dictionary) order. We call this problem **SORTWORDS**.

Figure 4.2 shows three attempts at solving this problem. The first attempt, `pythonSort.py`, uses the built-in Python function `sorted` to efficiently sort the list. Incidentally, at line 3 of `pythonSort.py` we see the string `join()` method, which was first mentioned in exercise 2.1. Don't be confused by the strange syntax; the code `' '.join(words)` creates a single string from the `words` array, with each element separated by a space character.

Returning to figure 4.2, the second attempt at solving **SORTWORDS** is `bubbleSort.py`. This program uses a well-known algorithm called *bubble sort*

**PROBLEM SORTWORDS**

- **Input:** A list of words separated by whitespace. (Example: “banana grape banana apple”.)
- **Solution:** A sorted version of the input list, in which the words are sorted lexicographically and separated by whitespace. Repeated words are included. (Solution for above example: “apple banana banana grape”.)

**Figure 4.1:** Description of the computational problem SORTWORDS.

to sort the list. The algorithm is correct, but inefficient. Spend a few minutes now to make sure you understand how and why this implementation of bubble sort works correctly.

The third program in figure 4.2 is `brokenSort.py`. This program is almost identical to the previous one. The only difference is the use of “`<=`” instead of “`<`” at line 13. This is a bug, since the function `isSorted()` will incorrectly report that a list with two consecutive identical elements is unsorted. Hence, the program works correctly on lists with distinct elements, but fails if there are any repeated elements. Specifically, the `while` loop at line 3 will never terminate, so `brokenSort.py` enters an infinite loop on inputs with repeated elements.

This sorting example demonstrates several obvious but important points. First, note the difference between figure 4.1 and figure 4.2: figure 4.1 is an abstract computational problem; figure 4.2 contains concrete attempts to solve that problem using computer programs. Second, there can be many, fundamentally different—yet correct—methods of solving a single problem (`pythonSort.py` and `bubbleSort.py` are examples of this). Third, proposed methods of solution can be wrong in subtle ways (e.g., `brokenSort.py` works correctly on many inputs, but fails on others). A computer program does not truly solve a problem unless it terminates on all inputs, and produces a correct answer.

So, the goal of this chapter is to first give a formal definition of “computational problem”; and then give a formal definition of what it means to “solve” a computational problem. To do that, we must first understand various other mathematical concepts, including graphs, alphabets, strings, and languages. We’ll step through each of these in turn.

## 4.1 GRAPHS, ALPHABETS, STRINGS, AND LANGUAGES

### Graphs

Many computational problems are stated in terms of graphs, so it’s important to know some graph terminology. Figure 4.3 demonstrates the five most important concepts:

- A *graph* is a collection of *nodes* with edges between some of the nodes. Sometimes we call this an *undirected graph*, because the edges have no

```

1 def pythonSort(inString):
2     words = sorted(inString.split());
3     return ' '.join(words) # single string of words separated by spaces

4
5 def bubbleSort(inString):
6     words = inString.split();
7     while not isSorted(words):
8         for i in range(len(words)-1):
9             if words[i+1] < words[i]:
10                 # swap elements i and i+1
11                 words[i], words[i+1] = words[i+1], words[i]
12     return ' '.join(words) # single string of words separated by spaces

13
14 def isSorted(words):
15     for i in range(len(words)-1):
16         if words[i+1] < words[i]:
17             return False
18     return True

19
20 def brokenSort(inString):
21     words = inString.split();
22     while not isSorted(words):
23         for i in range(len(words)-1):
24             if words[i+1] < words[i]:
25                 # swap elements i and i+1
26                 words[i], words[i+1] = words[i+1], words[i]
27     return ' '.join(words) # single string of words separated by spaces

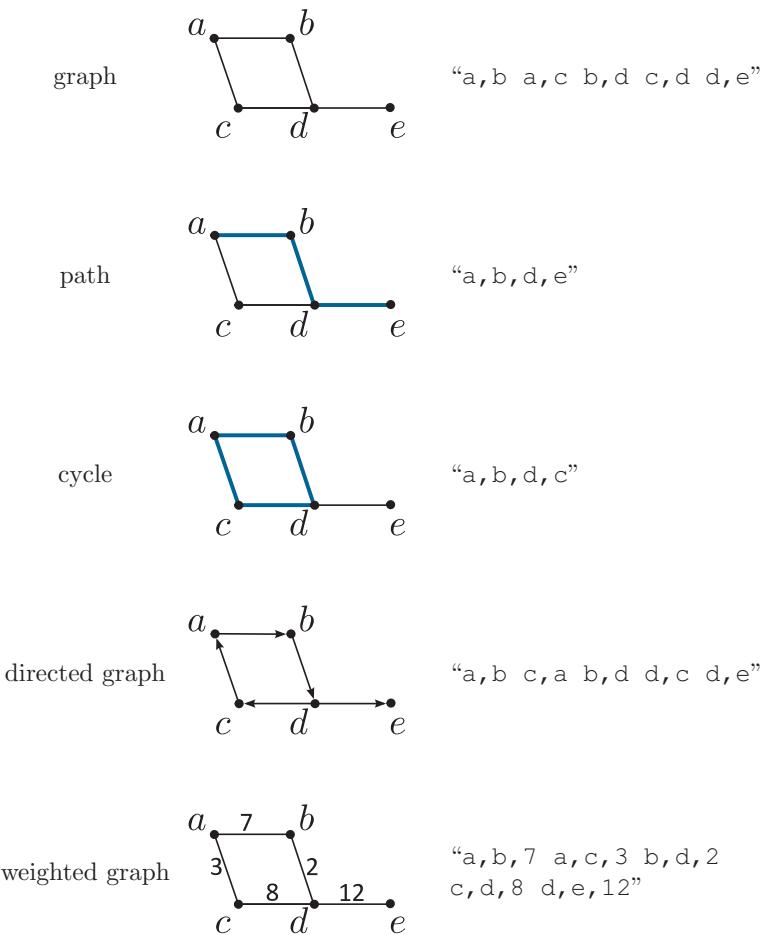
28
29 def isSorted(words):
30     for i in range(len(words)-1):
31         # next line is a BUG: should be "<", not "<="
32         if words[i+1] <= words[i]:
33             return False
34     return True

```

**Figure 4.2:** Three Python programs: `pythonSort.py`, `bubbleSort.py`, and `brokenSort.py`. The first two solve the computational problem SORTWORDS; the third does not solve this problem because it does not terminate on all instances.

direction. In some books, nodes are called *vertices*. Sometimes we allow multiple edges between the same pair of nodes.

- A *path* is a sequence of nodes joined by edges. Usually we insist that the path does not repeat any edges. Paths are usually described as a sequence of nodes, without explicitly listing the edges involved. A *Hamilton path* is a path that visits each node exactly once, without repeating any edges.
- A *cycle* is a path that starts and ends at the same node, without repeating any edges. A *Hamilton cycle* is a cycle that visits each node exactly once, except for the start node which is visited again at the end.

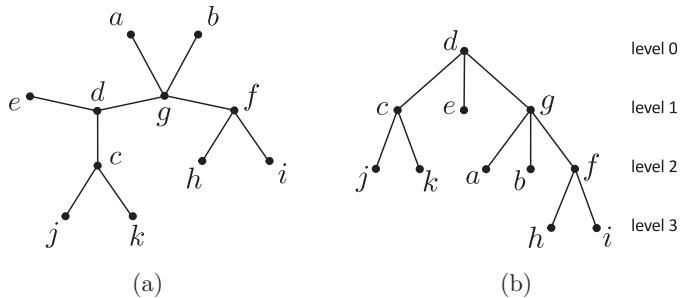


**Figure 4.3: Graph concepts.** The right-hand column shows possible ASCII string representations of the corresponding examples.

- A *directed graph* is the same as a graph, but with directed edges.
- A *weighted graph* is the same as a graph, but each edge has a numerical weight.

Various obvious combinations of these definitions are possible, such as directed weighted graphs, directed cycles, and directed paths. A graph is *connected* if there's a path between every pair of nodes. Node  $v$  is a *neighbor* of node  $w$  if there's an edge from  $w$  to  $v$ .

We will be using graphs, paths, and so on as inputs and outputs to computer programs. But as we saw in chapter 2, we're most interested in computer programs that use only ASCII strings as input and output. Fortunately, it's easy to convert graphs, paths, and other graph-theoretic concepts into ASCII strings. The exact conventions for doing so are not important, but the right column of



**Figure 4.4:** (a) A tree with 11 nodes. (b) The same tree, this time represented as a rooted tree with root node  $d$ .

figure 4.3 shows one possible way of doing this, for each of the five main object types.

## Trees and rooted trees

Trees and rooted trees are special kinds of graphs that are often encountered in computer science. Their definitions are as follows:

- A *tree* is a connected graph with no cycles. Figure 4.4(a) gives an example.
- A *rooted tree* is a tree in which one of the nodes has been designated the *root* of the tree. Figure 4.4(b) gives an example. This is exactly the same tree as in (a), but now node  $d$  has been designated the root. Note that we can lay out a rooted tree in *levels*: the root is at level 0, its neighbors are at level 1, and so on. This also gives us a notion of parent and child nodes. The parent of a node is its neighbor in the level above; the children of a node are its neighbors in the level below. In figure 4.4(b), for example, the parent of  $k$  is  $c$ , and the children of  $g$  are  $a$ ,  $b$ , and  $f$ . A node with no children is called a *leaf*. In figure 4.4(b), the leaves are  $j$ ,  $k$ ,  $e$ ,  $a$ ,  $b$ ,  $h$ ,  $i$ . You may have noticed that, in computer science, rooted trees are upside down. Whereas a real-world tree has its roots at the bottom and leaves at the top, computer science trees have the root at the top and leaves at the bottom. Get used to it!

In practice, rooted trees are so common in computer science that they are often simply called “trees.” It’s usually clear from the context whether or not a tree has a designated root. And we can easily convert a tree into a rooted tree by selecting any node as the root.

## Alphabets

An *alphabet* is a finite set of symbols. Common examples include the binary alphabet  $\{0, 1\}$ , the alphabet of digits  $\{0, 1, \dots, 9\}$ , the alphanumeric alphabet  $\{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9\}$ , and the ASCII alphabet—the set of 128 characters described on page 21. Note that in ordinary English, “the alphabet” refers to the set of English letters  $\{a, b, c, \dots, z\}$ . Don’t let this confuse you: in

computer science, “the” English alphabet is just one of many possible alphabets. In this book, we will use the ASCII alphabet whenever possible.

Sometimes, we will need to translate between alphabets. The exact details of how to do this are unimportant, since it’s always possible to agree on *some* unambiguous translation technique. For example, to translate from ASCII to binary, we can use the internationally agreed assignment of 7-bit binary numbers to ASCII characters. (Look it up, if you’re interested—“a” becomes “1100001” and “%” becomes “0100101”, for example.) Thus, any single ASCII character gets translated into seven binary symbols, and vice versa.

In theoretical computer science, an alphabet is usually represented by the Greek letter  $\Sigma$  (an uppercase sigma). Perhaps this is because the “s” in sigma reminds us of the “s” in symbol, and an alphabet is a set of symbols. In any case, this use of  $\Sigma$  has nothing to do with the more usual meaning of  $\Sigma$  as a summation symbol. So don’t be confused by this: if a proof or definition begins with “Let  $\Sigma = \{0, 1\}$ ,” it is merely stating that the binary alphabet will be used.

Why are we interested in alphabets? Because they define the basic building blocks for the input and output of computer programs. And this leads to our next concept, strings.

## Strings

Given an alphabet  $\Sigma$ , a *string* on  $\Sigma$  is a finite sequence of zero or more characters from  $\Sigma$ . The order matters, and repetitions are permitted. Thus, “1010”, “0101”, and “1” are all binary strings (i.e., strings on the binary alphabet). And the following are all ASCII strings: “asdf\$\_%j\*”, “Hello, world”, and “http://xyz.org”. But the sequence “1111...” (a sequence of infinitely many 1s) is *not* a string, since strings must be finite. On the other hand, strings are permitted to be empty, and we use the special symbol  $\epsilon$  (the Greek letter epsilon) to represent the empty string on any alphabet.

Strings will be represented in typewriter font, and will usually—but not always—be surrounded by quotation marks. Thus, “yes” and yes mean exactly the same thing, and we will use whichever version seems more readable in any given context.

We’ll often need to work with repetitions, concatenations, and lengths of strings, so let’s define some notation for these things. Given two strings  $s$  and  $t$ , the concatenation of  $s$  and  $t$  is written as  $st$ . (Example: If  $s = “abc”$  and  $t = “xy”$  then  $st = “abcxy”$ .) Given a string  $s$ , the notation  $s^n$  represents  $s$  repeated  $n$  times. (Examples: If  $s = “abc”$ , then  $s^0 = \epsilon$ ,  $s^1 = “abc”$ ,  $s^2 = “abcabc”$ ,  $s^3 = “abcabcabc”$ .) The length of string  $s$  is written  $|s|$ . (Examples:  $|\epsilon| = 0$ ;  $|“abc”| = 3$ ). Note that if  $|s| = k$ , then  $|s^n| = kn$ .

Because we rely very heavily on ASCII strings in this book, this would be a good time to reread the section introducing ASCII (section 2.3, page 21). The most essential points are as follows: (i) since ASCII includes newline characters, we can think of multiline blocks of text as being single strings; (ii) the entire contents of any ASCII text file stored on a computer can be regarded as a single string; (iii) the source code of any computer program written in ASCII can be

regarded as a single string; (iv) although programming languages such as Java and Python specify that the larger Unicode alphabet may be used for source code, we assume in this book that all our programs are written in ASCII, which is a subset of Unicode.

## Languages

A *language* or *formal language* is a set of strings from a given alphabet. A language can be finite or infinite, and may or may not contain the empty string. Languages on the ASCII alphabet include the following examples:

- **Binary strings** are strings consisting of only 0s and 1s, such as “1000110”.
- **English words** are strings that appear in (say) the 1989 second edition of the Oxford English Dictionary. Thus, “apple” is in this language, but “sdijofs” is not.
- **Any finite set of strings** is a language. For example, the following set is a language:

{“apple”, “banana”, “orange”}.

- **The empty language** is the language that has no strings in it. This is sometimes denoted by  $\emptyset$  (the empty set symbol), and sometimes by { }, which also represents an empty set.
- **The language consisting of the empty string** is the language  $\{\epsilon\}$ . Note that this language is not the same thing as the empty language  $\emptyset$ . Whereas  $\emptyset$  contains zero strings,  $\{\epsilon\}$  contains one string. Some further examples might help demonstrate this distinction: the language {“abc”, “xyz”} contains two strings of length 3; the language {“abc”,  $\epsilon$ } contains two strings, one of length 3 and one of length 0; the language {“abc”} contains one string of length 3; the language { $\epsilon$ } contains one string of length 0; and the language { } (also written  $\emptyset$ ) contains zero strings.
- **Three or more g's** is the language of ASCII strings containing at least three g's, such as “agbgcg”, “gggggggg”, and “gzxcgyy”.
- **Java programs** comprise the language of ASCII strings that produce no errors when compiled by (some particular version of) the Java compiler.
- **Python programs** comprise the language of ASCII strings that meet our formal definition of a Python program (page 24).
- **Prime numbers** comprise the language of numeric strings that represent prime numbers, such as “5” and “31”.
- **All strings**, the set of all ASCII strings, is a language. We denote this language by ASCII\*.
- **Python programs that return prime numbers** is a language of ASCII strings that represent a particular set of Python programs. A program is in this language if, when run with the empty string as input, it returns a prime number (or more precisely, it returns the string representation of a prime number).
- **Strings accepted by `SomeProg.py`** form a language consisting of all ASCII input strings for which a given, fixed Python program `SomeProg.py`

returns “yes”. (As we’ll discover on page 65, this is the language *recognized* by `SomeProg.py`.)

The “all strings” example above introduces some important notation: given any alphabet  $\Sigma$ , the set of all possible strings on that alphabet is a language denoted by  $\Sigma^*$ . The “ $*$ ” symbol here is known as the *Kleene star*. It is named after the mathematician Stephen Kleene and is described in more detail below.

It’s important to distinguish between three different meanings of the word “language”: first, there is the everyday meaning of the word as a *language spoken by humans* (such as the English language or the Japanese language); second, there is the notion of a *programming language* (such as Java, C++, or Python); and third, our concept of a *formal language* in theoretical computer science, as a *set of strings on a given alphabet*. There is some overlap between the three meanings. For example, the everyday concept of the “English language” closely resembles the set of strings described as “English words” above—it is just a finite list of words designated as English words by a particular dictionary. And the everyday concept of “Java programming language” closely resembles the formal language of “Java programs” defined above. Nevertheless, we will be using the strict technical meaning of language as a set of strings from now on.

Why are we interested in this definition? Because many problems in theoretical computer science can be boiled down to whether or not a program can recognize the strings in a particular formal language. For example, the Java compiler can recognize all strings in the language of Java programs defined above. And we can easily write a program that recognizes the “prime numbers” language defined above. But the last two examples above are more difficult: can we write programs that recognize these languages? The next three chapters will be devoted to questions like this.

Because languages are sets, we can apply standard set operations such as union, intersection, and complement to languages. And there are two further important operations on languages: *concatenation* and *repetition*. Formally, if  $L$ ,  $L_1$ , and  $L_2$  are languages on  $\Sigma$ , then we have the following operations:

- **union:**  $L_1 \cup L_2$  is the language of all strings in  $L_1$  or  $L_2$ .
- **intersection:**  $L_1 \cap L_2$  is the language of all strings in both  $L_1$  and  $L_2$ .
- **complement:**  $\bar{L}$  is the language of all strings in  $\Sigma^*$  but not in  $L$ ; for example, if  $\Sigma = \{G\}$  and  $L = \{G, G^3, G^5, \dots\}$  then  $\bar{L} = \{\epsilon, G^2, G^4, G^6, \dots\}$ .
- **concatenation:**  $L_1 L_2$  is the language of all strings formed by concatenating a string in  $L_1$  with a string in  $L_2$ ; for example, if  $L_1 = \{C, CC\}$  and  $L_2 = \{A, T\}$  then  $L_1 L_2 = \{CA, CT, CCA, CCT\}$ .
- **repetition (Kleene star):**  $L^*$  is the language formed by repeating elements of  $L$  zero or more times in arbitrary combinations; for example, if  $L = \{G, TT\}$  then

$$L^* = \{\epsilon, G, TT, GG, GTT, TTG, TTTT, \dots\}.$$

We will study the Kleene star operation in more detail in chapter 9.

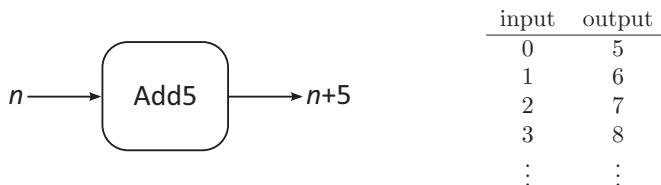


Figure 4.5: Two ways of thinking about the Add5 function.

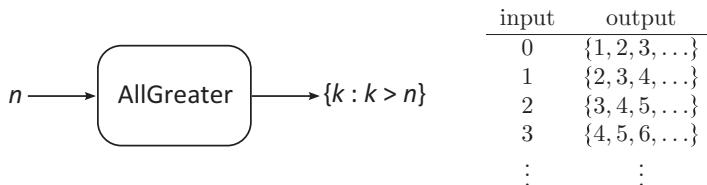


Figure 4.6: Two ways of thinking about the AllGreater function.

## 4.2 DEFINING COMPUTATIONAL PROBLEMS

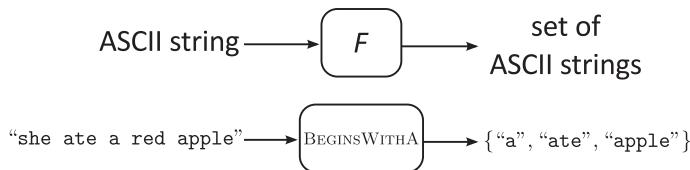
In mathematics, a *function* maps every object in some set, called the *domain*, to an object in a (possibly different) set, called the *codomain*. A simple example would be the Add5 function, which takes a nonnegative integer  $n$  as input and maps it to the output  $n + 5$ . The domain and codomain of Add5 are both  $\mathbb{N}$ , the set of natural numbers  $\{0, 1, 2, \dots\}$ . Figure 4.5 shows two common ways of thinking about functions: either as a box that receives inputs and produces outputs, or as a table of inputs with corresponding outputs.

Of course, the domain and codomain need not be the same. One particularly interesting case is when the function produces *sets* of elements from the domain. For example, we can define the function AllGreater, whose domain is again  $\mathbb{N}$ . On input  $n$ , AllGreater maps  $n$  to the set of all integers greater than  $n$ . In mathematical notation,  $\text{AllGreater}(n) = \{k \in \mathbb{N} : k > n\}$ —see figure 4.6 for two ways of thinking about this function. Note that AllGreater always maps a single natural number to a set of natural numbers. Thus, we can take the codomain of AllGreater to be the set of all *subsets* of  $\mathbb{N}$ .

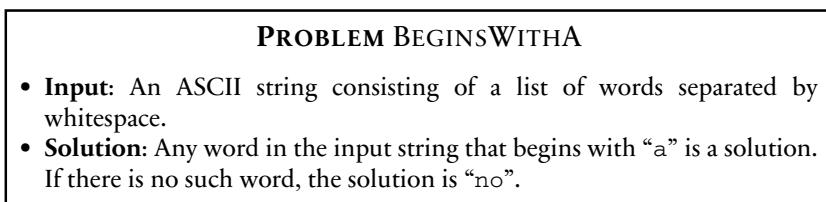
The idea of a function mapping elements of its domain to subsets of its domain underlies the key definition of this chapter:

**Definition of a “computational problem” or a “general computational problem.”** Given an alphabet  $\Sigma$ , a *computational problem* is a function mapping strings on  $\Sigma$  to sets of strings on  $\Sigma$ . (The phrase *general computational problem* means exactly the same thing as *computational problem*, but emphasizes the contrast with decision problems, which are defined later.)

In other words, a computational problem is a function whose domain is  $\Sigma^*$  and whose codomain is the subsets of  $\Sigma^*$ . Given a computational problem  $F$  and a string  $s$  on  $\Sigma$ , the value of  $F(s)$  is a set of strings on  $\Sigma$ . We call  $F(s)$  the



**Figure 4.7:** *Top:* A computational problem  $F$  on the ASCII alphabet. Each input is an ASCII string, which is mapped to a set of solutions that are also ASCII strings. *Bottom:* A specific example of the computational problem  $BEGINSWITHA$  mapping an ASCII string to a set of ASCII strings.

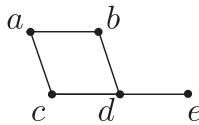


**Figure 4.8: Description of the computational problem BEGINSWITHA.** The solution sets of this problem can have multiple elements.

*solution set* for  $s$ , and any single element of  $F(s)$  is a *solution* for  $s$ . In the particular case that  $\Sigma$  is the ASCII alphabet, a computational problem  $F$  takes a single ASCII string as input, and maps it to a set of solutions, each of which is also an ASCII string—see figure 4.7.

As a very simple example, consider the problem  $BEGINSWITHA$  in figure 4.8, which searches the input for a word beginning with “a”. The solution sets of this problem can have multiple elements, as in the example of figure 4.7. The input “she ate a red apple” maps to a solution set of three elements,  $\{ "a", "ate", "apple" \}$ . A program solving this problem could return any one of the three solutions for this input.

A more interesting and realistic example is provided by the computational problem  $SHORTESTPATH$ .  $SHORTESTPATH$  was defined informally on page 4. Now we examine the formal definition, given in figure 4.10. Notice how, even in this supposedly “formal” definition, we don’t aim for complete precision and rigor. The problem input includes the description of a graph, but the “formal” definition doesn’t even state how the graph should be described as a string. Typically, the meaningful properties of a computational problem are not affected by the choice of encoding used for the input, provided we exclude certain unreasonable encodings discussed in section 11.7. Let’s just agree to use any sensible convention for graphs, such as the one in figure 4.3 (page 48). Similarly, the output requires us to describe a path in the graph using a string, without explicitly saying how to do this. Again, the convention of figure 4.3 works fine. We will also usually allow an arbitrary combination of whitespace between the main components of the input or output of a computational problem. This allows the strings to be formatted for readability, with newlines and indentation wherever that’s helpful.



**Figure 4.9:** A graph used as input to SHORTESTPATH.

### PROBLEM SHORTESTPATH

- **Input:** A graph  $G$  and two of  $G$ 's nodes  $v, w$  (the *source* and the *destination*).  $G$ ,  $v$ , and  $w$  are separated by semicolons and optional whitespace. For example, suppose we are interested in finding the shortest path from node  $a$  to node  $e$  in the graph of figure 4.9. Then the input could be “ $a, b \ a, c, b, d \ c, d, e ; a ; e$ ”.
- **Solution:** A shortest path from  $v$  to  $w$  in  $G$ , or “no” if no such path exists. In the above example, “ $a, b, d, e$ ” and “ $a, c, d, e$ ” are both valid solutions.

**Figure 4.10:** Description of the computational problem SHORTESTPATH.

Another lack of formality is that the solution for invalid inputs is not described explicitly. For SHORTESTPATH, an input is invalid if it doesn't describe a graph, source, and destination according to the agreed input format. We get around this by agreeing to the following convention:

The solution for an invalid input to any computational problem is always the string “no”.

Finally, notice that the solution set is not described in its entirety. In the particular example considered here—a shortest path from node  $a$  to node  $e$  in the graph of figure 4.9—there happen to be two equally short paths of minimal length from the source to destination. So the solution set consists of a set of two strings: “ $a, b, d, e$ ” and “ $a, c, d, e$ ”. In rigorous mathematical notation, we would write the input and output as follows:

$$\begin{aligned} \text{SHORTESTPATH}(&“a, b \ a, c \ b, d \ c, d, e ; a ; e”) \\ &= \{“a, b, d, e”, “a, c, d, e”\}. \end{aligned}$$

In practice, we will rarely write out complete solution sets, but it is important to understand that many natural problems can have multiple solutions. When no solution exists, the solution set of the problem will always be defined to be the singleton {“no”}.

### Positive and negative instances

This brings us to the concept of positive and negative instances. Any particular input to a problem is called an *instance* of the problem. Every instance is either

positive or negative. For problems defined on ASCII inputs, an instance is *negative* if its solution set is the singleton {"no"}. Otherwise the instance is *positive*. For computational problems that use non-ASCII alphabets, the special string "no" produced by negative instances can be replaced by any other fixed string on the alphabet.

Here are some examples of positive and negative instances for SHORTEST-PATH:

Instance	Type	Reason
"a, b c,d; a; c"	negative	there is no path from a to c.
"a, b c,d"	negative	the input is invalid—it doesn't specify a source and destination.
"a, b c,d; a; b"	positive	"a,b" is a solution.

You might be wondering why we insist on having the particular solution set {"no"} for negative instances. If there are really no solutions to a particular instance of a problem, wouldn't it be more sensible for the solution set to be empty? This would be more elegant for some purposes. But on balance, it's preferable for the computer programs in this book to have a concrete, easily identifiable output even when there's no solution to a given input. That's why we will require these programs to output "no", rather than producing no output at all.

Our special use of the string "no" to signal a negative instance does have one drawback. In some very unusual circumstances, you might encounter a problem that would naturally produce the string "no" as a solution to a positive instance. For example, consider the problem that takes a list of words as input and outputs all words of length 2 from the input (e.g., on input "he hit it", a solution is "he it"). What about the input "no not now"? The solution should of course be "no", but according to our convention, that would mean there were no solutions. Obviously, this is a contrived scenario that's unlikely to occur in practice, and we can always define our computational problems so that the output "no" is unambiguous. In this case, for example, we could have specified that any words in the output should be surrounded by single quotes (e.g., input "he hit it" produces output "'he' 'it'", and input "no not now" produces "'no'"—which is distinct from the negative-instance output "no").

## Notation for computational problems

Because computational problems are really functions, we will often use the symbol  $F$  to represent them (and sometimes also the symbols  $G, H, \dots$ ). Decision problems, defined below, will often be represented by  $D$ . It's particularly worth noting two symbols that will *never* represent computational problems in this book:

- $f$ : A lowercase  $f$  represents a standard mathematical function, with an arbitrary domain and codomain. This contrasts with uppercase  $F$ , whose domain is strings on an alphabet and codomain is solution sets of strings on that alphabet.

- $P$ : An uppercase  $P$  will always represent a computer *Program*, not a computational *Problem*. It's unfortunate that the words "problem" and "program"—perhaps the two most fundamental words in this book—both start with the same letter. We avoid any confusion this might cause by remembering that "problems" are really just "functions," and that's why problems are represented with the letter  $F$ .

## 4.3 CATEGORIES OF COMPUTATIONAL PROBLEMS

All of the computational problems considered in this book fit into the framework of general computational problems, as defined on page 53: that is, all problems are functions that map strings to solution sets of strings. But in practice, real-world problems to be solved by computer programs are usually stated as questions or tasks, not functions. So in this section, we examine several types of real-world computational problems: search problems, optimization problems, threshold problems, function problems, and decision problems.

### Search problems

A *predicate* is a function that returns *true* or *false*. A *search problem* is characterized by a predicate  $Q(I, S)$ , where  $I$  and  $S$  are string parameters. Given an input string  $I$ , a search problem asks, "Find a string  $S$  such that  $Q(I, S)$  is true (or return 'no' if no such  $S$  exists)." For example, we could define the search problem `FINDPATH` whose input string has the same format as `SHORTESTPATH`, encoding a graph  $G$  and two nodes  $v, w$ . A solution  $S$  is any path from  $v$  to  $w$  in  $G$ .

### Optimization problems

In this book, a *numerical function* is a function returning values that are numbers, and we usually consider only integer-valued numerical functions. An *optimization problem* is characterized by a numerical function  $V(I, S)$ , where  $I$  and  $S$  are string parameters. The function  $V$  is also called an *objective function*. Given an input string  $I$ , an optimization problem asks, "Find a string  $S$  such that  $V(I, S)$  is minimized." (Alternatively, the problem could ask that  $V$  be maximized.) For example, `SHORTESTPATH` is an optimization problem, where the ASCII input  $I$  encodes a graph  $G$  and two nodes  $v, w$ , and the function  $V(I, S)$  represents the length of the path  $S$  from  $v$  to  $w$  in  $G$ .

### Threshold problems

A *threshold problem* is characterized by a numerical function  $V(I, S)$  and a numerical threshold  $K$ . A threshold problem typically asks, given  $I$  and  $K$ , find an  $S$  such that  $V(I, S) \leq K$ . Sometimes we instead ask that  $V(I, S) = K$ . Another variant asks a decision version of the same question: Does there exist an  $S$  such that  $V(I, S) \leq K$ ? As a typical example of a threshold problem, we could define `FINDSHORTPATH` that takes as input a string encoding of  $G, v, w, K$ . A solution

### PROBLEM MULTIPLY

- **Input:** Two positive integers  $M, M'$  in decimal notation, separated by whitespace. (Example: “5 21”.)
- **Solution:** The product  $M \times M'$ , in decimal notation. (Example: On input “5 21”, the solution is “105”.)

**Figure 4.11: Description of the computational problem MULTIPLY.** This problem has unique solutions: for every input, there is exactly one element in the solution set.

is any path from  $v$  to  $w$  in  $G$  with length at most  $K$ , and “no” if no such path exists.

## Function problems

A function problem is characterized by a function  $f(I)$  that takes a single string as input and returns a single string as output. A function problem asks the obvious question: Given  $I$ , what is  $f(I)$ ? Here’s a formal definition:

**Definition of a function problem.** A *function problem* is a computational problem whose solution sets are all singletons. That is, there is a unique solution for each possible input string  $I$ .

For example, we could define the function problem SHORTESTPATHLENGTH that takes as input a string describing a graph  $G$  and two nodes  $v, w$ . The solution is the length of the shortest path from  $v$  to  $w$  in  $G$ , or “no” if no such path exists.

Function problems are very common, so let’s examine another example: MULTIPLY, defined formally in figure 4.11. Every valid input consists of two positive integers, and the unique solution is obtained by multiplying these two integers. When discussing function problems, we’ll usually abuse our terminology and notation, by ignoring the difference between “solution” and “solution set.” For example, we might write

$$\text{MULTIPLY}(“5 21”) = “105”,$$

instead of the more technically correct

$$\text{MULTIPLY}(“5 21”) = \{“105”\}.$$

## Decision problems

Decision problems play a central role in the theory of computation, so let’s start with a formal definition:

**Definition of a decision problem.** A *decision problem* is a function problem with exactly two possible solutions. For problems using the

	<b>Decision problems</b>	<b>General computational problems</b>
Advantage	more elegant for stating and proving <i>theoretical</i> results (especially impossibility and hardness theorems)	correspond to the way computers are actually used in <i>practice</i>
Disadvantage	rarely employed directly in <i>practical</i> applications	can get messy when stating and proving <i>theoretical</i> results

**Figure 4.12:** The advantages and disadvantages of using decision problems to analyze the theoretical foundations of computer science.

ASCII alphabet, the two possible solutions are always denoted “yes” and “no”. For problems using other alphabets, any other two fixed strings can be used.

For some concrete examples of decision problems, look ahead to HASPATH, HASSHORTPATH, and CHECKMULTIPLY in the right-hand column of figure 4.13 (page 60).

Decision problems are important in computational theory because it is relatively easy to state and prove theorems about them. In practice, however, people rarely use computers to solve decision problems. For example, it might be useful to have a program that solves the decision problem “Is this input file larger than 1 megabyte?” But most people would agree that it is much more useful to have a program that solves the general computational problem “How big is this input file?”

The relative merits of decision problems and general problems for the theory and practice of computer science are illustrated in figure 4.12. The contrasts highlighted in the figure present a particular challenge for this book. As you know, a primary objective of this book is to be as practical as possible, while still presenting the profound ideas at the heart of computer science (see page 9). This suggests we should deal with general computational problems as much as possible, ignoring decision problems because they are less practical. On the other hand, another objective of this book is to present the profound ideas clearly and simply. For this, decision problems provide the most elegant route. Therefore, the remainder of the book uses a hybrid approach. We will discuss and interpret general computational problems whenever possible, but will retreat into decision problems when the extra elegance and clarity warrant it.

### Converting between general and decision problems

Most general computational problems have a closely related decision problem that captures the same fundamental computational idea, and vice versa. Figure 4.13 gives examples of how to do this for a search problem, an

General version	Decision version
FINDPATH: Input is graph $G$ , source node $v$ and destination node $w$ . Solution is a path from $v$ to $w$ , or “no” if none exists.	HASPATH: Input is $G, v, w$ as before. Solution is “yes” if a path from $v$ to $w$ exists, and “no” otherwise.
SHORTESTPATH: Input is graph $G$ , source node $v$ and destination node $w$ . Solution is a shortest path from $v$ to $w$ , or “no” if none exists.	HASShortestpath: Input is $G, v, w$ as before, and also a threshold integer $K$ . Solution is “yes” if the shortest path from $v$ to $w$ has length at most $K$ , and “no” otherwise.
MULTIPLY: Input is positive integers $M, M'$ . Solution is $M \times M'$ .	CHECKMULTIPLY: Input is positive integers $M, M', K$ . Solution is “yes” if $M \times M' = K$ , and “no” otherwise.

**Figure 4.13:** Converting between general computational problems and decision problems usually involves adding extra details to the input.

optimization problem, and a function problem. For the search problem FINDPATH, instead of asking for the path itself, we ask, does the path exist? For the optimization problem SHORTESTPATH, we convert to a related threshold problem and then ask whether a solution exists: instead of asking what the shortest path from  $v$  to  $w$  is, we ask, does the shortest path from  $v$  to  $w$  have length at most  $L$ ? For the function problem MULTIPLY, we again convert to a related threshold problem: instead of asking what  $M \times M'$  is, we ask, is it true that  $M \times M' = K$ ?

The key point here is that, for most problems of practical interest, both approaches can be used to capture the interesting part of the computation. So we can usually choose whether to analyze a decision problem or a general problem, and convert between them relatively easily. Converting from a general problem to a decision problem usually involves (i) incorporating a threshold, and/or (ii) asking for the existence of a solution rather than searching for a solution. We can also consider the *trivial* conversion that converts all solutions other than “no” into “yes”, but this trivial conversion sometimes fails to capture the spirit of the problem. For example, the trivial conversion maps both FINDPATH and SHORTESTPATH to HASPATH, so any notion of “short” paths is lost in the conversion.

Finally, it’s important to note that, under certain mild conditions, an efficient method for solving a threshold problem can be transformed into an efficient method for solving the corresponding optimization problem. The basic idea is to use binary search to home in on the optimal value of the objective function. Exercise 4.16 guides you through the details of this process.

## Complement of a decision problem

Later we will need the concept of complementing (or negating) a decision problem. If  $D$  is a decision problem, then its *complement*  $\overline{D}$  returns “yes” whenever

$D$  returns “no”, and vice versa. For example, we could define NOTHASPATH as a decision problem that receives string descriptions of  $G, v, w$  as in figure 4.13. The solution is “yes” if and only if  $G$  has no path from  $v$  to  $w$ . Then NOTHASPATH is the complement of HASPATH.

## Computational problems with two input strings

Sometimes we want to consider computational problems that receive two ASCII strings as input. In mathematical notation, we would have a computational problem  $F$  that maps two ASCII strings  $(I_1, I_2)$  to the solution set  $F(I_1, I_2)$ . A simple example would be BOTHCONTAININGAGA( $I_1, I_2$ ), which takes two genetic strings  $(I_1, I_2)$  as input. The solution is “yes” if both  $I_1$  and  $I_2$  contain “GAGA” as a substring, and “no” otherwise. Clearly, we could generalize the formal definition of computational problem to account for multiple inputs, but it turns out there’s no need to do that.

This is because it’s always possible to convert a two-input problem into a single-input problem—we just need to agree on an unambiguous method of combining the two input strings into a single string. In the case of BOTHCONTAININGAGA, we could concatenate the two inputs and separate them with a space character. For example, the 2-string input (“CAGA”, “AAT”) would be encoded as the single string “CAGA AAT”. This is unambiguous, because  $I_1$  and  $I_2$  are genetic strings and therefore cannot themselves contain whitespace. But this approach would be ambiguous if  $I_1$  and  $I_2$  were permitted to contain space characters.

A more general approach that works unambiguously for any two-input problem is to encode  $(I_1, I_2)$  as a single string in the following way: first insert the length of  $I_1$ , followed by a space character; then insert  $I_1$  and  $I_2$  with no other separators. In this scheme, the 2-string input (“CAGA”, “AAT”) would be encoded as the single string “4 CAGAAAT”. We’ll denote this encoding by ESS (an abbreviation for Encode as Single String). In Python-esque notation, with “+” representing ordinary string concatenation and space representing a single space character, we have

$$\text{ESS}(I_1, I_2) = \text{len}(I_1) + \text{space} + I_1 + I_2.$$

The inverse of this function, DESS (DEcode from Single String) is defined in the obvious way:

$$\text{DESS}(I) = (I_1, I_2),$$

where  $I$  is assumed to have the form “ $n S$ ” for some integer  $n$  and string  $S$ ;  $I_1$  comprises the first  $n$  characters of  $S$ ; and  $I_2$  comprises the remaining characters of  $S$ . Python implementations of ESS and DESS are provided with the book’s `utils` module.

Thus, any two-input problem can be converted into an equivalent single-input problem. For the remainder of the book, we will often discuss two-input problems as if they meet the formal definition of a computational problem, without mentioning this technicality.

## 4.4 THE FORMAL DEFINITION OF “SOLVING” A PROBLEM

We now have all the mathematical tools to give a formal definition of what it means to “solve” a computational problem. Recall from page 53 that a computational problem  $F$  on the ASCII alphabet is a function mapping an ASCII input string  $I$  to a solution set  $F(I)$ . The set  $F(I)$  contains one or more ASCII strings that we call “solutions.” Also recall from page 24 that a Python program  $P$  takes an ASCII string  $I$  as input; the return value is either undefined, or it is an ASCII string denoted  $P(I)$ .

**Definition of “solve,” “compute,” and “decide.”** Suppose  $F$  is a computational problem on the ASCII alphabet, and  $P$  is a Python program. We say that  $P$  *solves* or *computes*  $F$  if  $P(I) \in F(I)$  for all inputs  $I$ . If  $F$  is a decision problem solved by  $P$ , we also say that  $P$  *decides*  $F$ .

In other words,  $P$  computes  $F$  if  $P(I)$  is always one of the solutions to  $F$ . Implicitly, this requires that  $P(I)$  is defined for all inputs. Note that we use the words “solve” and “compute” interchangeably. “Decide” also means exactly the same thing, but it applies only to decision problems and decision programs. Also note that the definition implicitly requires that  $P$  terminates successfully and returns an ASCII string, for all inputs (since otherwise  $P(I)$  wouldn’t be defined). Finally, note that the above definition can be generalized to computational problems on non-ASCII alphabets. We still use programs  $P$  that receive and return ASCII; we just need to agree on a fixed method of translating from the problem’s alphabet into ASCII, and vice versa. The discussion on page 50 gives more details on this.

For a first example of this formal notion of solving a problem, refer back to the discussion of SORTWORDS on pages 46–47. Figure 4.2 proposed three potential methods of solution for SORTWORDS: `pythonSort.py`, `bubbleSort.py`, and `brokenSort.py`. Recall that `brokenSort.py` works correctly on most inputs, and in fact never gives a wrong answer—but it does go into an infinite loop on some inputs. Therefore, according to the formal definition, `brokenSort.py` does not solve SORTWORDS. In contrast, `pythonSort.py` and `bubbleSort.py` both solve SORTWORDS, using quite different approaches.

To further practice the formal definition of “solve,” let’s restate the key results of chapter 3. Those results were stated in a deliberately vague manner, referring to programs that are “impossible” or “can’t exist.” It’s now clear that first we need to give a clear definition of the computational *problems* these programs are attempting to solve. This is done in figure 4.14, which formally defines the two decision problems YESONSTRING and CRASHONSTRING.

To discuss these problems properly, we need some further vocabulary:

**Definition of “computable,” “uncomputable,” “decidable,” “undecidable.”** A computational problem  $F$  is *computable* if there exists some program that computes it. Otherwise  $F$  is *uncomputable*. *Decidable* and *undecidable* mean the same as computable and uncomputable respectively, but they apply only to decision problems.

In chapter 3, we showed that no computer program can solve YESONSTRING or CRASHONSTRING. Hence, both problems are uncomputable. And because

**PROBLEM YESONSTRING**

- **Input:** Two ASCII strings: a program  $P$  and an input  $I$ .
- **Solution:** “yes” if  $P(I)$  is defined and equals “yes”; “no” otherwise.

**PROBLEM CRASHONSTRING**

- **Input:** Two ASCII strings: a Python program  $P$  and an input  $I$ .
- **Solution:** “yes” if  $P$  throws a Python exception on input  $I$ ; “no” otherwise.

It’s interesting to note that this definition of CRASHONSTRING works only for Python programs, since it depends on the specifics of when Python programs throw Python exceptions. As discussed on page 26, almost all of the definitions and results in this book work for any reasonable definition of “computer program”; CRASHONSTRING is one of the few exceptions to this rule.

**Figure 4.14:** Formal description of the two undecidable problems discussed in chapter 3.

**PROBLEM ISMEMBER $_L$** 

Given language  $L$ ,

- **Input:** String  $I$
- **Solution:** “yes” if  $I \in L$ , and “no” otherwise.

**Figure 4.15:** Description of the computational problem ISMEMBER $_L$ .

they are both decision problems, we can also call them undecidable. For later reference, we summarize this as a claim:

**Claim 4.1.** YESONSTRING and CRASHONSTRING are both undecidable.

### Computable functions

Occasionally, we will want to discuss the computation of ordinary mathematical functions, rather than computational problems. In this context, the words “compute” and “computable” have their obvious definitions. A function  $f$ , mapping strings to strings, is *computed* by program  $P$  if  $P(I) = f(I)$  for all  $I$  in the domain of  $f$ . And  $f$  is *computable* if it can be computed by some program.

## 4.5 RECOGNIZING AND DECIDING LANGUAGES

Recall from page 51 that a language is a set of strings—and in this book, the strings are typically ASCII strings. There is a close relationship between languages and decision problems. Given any language  $L$ , we can define the

Language $L$	Decision problem $\text{ISMEMBER}_L$
Prime numbers	$\text{ISP}RIME$ : Given input $M$ , is $M$ prime?
Java programs	$\text{ISJAVA}PROGRAM$ : Given input $S$ , does the Java compiler compile $S$ with no errors?
Genetic strings containing “GAGA”	$\text{CONTAINS}GAGA$ : Does the input string contain “GAGA”?
{“CACA”, “TATA”}	Solution is “yes” if input is “CACA” or “TATA”, and otherwise solution is “no”.
Strings of even length	$\text{ISEVENLENGTH}$ : Does the input have even length?
ASCII* (all ASCII strings)	$\text{YES}$ : Solution is “yes” for all inputs
$\emptyset$ (the empty language)	$\text{NO}$ : Solution is “no” for all inputs

Figure 4.16: Every language represents a decision problem, and vice versa.

decision problem  $\text{ISMEMBER}_L$ , which determines whether its input is a member of  $L$  (see figure 4.15 for a formal definition). And we can go the other way too: given any decision problem  $D$ , we can define a language  $L_D$  consisting of the positive instances of  $D$ . That is,

$$L_D = \{I \mid D(I) = \text{"yes"}\}.$$

The equivalence between languages and decision problems is demonstrated in figure 4.16. The last two lines of this table introduce two new trivial decision problems: YES (whose solution is always “yes”) and NO (whose solution is always “no”). These trivial problems correspond to the language of all strings, and the empty language, respectively. Note that the definition of complementing a decision problem (page 60) is precisely equivalent to the definition of complementing a language (page 52): given decision problem  $D$ , we have  $\overline{D} = \text{ISMEMBER}_{\overline{L}_D}$ .

Because of the equivalence between languages and decision problems, the words “decide” and “decidable” apply equally well to both. Specifically, a program  $P$  decides a language  $L$  if and only if  $P$  decides  $\text{ISMEMBER}_L$ ; and  $L$  is decidable if and only if  $\text{ISMEMBER}_L$  is decidable. All of the languages in figure 4.16 are decidable.

Have we seen any examples of undecidable languages yet? Yes we have, two of them: the languages corresponding to the undecidable problems YESONSTRING and CRASHONSTRING. There is one technicality in defining these languages

properly. Both of these decision problems require *two* strings as input—a program  $P$  and an input  $I$  for that program. Here we must remember that, formally, a computational problem takes only a single string as input. If we want to provide two strings  $(P, I)$ , we first encode them as a single string using the ESS function defined on page 61. With this in mind, we can define the language corresponding to YESONSTRING:

$$L_{\text{YESONSTRING}} = \{\text{ESS}(P, I) \mid P(I) = \text{"yes"}\}.$$

The language  $L_{\text{CRASHONSTRING}}$  is defined similarly, and both are undecidable languages. This follows immediately from the fact that the corresponding decision problems are undecidable.

## Recognizable languages

When faced with an undecidable language such as  $L_{\text{YESONSTRING}}$ , it is natural to wonder just how bad the situation is. We know that no program can always correctly determine membership of the language. But is there an easier question that we can answer instead? One possibility is to focus only on the strings that *are* in the language. Can we always obtain a correct answer now? This motivates the following definition:

**Definition of “recognize” and “recognizable.”** A program  $P$  *recognizes* a language  $L$  if

- for all  $I \in L$ ,  $P(I) = \text{"yes"}$ ; and
- for all  $I \notin L$ , either  $P(I)$  is undefined or  $P(I) = \text{"no"}$ .

A language  $L$  is *recognizable* if there exists some program that recognizes it.

We extend these definitions from languages to decision problems in the obvious way: a decision problem  $D$  is recognizable if  $L_D$  is a recognizable language.

So, what is the difference between recognizing a language and deciding it? For positive instances, there is no difference: a program that recognizes or decides a language must return “yes” for all positive instances. For negative instances, there is an important difference: a program that decides a language must return “no” for all negative instances, whereas a program that recognizes a language can, optionally, enter an infinite loop and thus fail to return a value on negative instances. Note, however, that if a recognizer does return a value for a negative instance, the value must be the correct one (i.e., “no”). In other words, a recognizer never gives a wrong answer, but it may fail to give any answer on some or all negative instances.

The program `recognizeEvenLength.py`, shown in figure 4.17, provides a silly but instructive example of this. The program successively checks whether the length of the input is equal to 0, 2, 4, 6, .... If the input length is even, the program eventually finds a match and returns “yes”. But if the input length is odd, a match will never be found and the program loops forever. Hence, this

```

1  def recognizeEvenLength(inString) :
2      i = 0
3      while True:
4          if len(inString) == i:
5              return 'yes'
6          else:
7              i = i + 2

```

**Figure 4.17:** The Python program `recognizeEvenLength.py`. This program recognizes, but does not decide, the language of even-length ASCII strings.

program recognizes, but does not decide, the language of even-length ASCII strings.

This example is, however, rather unsatisfying. Clearly, we can write a more sensible program that does decide the language of even-length ASCII strings. So the language itself is decidable. It would be much more interesting to see an example of a language that is undecidable, but still recognizable. In fact, we've already seen two such languages:  $L_{\text{YESONSTRING}}$  and  $L_{\text{CRASHONSTRING}}$ —but we don't yet have the tools to prove that they are recognizable. We will return to these examples in section 6.5 and again in section 8.8, increasing our understanding of decidability and recognizability using the new tools presented in those chapters.

It should be obvious by now that decidability is a stronger property than recognizability. We can formalize this:

**Claim 4.2.** If  $L$  is a decidable language, then  $L$  is recognizable.

*Proof of the claim.* Since  $L$  is decidable, there exists some program  $P$  that decides it. The program  $P$  can also be used to recognize  $L$ .  $\square$

## Recursive and recursively enumerable languages

Some books refer to decidable languages as *recursive* languages, and recognizable languages as *recursively enumerable* languages. This terminology comes from an equivalent theory of computation based on recursive functions, and it was the dominant terminology in the early days of computational theory. But the words “decidable” and “recognizable” make more sense for an approach based on computer programs.

## EXERCISES

**4.1** In this question, we use the conventions of figure 4.3 for converting between graphs and ASCII strings.

- (a) Let  $G$  be the undirected graph represented by “`a1,a2 a1,b1 a2,b1  
a2,b2 a2,b3 a2,a3 a3,b2 a3,b3`”. What is the shortest path from `a1` to `b3` in  $G$ ?

- (b) Is  $G$  a tree?  
 (c) Let  $H$  be the directed graph represented by

“apple,apple apple,banana banana,coconut  
 coconut,banana coconut,apple”

List all (directed) cycles present in  $H$ . For the purposes of this question, we are interested only in simple cycles (i.e., cycles that have no repeated nodes), and the start node is irrelevant (e.g.,  $x, y, z$  and  $y, z, x$  represent the same cycle).

- (d) Consider the tree  $T$  represented by “ $p, y \ y, v \ y, x \ y, z \ z, w \ z, u$ ”. We will regard  $T$  as a rooted tree with root “ $u$ ”.
- (i) Which nodes are at level 2?
  - (ii) Which nodes are leaves of the tree?

**4.2** How many symbols are in the ASCII alphabet, as it is defined in this book?

**4.3** Consider the following set  $S$  of symbols, which consists of all horizontal lines whose length is some multiple of 2 millimeters:  $S = \{_, __, ___, \dots\}$ . Is  $S$  an alphabet?

**4.4** Consider the alphabet  $\Sigma = \{C, A, G, T\}$ .

- (a) How many strings of length 0, 1, 2, and 3 can be made using  $\Sigma$ ?
- (b) How many strings of length  $n$  can be made using  $\Sigma$ ?
- (c) How many strings of length at most  $n$  can be made using  $\Sigma$ ?
- (d) Approximately how long would it take for a modern computer to iterate over all strings on  $\Sigma$  of length exactly 20? (Assume each string can be processed in 1 ns.) How about for strings of length exactly 50? Express your answers in sensible units, such as minutes, days, or years. How do your answers compare to the age of the universe?

**4.5** Assume we are using the ASCII alphabet and take  $s = “abc”$ ,  $t = “yz”$ . Determine the following strings or values:

- (a)  $sts$
- (b)  $t^2s^0t\epsilon$
- (c)  $|(s^2)^9|$
- (d)  $\epsilon^4$

**4.6** How many strings are in each of the following languages?

- (a) Binary strings of length at most 3
- (b) Binary strings containing exactly three 1s and exactly two 0s
- (c) Binary strings containing exactly three 1s
- (d)  $\emptyset$
- (e) {“Japan”, “Korea”, “China”, “Vietnam”,  $\epsilon$ }

**4.7** Take our alphabet  $\Sigma$  to be  $\{C, A, G, T\}$  and let  $L$  be the language on  $\Sigma$  consisting of all strings accepted by the program `containsGAGA.py` (see figure 2.1).

**68** • 4 What is a computational problem?

- (a) Is  $\epsilon \in L$ ?
- (b) How many strings of length 4 are in  $L$ ?
- (c) How many strings of length 6 are in  $L$ ?

**4.8** Suppose that  $L$  is defined as in the previous question, except that the ASCII alphabet is used.

- (a) Is the source code of `containsGAGA.py` an element of  $L$ ? Explain your answer.
- (b) Suppose `another.py` is a Python program that is equivalent to `containsGAGA.py`. Can we conclude that `another.py` is an element of  $L$ ? Explain your answer. (Hint: See exercise 3.3 (page 42).)

**4.9** Let  $J$  be the language of Java programs as defined in this book. That is,  $J$  is the set of ASCII strings that produce no errors when processed by some particular choice of the Java compiler. Conduct an experiment to determine whether  $\epsilon \in J$ . Briefly describe your experiment and your conclusion. If you're not familiar with Java, you can do a similar experiment for any other programming language.

**4.10** Write a Python program that decides the language of prime numbers represented in decimal as ASCII strings. That is, your program should output “yes” on inputs like “5” and “41”, but “no” on inputs like “8” and “39”. Your program need only work correctly on inputs that represent positive integers.

**4.11** Let  $\Sigma = \{C, A, G, T\}$  and let  $F$  be a function on  $\Sigma^*$  mapping a string  $s$  to the set of all substrings of  $s$  that begin and end with “A”. For each value of  $s$  below, give  $F(s)$ :

- (a) “CAGGATAACC”
- (b) “GAGA”
- (c) “CGCGATT”

**4.12** Write out the full solution set for

- (a) `SHORTESTPATH(“1, 2 1, 3 1, 4 2, 4 2, 5 3, 5 6, 7; 1 ; 5”)`
- (b) `SHORTESTPATH(“1, 2 1, 3 1, 4 2, 4 2, 5 3, 5 6, 7; 3 ; 6”)`
- (c) `SHORTESTPATH(“1, 2 1, 3 1, 4 2, 4 2, 5 3, 5 6, 7; 3”)`

**4.13** For each of the instances of `SHORTESTPATH` given below, state whether the instance is positive or negative:

- (a) “x, y x, z u, v; x; u”
- (b) “x, y x, z u, v; y; z”
- (c) “x, y x, z u, v; u; v”
- (d) “x, y, x z, u, v; y; w”

**4.14** Give the solution set for each of the problem instances below:

- (a) `SHORTESTPATH(“a, b b, c c, d d, e e, f f, a; a; d”)`
- (b) `HASSHORTPATH(“a, b b, c c, d d, e e, f f, a; a; d; 2”)`
- (c) `HASSHORTPATH(“a, b b, c c, d d, e e, f f, a; a; d; 3”)`
- (d) `MULTIPLY(“3 4”)`

- (e) `CHECKMULTIPLY("3 4 10")`
- (f) `CHECKMULTIPLY("3 4 12")`

**4.15** Let `COUNTGs` be a computational problem that takes an ASCII string  $I$  as input. The (unique) solution is the number of times “G” occurs in  $I$ , written in decimal notation.

- (a) Does `COUNTGs` have any positive instances? If so, give an example.
- (b) Does `COUNTGs` have any negative instances? If so, give an example.
- (c) Suggest a definition of `COUNTGsDECISION`, a decision problem that intuitively seems to require the same computational process as the function problem `COUNTGs`.

**4.16** This exercise leads you through a proof that, under certain reasonable conditions, we can use an efficient algorithm for a threshold problem to efficiently solve the equivalent optimization problem. So, suppose we wish to solve an optimization problem  $F$ . Specifically, assume  $F$  is a minimization problem whose objective function  $V(I, S)$  maps to nonnegative integers. We also assume the existence of an efficient program `FThresh.py` solving the equivalent threshold problem: given instance  $I$  and threshold  $K$ , `FThresh.py` returns an  $S$  such that  $V(I, S) \leq K$ , if such an  $S$  exists.

- (a) Assuming `FThresh.py` is available, write an efficient program `threshToOpt.py` that minimizes  $V$ . Specifically, `threshToOpt.py` (i) takes two parameters  $(I, U)$ , where  $I$  is an instance of  $F$  and  $U$  is an upper bound on the optimal value of  $V(I, S)$  for the given  $I$ ; and (ii) returns a solution  $S$  that minimizes  $V(I, S)$ , provided there exists some  $S$  with  $V(I, S) \leq U$ . (Hint: Use binary search, invoking `FThresh` whenever necessary. Extreme hint: If necessary, look at the provided version of `threshToOpt.py` in the book materials.)
- (b) Explain why the running time of `threshToOpt.py` is, at worst, approximately a factor of  $\log_2 U$  greater than the running time of `FThresh.py`.

**4.17** Compute the following values:

- (a) `ESS("CGGATT", "CATTA")`
- (b) `ESS(ε, "CATTA")`
- (c) `ESS("CGGATT", ε)`
- (d) `ESS("GG", ESS("CC", "GG"))`
- (e) `DESS("3 CAGGAC")`
- (f) `DESS("6 CAGGAC")`
- (g) `DESS("0 CAGGAC")`
- (h) `DESS(ESS( $s, t$ ))` for any strings  $s, t$

**4.18** Write out a formal definition of the computational problem `NOON-STRING`, which decides whether a program outputs “no” on a given input.

**4.19** Let  $Q$  be the language of ASCII strings that end in a question mark.

- (a) Is  $Q$  a decidable language? Explain your answer.
- (b) Is  $Q$  a recognizable language? Explain your answer.
- (c) Give the solution of `ISMEMBER_Q("Am I in Q?")`
- (d) Give the solution of `ISMEMBER_Q("I am in Q!")`

**4.20** Let TWOZs be the decision problem that decides whether an ASCII string contains exactly two “z” characters.

- (a) Is TWOZs a decidable problem? Explain your answer.
- (b) Is  $L_{\text{TWOZs}}$  a decidable language? Why?
- (c) Is  $L_{\text{TWOZs}}$  a recognizable language? Why?
- (d) Is “XYZ”  $\in L_{\text{TWOZs}}$ ?
- (e) Assuming the problem TWOYs is defined similarly for the character “Y”, give an example of a string that is a member of  $L_{\text{TWOZs}} \cap L_{\text{TWOYs}}$ .

**4.21** Let LN be a computational problem that takes a positive integer  $n$  as input. A solution is the natural logarithm of  $n$ ,  $\log_e n$ , expressed as a decimal rounded to an accuracy of at least 3 decimal places. Suggest a variant of LN that is a decision problem.

**4.22** Let NUMPATHS be a function problem whose input  $G, u, v$  consists of a graph  $G$ , source node  $u$ , and destination node  $v$ . The (unique) solution is the number of distinct simple paths from  $u$  to  $v$  in  $G$ . (A *simple* path is a path that does not repeat any nodes.) Suggest a variant of NUMPATHS that is a decision problem.

**4.23** Give an example of an undecidable language that is not  $L_{\text{YESONSTRING}}$  or  $L_{\text{CRASHONSTRING}}$ . (Hint: Look back to exercises 3.11–3.13.) Briefly explain why your example is undecidable.

**4.24** Let  $L$  be the language of ASCII strings that contain the character “z”. Write a Python program that recognizes  $L$ , but does not decide  $L$ .

**4.25** Let  $L$  be a decidable language.

- (a) Prove that  $\overline{L}$  is also decidable.
- (b) Prove that the union and intersection of decidable languages are also decidable.
- (c) Is  $L^*$  guaranteed to be decidable when  $L$  is decidable? Either prove this or give a counterexample.

**4.26** Let  $L_1, L_2$  be recognizable languages. Prove or give counterexamples for the following statements:

- (a)  $\overline{L_1}$  is recognizable.
- (b)  $L_1 \cup L_2$  is recognizable.
- (c)  $L_1 \cap L_2$  is recognizable.

# 5



## TURING MACHINES: THE SIMPLEST COMPUTERS

These two memoirs taken together furnish, to those who are capable of understanding the reasoning, a complete demonstration—*That the whole of the developments and operations of analysis are now capable of being executed by machinery.*

—Charles Babbage, *Passages from the Life of a Philosopher*  
(1864)

Python programs are an excellent tool for studying the theory of computer science, since they bridge the gap between theory and practice. However, unless we are very careful with our definitions, Python programs can be problematic for giving rigorous proofs of theorems. One reason for this is that, in the real world, we can't be absolutely certain about the output of a Python program. Consider the following very simple program, `yes.py`:

```
def yes(inString):
    return 'yes'
```

Obviously, the intention here is for the program to return “yes” on all inputs. But on my computer, the command

```
>>> yes(rf('bigFile.txt'))
```

causes a memory error whenever `bigFile.txt` is larger than about 500 MB. And there could be other obscure behaviors caused by limitations or bugs in the Python interpreter, the operating system, or the computer hardware. The main problem is that we are relying on a huge bundle of complex hardware and software, even to execute the simplest of programs. To produce completely rigorous mathematical proofs about computation, we need to study a much more simple, abstract, and fundamental model of computation: the *Turing machine*.

The concept we now call a Turing machine was introduced by Alan Turing in 1936, and it is still widely used as a mathematical model for computation. Our high-level strategy will be to show that Turing machines and Python programs are equivalent, in terms of what computational problems they can solve. After that,

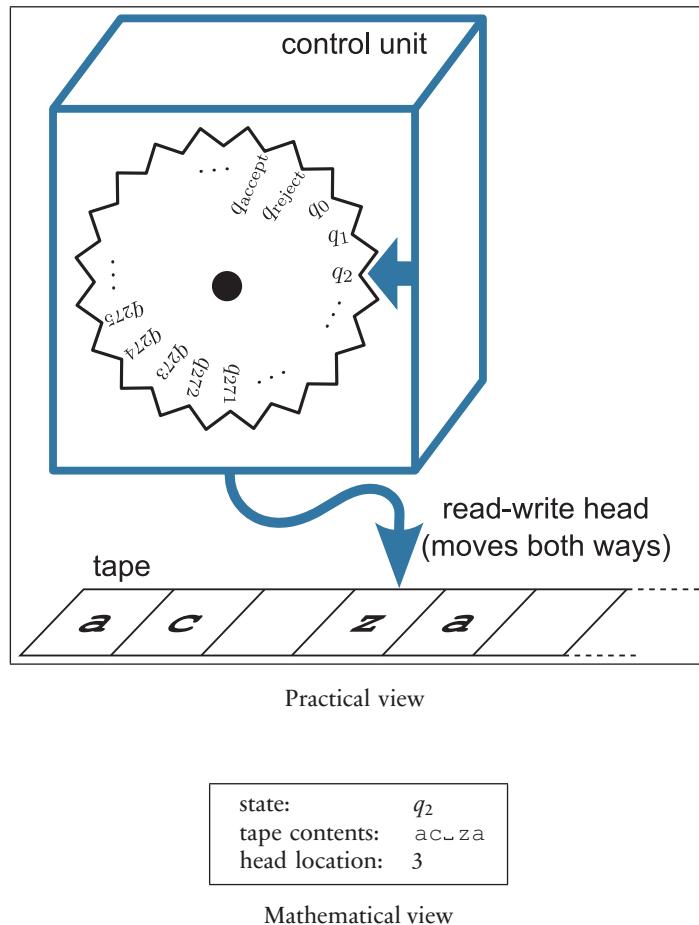


Figure 5.1: Two views of a Turing machine.

we can continue using Python programs as our primary computational model, secure in the knowledge that all arguments could—if desired—be translated into the completely abstract, mathematical world of Turing machines.

## 5.1 DEFINITION OF A TURING MACHINE

Before trying to understand the definition of a Turing machine, take a look at figure 5.1. Although Turing machines are completely abstract, it's helpful for us humans to maintain two distinct views of them: (i) a practical view, as a machine that could be constructed from physical materials and (ii) a mathematical view, as a collection of mathematical objects with no physical counterparts. Figure 5.1 shows these two views. In the practical view, we see the three main physical pieces of the Turing machine: the *control unit*, the *read-write head*, and the *tape*. The tape contains *cells*, and each cell can store a single symbol. The possible symbols include a special *blank* symbol, which we'll usually write as “ $\underline{\phantom{x}}$ ”. The read-write

head can move up and down the tape, informing the control unit of what symbols it sees. The symbol currently under the head is called the *scanned symbol*. The head can erase the scanned symbol and write a new one in its place. The control unit issues instructions to the read-write head, and switches between the various states labeled on the dial at the front of the control unit. Note that this dial is for display only. We can use the dial to observe the current state of the control unit, but we can't reach in and turn the dial ourselves—it moves exactly as instructed by the control unit.

The mathematical view of this Turing machine is shown in the bottom panel of figure 5.1: the machine is in state  $q_2$ , the content of the tape is “ac $\_\_$ za” followed by infinitely many blanks, and the location of the head is cell 3 (assuming we index the tape cells starting with 0). A complete description of a Turing machine at a particular moment in time is called a *configuration*. A configuration must specify the state, the tape contents, and the head position. Our notation for configurations will specify the state and tape contents explicitly, and add a box around the scanned symbol to represent the head position. For example, the machine in figure 5.1 has configuration  $q_2 : a \, c \, \boxed{z} \, a$

Now we're ready for a more formal, mathematical definition a Turing machine. A Turing machine is composed of five separate mathematical objects: two sets and three functions. First, we describe the two sets in the definition of a Turing machine:

- **Alphabet.** The alphabet  $\Sigma$  is a finite set of *symbols*, which must include a special symbol called the *blank symbol*. See page 49 for a more detailed discussion of alphabets. The Turing machines in this book will almost always use the 128-character ASCII alphabet, but augmented with an additional blank symbol. Thus, there will typically be 129 symbols in  $\Sigma$ . As mentioned above, the blank symbol will be denoted “ $\_\_$ ”. Note that the space character and blank symbol are distinct. For example, “CAG AG” and “CAG $\_\_$  AG” are different strings.
- **State set.** The state set  $Q$  is a finite set of *states*, which must include a special state  $q_0$ , called the *start state*. The state set  $Q$  must also include one or more of the three special *halting states*: the *accept state*  $q_{\text{accept}}$ , the *reject state*  $q_{\text{reject}}$ , and the *halt state*  $q_{\text{halt}}$ . The remaining states in  $Q$  will be denoted  $q_1, q_2, \dots$  (The letters  $Q$  and  $q$  do not appear to stand for anything meaningful. Alan Turing used  $q_i$  as the symbol for states in his original paper, and we still use that notation today.)

The three functions in the definition of a Turing machine each take as inputs the current state  $q$  and the scanned symbol  $x$ :

- **New state function,**  $q' = \text{NewState}(q, x)$ . Output is the new state  $q'$  that the Turing machine will transition into. It is possible that  $q' = q$ , in which case the machine remains in the same state as previously.
- **New symbol function,**  $x' = \text{NewSymbol}(q, x)$ . Output is the new symbol  $x'$  that the head writes in the current tape cell. Of course, it's possible to have  $x' = x$ , in which case the symbol on the tape remains unchanged.
- **Direction function,**  $d' = \text{Direction}(q, x)$ . Output  $d'$  is either Left, Right, or Stay, depending on whether the head should move left, right, or stay where

it is. If the head is already at the left end of the tape, and it is commanded to move left, then it stays where it is instead.

Collectively, these three functions are called *transition functions*, since they specify how the Turing machine transitions from one configuration to another. Often, it's more convenient to think of the three transition functions as a single function of  $(q, x)$  that returns a 3-tuple  $(q', x', d')$ . This combined function is called the transition function of the Turing machine.

Let's now formalize our definition of Turing machines:

**Definition of a Turing machine.** A *Turing machine* consists of

- an alphabet  $\Sigma$  of symbols, as defined above, including a blank symbol;
- a state set  $Q$ , as defined above, including the start state  $q_0$ , and at least one of the halting states  $q_{\text{accept}}$ ,  $q_{\text{reject}}$ ,  $q_{\text{halt}}$ ;
- a transition function mapping  $(q, x)$ —where  $q$  is a state and  $x$  is a symbol—to the triple  $(q', x', d')$ , where  $q'$  is the new state,  $x'$  is the new symbol, and  $d'$  is a direction (one of Left, Right, or Stay). The transition function is often denoted by the Greek letter  $\delta$  (delta), so we have

$$\delta(q, x) = (q', x', d').$$

The definition may seem rather detailed and abstract, but the actual operation of a Turing machine is simple and intuitive. The machine begins a computation in its start state  $q_0$ , with some finite sequence of non-blank symbols—the *input*—already written on the tape, followed by infinitely many blanks. The read-write head is initially at position zero. The machine then applies the transition function over and over again (typically writing some new symbols on the tape and moving the head around) until it reaches one of the halting states  $q_{\text{accept}}$ ,  $q_{\text{reject}}$ , or  $q_{\text{halt}}$ . At this point, the machine halts, and the sequence of symbols left on the tape (up to the first blank) is defined to be the *output* of the computation. In addition, if the machine halted in the accepting state  $q_{\text{accept}}$ , we say the machine *accepted* the input. If the machine halted in the rejecting state  $q_{\text{reject}}$ , we say the machine *rejected* the input.

As our first example, let's define a Turing machine called `lastTtoA` that works on genetic strings, so its alphabet is {C, G, A, T,  $\underline{\phantom{x}}$ }. The machine will output an exact copy of the input, except that the last T of the input will be mutated to an A. (For simplicity, we will assume for now that the input is guaranteed to contain at least one T.) It turns out we need three states for this machine, so the state set is  $Q = \{q_0, q_1, q_{\text{halt}}\}$ .

The transition functions for the `lastTtoA` machine are defined in figure 5.2. We don't bother giving transition functions for the state  $q_{\text{halt}}$ , since the machine is defined to halt on entering that state. By studying the transition functions carefully, you can probably determine how this machine works. Described in plain English, it proceeds as follows:

1. While the scanned symbol is not a blank, move right (and remain in  $q_0$ ).
2. When a blank is found, transition into  $q_1$ .

State $q$	Symbol $x$	NewState( $q, x$ )	NewSymbol( $q, x$ )	Direction( $q, x$ )
$q_0$	C	$q_0$	C	Right
	G	$q_0$	G	Right
	A	$q_0$	A	Right
	T	$q_0$	T	Right
	$\sqcup$	$q_1$	$\sqcup$	Left
$q_1$	C	$q_1$	C	Left
	G	$q_1$	G	Left
	A	$q_1$	A	Left
	T	$q_{\text{halt}}$	A	Stay
	$\sqcup$	$q_1$	$\sqcup$	Left

**Figure 5.2:** Transition functions for Turing machine `lastTtoA`, which changes the last T of a genetic string to an A.

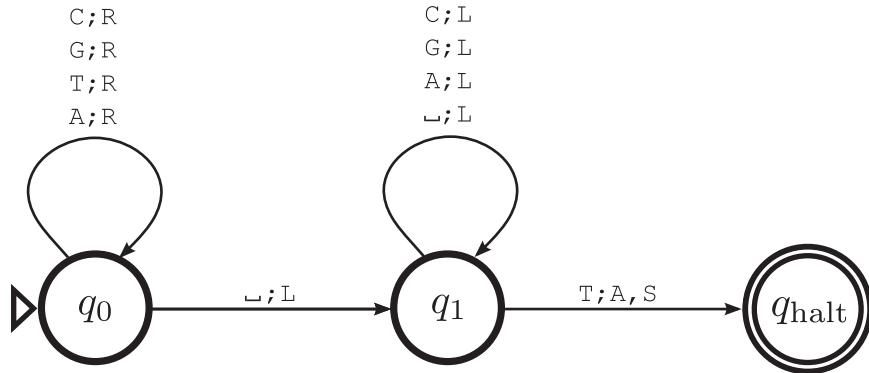
3. While the scanned symbol is not a T, move left (and remain in  $q_1$ ).
4. When a T is found, replace that T with an A, and enter  $q_{\text{halt}}$ . (This halts the machine.)

Even for this particularly simple `lastTtoA` machine, the task of translating the table of transition functions (figure 5.2) into the above plain English description is difficult, tedious, and error prone. For this reason, we will avoid tabulating the transition functions explicitly from now on. For any given Turing machine, it is always possible to write out a complete table of the transition functions. But we will use more intuitive descriptions of Turing machines. For example, Turing machines can be described using a *state diagram* like the one shown in figure 5.3. The states of the machine are represented by circles. Arrows between the states indicate transitions, and the labels on these arrows give details about the transition. Specifically, these labels consist of the scanned symbol followed by a semicolon, then the new symbol to be written (if any) followed by a comma, then the direction to move (using the obvious abbreviations L, R, and S). For example, the label “C; R” means “if the scanned symbol is a C, follow this arrow and move the head right.” Another example from figure 5.3 is “T; A, S”, which means “if the scanned symbol is a T, replace it with an A, follow this arrow, and leave the head where it is.”

For extra readability, the start state is highlighted with a small triangle, and the states corresponding to successful terminations ( $q_{\text{halt}}$  and  $q_{\text{accept}}$ ) are highlighted with a double circle. The  $q_{\text{reject}}$  state is not highlighted.

We’re ready to do a computation on this Turing machine now. Suppose the input is CTCGTA. Then the initial configuration of the machine is  $q_0 : \boxed{C} T C G T A$ . (Here we are using the configuration notation described on page 73.) The entire computation is shown in figure 5.4. Note that we generally don’t bother showing any blanks after the last non-blank symbol on the tape. But when the head goes into that region, as in the seventh step of the computation in figure 5.4, we can explicitly show the blanks up to and including the head location.

Reading off the last line of figure 5.4, we see the output of the computation is “CTCGAA”. Therefore, the `lastTtoA` Turing machine has indeed transformed the input by changing the last T to an A.

Figure 5.3: State diagram for the `lastTtoA` Turing machine.

$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_0:$	C	T	C	G	T	A
$q_1:$	C	T	C	G	T	A
$q_1:$	C	T	C	G	T	A
$q_{\text{halt}}:$	C	T	C	G	A	A

Figure 5.4: A complete computation of the `lastTtoA` machine.

## Halting and looping

To keep things simple, we assumed the input for `lastTtoA` contains at least one T. What happens if this assumption is violated? Try it out for yourself on an input such as “CGGA”. It should be clear that the machine gets stuck in  $q_1$  and runs forever. (Recall that if the head is commanded to move off the left end of the tape, it remains at cell 0. This behavior is built into our definition of a Turing machine.) If a Turing machine runs forever, we say that it *loops*. Otherwise we say it *halts* or *terminates*. By definition, a machine that halts either accepts (in  $q_{\text{accept}}$ ), rejects (in  $q_{\text{reject}}$ ), or halts without accepting or rejecting (in  $q_{\text{halt}}$ ). Note that the looping behavior in the example above is particularly simple, since the machine stays in a single state forever. Much more complex kinds of looping behavior are possible, including chaotic “loops” that don’t repeat in a regular way. Despite this, any kind of nonterminating behavior is still referred to as a loop. So, with a given input, every Turing machine either halts or loops. If it halts, it might accept, reject, or halt without accepting or rejecting.

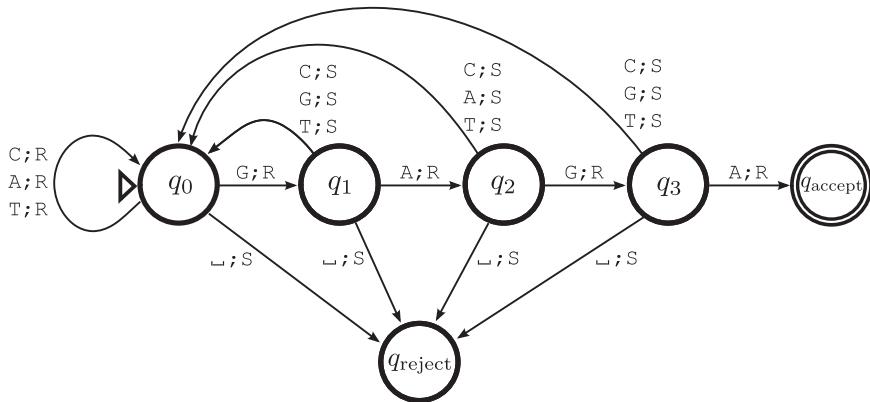


Figure 5.5: State diagram for the containsGAGA Turing machine.

### Acceptors and transducers

We have defined Turing machines in a way that lets them do two different tasks. Specifically, suppose we are given a Turing machine  $M$  and an input string  $I$ , and assume  $M$  halts on input  $I$ . Then  $M$  provides

1. an *output*,  $M(I)$ —the string of symbols (up to but not including the first blank) left on the tape at the end of the computation, and
2. (optionally) an accept/reject *decision*, determined by the state in which  $M$  halted ( $q_{\text{accept}}$  or  $q_{\text{reject}}$ ).

Sometimes, it's convenient to think of a Turing machine as doing only one of these two tasks. If we are interested only in the output, the machine is called a *transducer*. If we are interested only in the accept/reject decision, the machine is called an *accepter*. In reality, Turing machines that halt in  $q_{\text{accept}}$  or  $q_{\text{reject}}$  can do both tasks, but it can help to focus on only one of them. Two transducers  $M, M'$  are *equivalent* if  $M(I) = M'(I)$  for all  $I$ ; accepters are *equivalent* if they produce the same decision for all  $I$ .

The machine `lastTtoA` is an obvious example of a transducer. This machine contains a  $q_{\text{halt}}$  state, and does not contain  $q_{\text{accept}}$  or  $q_{\text{reject}}$  states. Thus, it's a transducer, converting inputs such as GTTG into outputs such as GTTAG.

As an example of an accepter, consider the Turing machine `containsGAGA` shown in figure 5.5. This machine never alters the input, so there's not much point in regarding it as a transducer. On the other hand, the machine accepts strings containing GAGA and rejects all others, so it is useful to think of this machine as an accepter. More precisely, `containsGAGA` accepts the language of genetic strings containing GAGA.

Once we have established the equivalence between Turing machines and Python programs (coming up, in section 5.4), this distinction between accepters and transducers might seem a little awkward. The problem is that Python programs, as we have defined them, map strings to strings—so they are *always* transducers. On page 25, we surmounted this problem by declaring that the special return value “yes” corresponds to *accept*, and “no” corresponds to

*reject*. If desired, we could do the same thing with Turing machines. That is, we could arrange that whenever a Turing machine enters  $q_{\text{accept}}$ , it erases the tape contents, prints “yes”, then halts. Similarly, on entering  $q_{\text{reject}}$ , it could erase the tape contents, print “no”, then halt. By adopting this convention, we could establish a perfect correspondence between Turing machines and Python programs. However, it will usually be preferable to think of our Turing machines halting immediately on entering  $q_{\text{accept}}$  or  $q_{\text{reject}}$ .

## Abbreviated notation for state diagrams

We can make our Turing machine state diagrams simpler by using the following abbreviations:

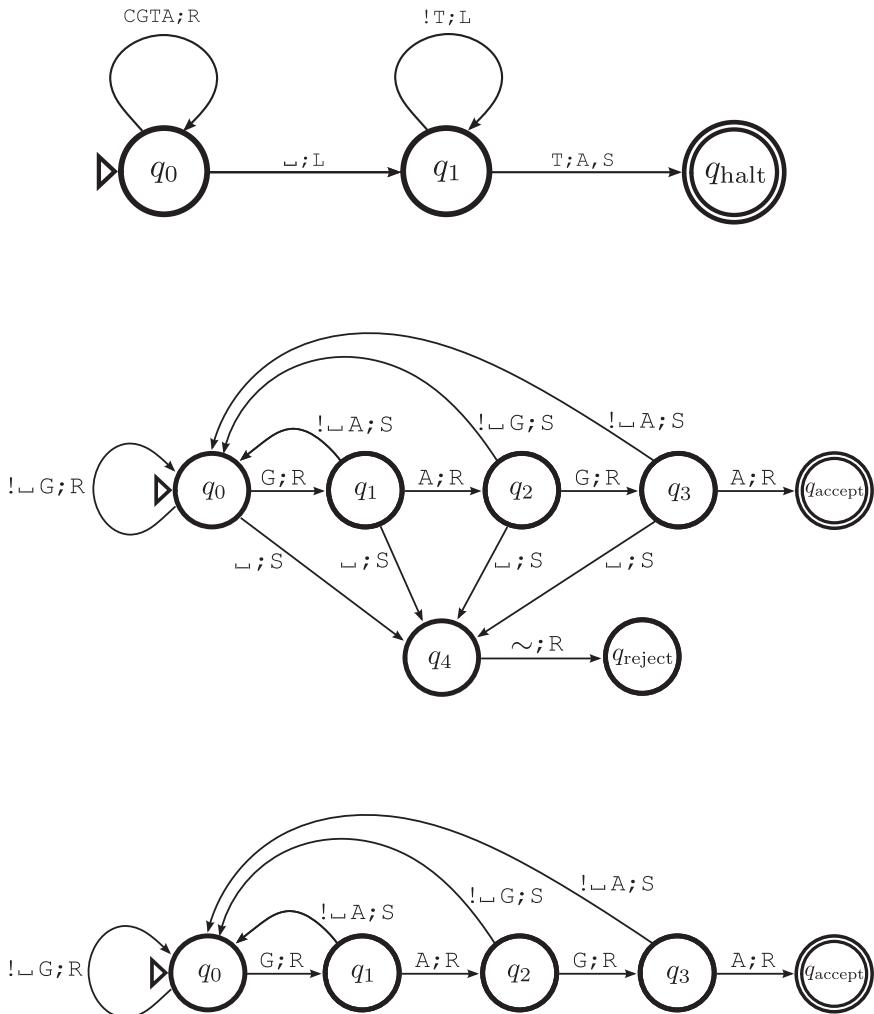
- When there are several possible scanned symbols that produce the same action, these can all be listed before the semicolon. Example: “CAG; T, L” means “if the scanned symbol is C, A, or G, replace it with T and move left.”
- To specify an action that applies to all scanned symbols *other* than those listed, use an exclamation point (“!”). Example: “!AT; R” means “if the scanned symbol is neither A nor T, move right.”
- To specify an action that applies to all symbols, use a tilde (“~”). Example: “~ ; R” means “move right, regardless of the scanned symbol.”
- Optionally, omit transitions to  $q_{\text{reject}}$ . We can assume that whenever a transition is unspecified, the machine halts and rejects by entering  $q_{\text{reject}}$ . The  $q_{\text{reject}}$  state itself can thus be omitted from the diagram.

Examples of these abbreviations are shown in figure 5.6, which shows abbreviated versions of `lastTtoA` and `containsGAGA`.

## Creating your own Turing machines

The software package JFLAP, developed by researchers at Duke University, provides a good way to experiment with your own Turing machines. You can download the software from <http://www.jflap.org>. JFLAP is largely self-explanatory, but here are a few points that will make it easier to use:

- JFLAP Turing machines have a *two-way infinite tape*—that is, the machine starts with a tape that extends infinitely in both directions. In contrast, our definition of a Turing machine uses a *one-way infinite tape*, which has a starting point at cell 0, and extends infinitely in only one direction. We will soon discover that the two definitions are equivalent in terms of what the Turing machines can compute (see page 88).
- In JFLAP, you must specify the value of the symbol to be written at each step, even if you want to leave the symbol unchanged. However, you can use the special character “~” to achieve this. So instead of “a,b;R” you can use “a,b;~,R”.
- JFLAP uses the terminology *final state*, instead of *accept state* or *halt state*. In JFLAP, machines can have multiple final states. The reject state is represented implicitly, as in the bottom panel of figure 5.6.

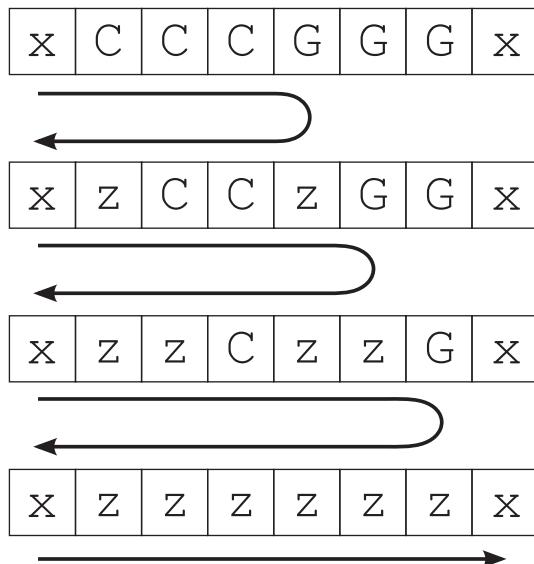


**Figure 5.6: State diagrams using abbreviated notation.** Top: `lastTtoA`, equivalent to figure 5.3. Middle: `containsGAGA`, equivalent to figure 5.5. The move labeled “ $\sim; R$ ” (from  $q_4$  to  $q_{\text{reject}}$ ) is unnecessary, but it demonstrates the use of the “ $\sim$ ” abbreviation. Bottom: `containsGAGA` again, this time demonstrating that some or all transitions to  $q_{\text{reject}}$  can be omitted. If a transition is omitted, we assume it points to  $q_{\text{reject}}$ .

Before reading on, it would be a good idea to get some hands-on experience of Turing machines. For example, use JFLAP to implement a Turing machine that changes every “C” to a “G”, and every “G” to a “C”. Test your machine on multiple inputs.

## 5.2 SOME NONTRIVIAL TURING MACHINES

The machines we’ve seen so far—`lastTtoA` and `containsGAGA`—are very simple, each with only a handful of states and transitions. To get a more realistic



**Figure 5.7: An example of the `moreCsThanGs` machine in operation.** The arrows show the motion of the Turing machine’s head.

feeling for what Turing machines can accomplish, let’s look at some more complex examples.

### The `moreCsThanGs` machine

Our first example, `moreCsThanGs`, is an accepter that takes a genetic string as input. If the input contains more C characters than G characters, the string is accepted; otherwise it is rejected. To simplify processing, the start and end of the genetic string are marked by x characters. For example, the strings `xCCGx`, `xCx`, and `xGTGCCATCx` are all inputs that would be accepted, whereas `xCGGx`, `xGCx`, and `xGGTGCATCx` would be rejected.

The first instinct of any programmer, when tackling this problem, would be to count the number of C’s, count the number of G’s, and then compare the results. This method can indeed be implemented on a Turing machine. But since Turing machines don’t have any built-in method of counting things, this approach is unnecessarily complex. Instead, we can use a clever trick: we will scan through the string, erasing one C for every G, and vice versa. If there are any C characters left over when all the G’s have been erased, the string is accepted.

An example of this approach is shown in figure 5.7. Here, each C and matching G is replaced with a z, and the head returns all the way to the left each time a match is found. A more precise description of this algorithm is given in figure 5.8, and the machine implementing it is in figure 5.9. It’s not too important to understand the details of this algorithm or the machine. If you’re comfortable with the idea that this task can be achieved using a Turing machine, feel free to skip over the details of figures 5.8 and 5.9.

Start at the left end of the tape. Scan characters one by one, moving to the right. There are three possibilities, depending on whether we first see a G, C, or x:

1. A G is encountered:
  - (a) Replace the G with a z.
  - (b) Continue scanning to the right. There are two possibilities, depending on whether or not we find a C before the end of the input:
    - i. If a C is encountered, replace the C with a z. An equal number of G's and C's have been replaced so far, so move the head to the left end of the tape, and restart the algorithm.
    - ii. If no C is encountered, we have a “leftover” G, so *reject*.
2. A C is encountered:
  - (a) Replace the C with a z.
  - (b) Continue scanning to the right. There are two possibilities, depending on whether or not we find a G before the end of the input:
    - i. If a G is encountered, replace the G with a z. An equal number of G's and C's have been replaced so far, so move the head to the left end of the tape, and restart the algorithm.
    - ii. If no G is encountered, we have a “leftover” C, so *accept*.
3. No C or G is encountered before the x at the end of the input: The number of C's replaced equals the number of G's replaced, and there are none left over, so *reject*.

**Figure 5.8:** The algorithm implemented by Turing machine `moreCsThanGs` (see figure 5.9).

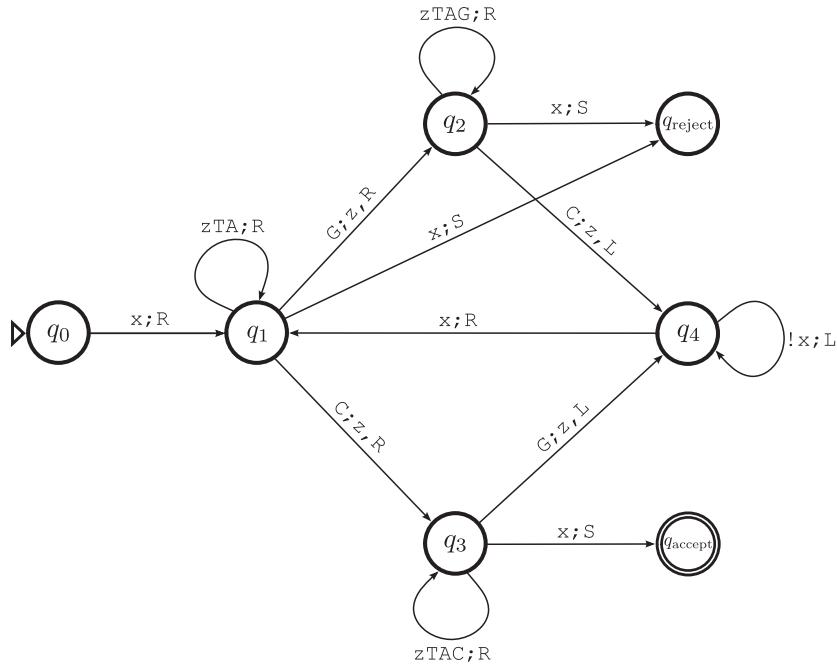
### The `countCs` machine

As our next example of a nontrivial Turing machine, we will construct a machine that counts the number of C's in a genetic string. As before, we will use “x” characters on either side of the input string to ease processing, and we will do the same thing with the output, which will be an integer in binary notation surrounded by “x” characters. For example, input xCCCCCx will produce output x101x, because there are five C's in the input and 101 is 5 in binary. Additional examples are

- input “xCgtacx” produces output “x10x”;
- input “xAGTTGx” produces output “xx”.

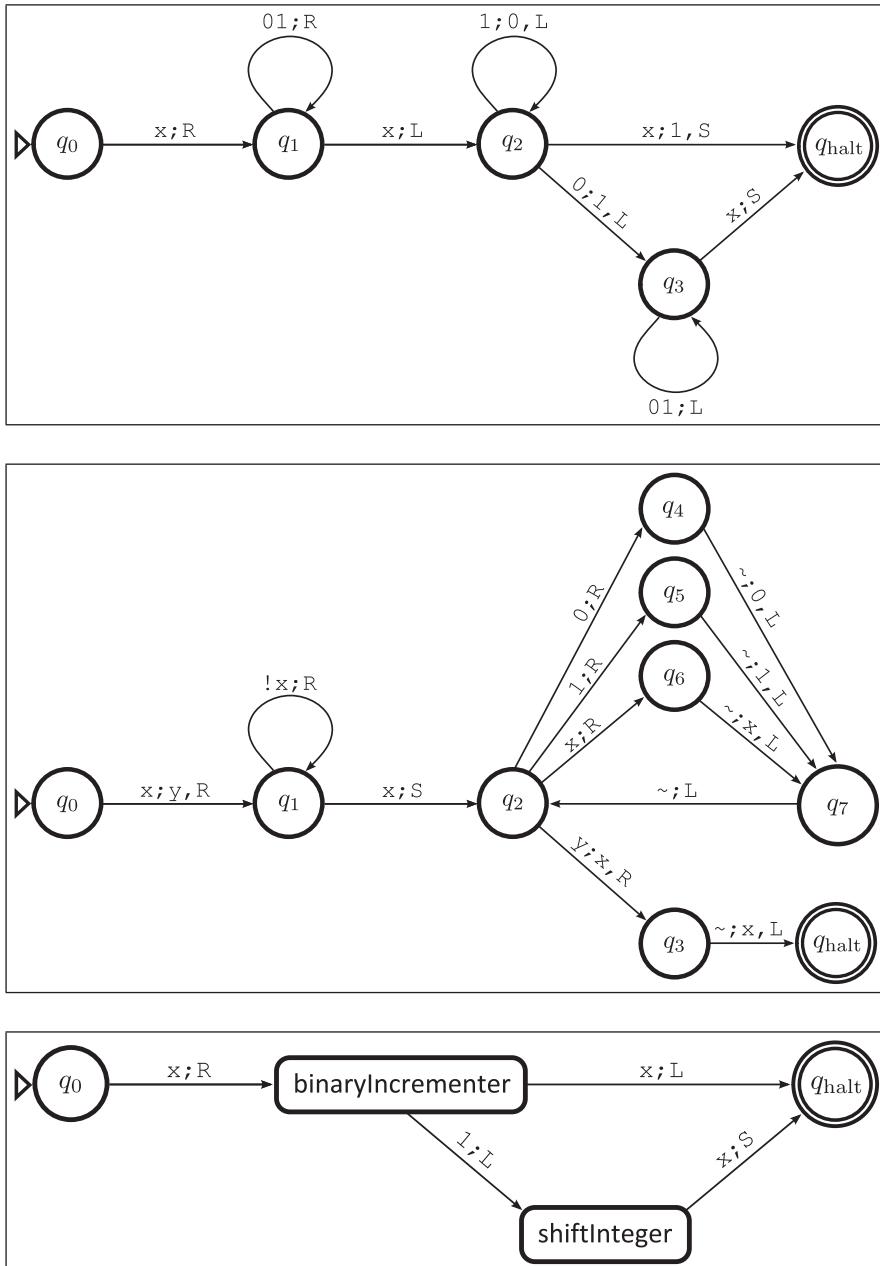
This final example demonstrates our convention that “xx” represents zero.

To build this machine, we will use the fact that large, complex Turing machines can be built out of smaller, simpler blocks that are themselves

Figure 5.9: The Turing machine `moreCsThanGs`.

Turing machines. Precisely how to do this will be demonstrated soon, but first let's create one of the simple building blocks that will be needed later. This `binaryIncrementer` machine takes a binary integer such as `x101x` as input, and increments it by 1 (so `x101x` becomes `x110x`, for example). If the integer overflows, the leading “`x`” is replaced by “`1`”. For example, `x111x` becomes `1000x`. Our `binaryIncrementer` machine also returns the head to the start of the output before halting. Of course, this does not affect the output, but it will make it easier to plug `binaryIncrementer` into a larger machine as a building block. Creating this machine for yourself is an extremely valuable exercise. Construct your own version of `binaryIncrementer` before looking at the top panel of figure 5.10, which gives one possible approach.

The `binaryIncrementer` machine of figure 5.10 has an interesting property that we haven't seen before: it employs both a  $q_{\text{halt}}$  state and an implicit  $q_{\text{reject}}$  state. The implicit  $q_{\text{reject}}$  state is used in various ways. For example, if the first scanned symbol is not an `x`, the machine immediately enters an implicit  $q_{\text{reject}}$  state and halts. The same thing happens if the machine ever encounters a blank. But for sensible inputs, the machine avoids these rejections and eventually reaches  $q_{\text{halt}}$ . Thus, `binaryIncrementer` functions mostly as a transducer, since we use it to increment binary integers. But the machine also functions partially as an accepter (or should we call it a “rejecter”?"). Specifically, `binaryIncrementer` rejects input strings that are not correctly formatted according to our convention (i.e., with leading and trailing `x`'s). Several of the Turing machines presented later—including the next one—have this dual transducer/rejecter role.



**Figure 5.10:** Top: `binaryIncrementer`. Middle: `shiftInteger`. Bottom: `incrementWithOverflow`, which uses the two machines above it as building blocks.

Now let's construct a Turing machine `shiftInteger` that shifts an integer one cell to the right, filling in the gap on the left with an `x`. For example, `x101x` becomes `xx101x`. A machine to do this is shown in the middle panel of figure 5.10. The method is fairly straightforward. The location of the initial `x`

is recorded by changing it to a  $y$ . The head then seeks the  $x$  at the end of the input, and works its way back to the left, shifting each symbol one cell to the right as it does so. When the  $y$ -marker is detected, it is changed back into  $x$ . Then the extra required  $x$  is inserted and the machine halts with the head over the initial  $x$ .

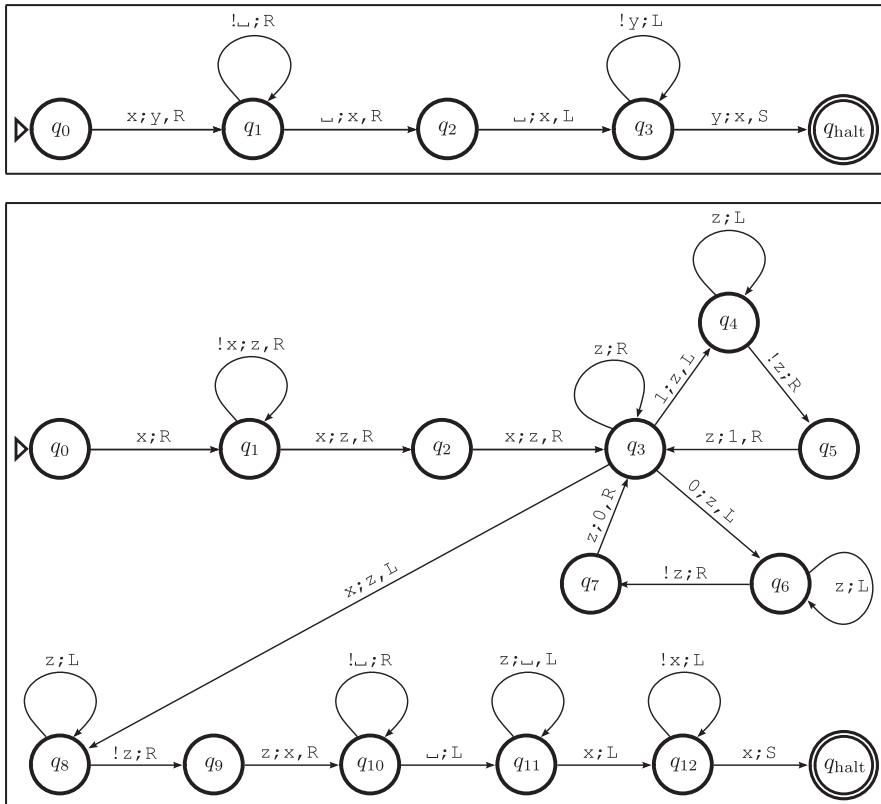
Next we can combine `binaryIncrementer` and `shiftInteger` into a larger machine that increments binary integers and copes with overflow correctly. This is achieved most easily by insisting on an extra  $x$  at the start of the input. So  $xx101x$  will become  $xx110x$ , and  $xx111x$  will overflow correctly to become  $xx1000x$ . With this convention, the two machines can be joined together to make `incrementWithOverflow`, as in the bottom panel of figure 5.10. The technique for using building block machines is obvious, but we describe it here for concreteness:

- A transition into a building block goes to that block's start state. Example: The " $x; R$ " transition out of `incrementWithOverflow`'s  $q_0$  enters `binaryIncrementer`'s  $q_0$ . The head may or may not be at the start of the tape when this occurs; computation continues from the current location of the head, based on the current scanned symbol.
- If and when a block reaches its  $q_{\text{accept}}$  or  $q_{\text{halt}}$  state, it follows the transition back out into the enclosing machine. Example: When the `binaryIncrementer` block halts without rejecting, the scanned symbol is always an  $x$  or a 1. If it's an  $x$ , `incrementWithOverflow` transitions to  $q_{\text{halt}}$ . If it's a 1, `incrementWithOverflow` transitions to the `shiftInteger` block.
- If a building block ever rejects, the entire machine halts and rejects. Example: If the input integer is not formatted correctly (e.g., doesn't begin and end with  $x$ ), the `binaryIncrementer` block rejects.

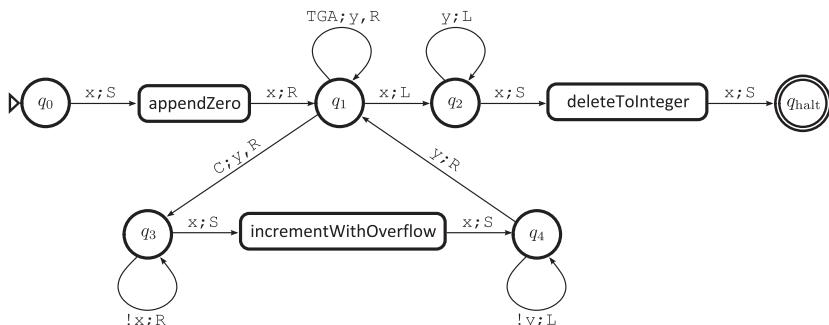
The only subtle point about `incrementWithOverflow` is the transition out of `binaryIncrementer`. By examining `binaryIncrementer` carefully, we can see that an overflow occurred if and only if the scanned symbol is a 1 when `binaryIncrementer` halts. But if there's an overflow, the integer needs to be shifted right by a cell. That's why `incrementWithOverflow` transitions to `shiftInteger` in this case, and otherwise halts immediately.

The general techniques for building our `countCs` machine should now be clear, so we will describe the remaining steps in a little less detail. It turns out we need two more blocks, which are shown in figure 5.11. The block `appendZero` appends the string "xx" (our representation of zero) to the input, so that the counting of "C" characters can get started with a binary counter initialized to zero. The block `deleteToInteger` is used when counting is complete, to delete everything up to the start of the binary counter, leaving only the desired output.

Finally, we can piece everything together, and build the `countCs` Turing machine. This is shown in figure 5.12. The basic strategy is to traverse the genetic string input from left to right, converting each occurrence of "C", "G", "A", or "T" to a "y". In addition, the counter is incremented every time a "C" is encountered. If you're interested, you can check, for example, that this machine really does transform the input  $xACACx$  into  $x10x$ .



**Figure 5.11:** Two additional blocks needed for the `countCs` machine. *Top:* `appendZero`. *Bottom:* `deleteToInteger`.



**Figure 5.12:** The `countCs` machine, including three building blocks from figures 5.10 and 5.11.

### Important lessons from the `countCs` example

What have we learned from the `countCs` example? First, the final `countCs` Turing machine of figure 5.12 really is a Turing machine according to the original

definition. The alphabet for this machine consists of 10 symbols:

$$\{C, G, A, T, x, y, z, 1, 0, \_.\}$$

The depiction of figure 5.12 uses various abbreviations and blocks, but these could all be expanded to produce a complete, unabbreviated state diagram if desired; the complete diagram would have about 40 states. Moreover, the complete diagram could be converted into a table of transition functions as in figure 5.2. This table would have about 400 rows (40 states multiplied by 10 symbols).

The second thing we have learned is that it is tedious, but not difficult, to implement basic operations such as copying, shifting, and deleting sections of the tape, and incrementing binary integers. From here, we could quickly build up a repertoire of other functionality if desired. For example, adding two integers can be done by incrementing one of them an appropriate number of times, so we can easily create an addition block. Multiplication can be implemented by repeated addition, so a multiplication block can also be created. Perhaps it already seems reasonable to you that all other common mathematical functions can be built up from these basic operations. In fact, this is precisely the path that Alan Turing took in his 1936 “On computable numbers” paper: in less than three pages, he proved that virtually all the numeric functions used by mathematicians are computable by Turing machines. However, we will not follow him down that path. We are more interested in connecting Turing machines to computers themselves, rather than to functions or numbers. Therefore, our next task is to see how the simplest computer of all—the Turing machine—can simulate a modern computer, in all its glory.

### 5.3 FROM SINGLE-TAPE TURING MACHINES TO MULTI-TAPE TURING MACHINES

Our objective now is to understand why Turing machines can solve the same set of problems as Python programs. In this part of the book, we won’t always give rigorous proofs. But we will give plausible arguments for every step in establishing the equivalence of Turing machines, Python programs, and all other known computational devices.

The concept underlying each one of these steps will be a *simulation*. In practice, it is rather common for different computational models or devices to simulate each other. For example, the Java virtual machine (JVM) is an abstract model capable of running any Java program. And the Linux operating system (used on many devices, including desktop computers) is capable of executing the JVM. Therefore, Linux devices can run any Java program. This is achieved through a *chain of simulations*: Linux simulates (or more precisely, executes) the JVM, which in turn simulates (or more precisely, executes) the desired Java program. Diagrammatically, we have

$$\text{Linux} \rightarrow \text{JVM} \rightarrow \text{Java},$$

where  $A \rightarrow B$  means  $A$  can simulate/emulate/execute  $B$ . Note that the words “simulate,” “emulate,” and “execute” have distinct technical meanings. But these

distinctions are not important for us. For simplicity, we will employ the single word “simulate” to mean “simulate/emulate/execute” from now on.

We will be using the chain-of-simulations approach to show that Turing machines (TMs) can simulate Python programs, using the following chain of simulations:

```
one-tape TM → multi-tape, single-head TM
    → multi-tape, multi-head TM
    → random-access TM
    → a real, modern computer
    → Python program.
```

We will also close the loop, by pointing out that Python programs can simulate Turing machines, and therefore all the above models are equivalent in terms of which problems they can solve. The first step in the chain is to consider a Turing machine that has two tapes.

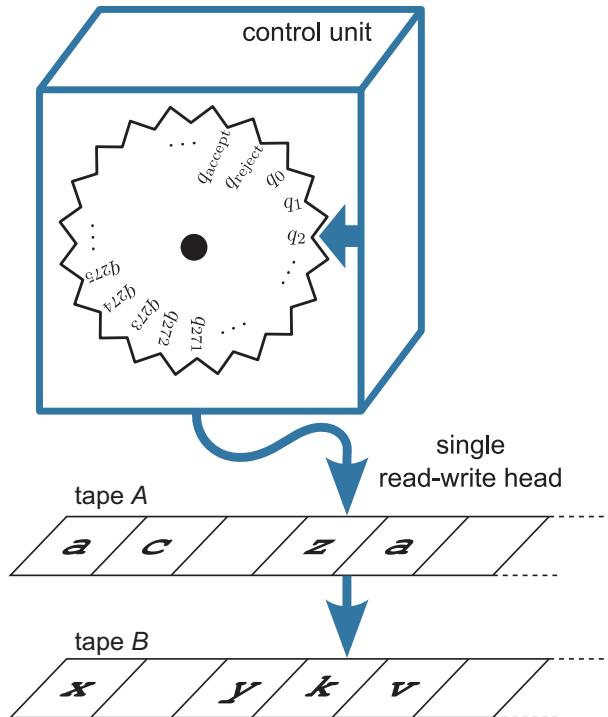
## Two-tape, single-head Turing machines

A two-tape Turing machine is identical to the standard (single-tape) version, except that there are now two infinite tapes, which we might call tape *A* and tape *B*—see figure 5.13. The input is placed on tape *A* before the computation begins, and tape *B* is initially filled with blanks. The machine has a single read-write head, which can be positioned over any single cell on *both* tapes. For example, in figure 5.13 the head is currently positioned at cell 3 on both tapes. It is *not* possible for the head to be positioned simultaneously at different cells on each tape—this feature will be added soon, when we have multiple heads. The two-tape, single-head machine can read, erase, or write the current cell on both tapes in a single step. For example, given the configuration in figure 5.13, the machine could erase the pair  $(z, k)$  at cell 3 and write  $(y, \square)$  instead. The transition function depends on the values in the current cell on both tapes, of course.

Compared to the standard single-tape machine, it is sometimes easier to program this two-tape, single-head machine. For example, implementing `countCs` on the two-tape model would be simplified a little because we could store the genetic string input on tape *A* and the binary counter on tape *B*.

But is a two-tape machine actually *more powerful* than a single-tape machine? This depends on what we mean by “more powerful.” If we are concerned with how many steps it takes to complete a computation, then it can be shown that a two-tape, single-head machine is a little more efficient than a single-tape one with the same alphabet. But if we are concerned with whether a computation can be performed at all, it turns out that the two models are equivalent. Specifically, a single-tape machine can *simulate* a two-tape, single-head machine. This is formalized in the following claim.

**Claim 5.1.** Given a two-tape, single-head Turing machine *M*, there exists a single-tape Turing machine *M'* that computes the same function as *M*.

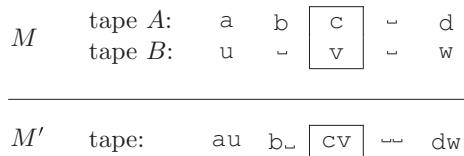


**Figure 5.13:** A two-tape, single-head Turing machine.

*Proof of the claim.* Recall that our Turing machines usually use the set of ASCII characters as their alphabet—an alphabet of 129 symbols, including alphanumeric symbols such as `a`, `b`, `5`, `X`, `Y`, and the special non-ASCII blank “`_`”. However, in defining a Turing machine, we are free to use any alphabet we want. The machine  $M'$  will instead use *pairs* of ASCII characters (or blanks) as its alphabet. This gives us a huge set of  $129 \times 129 = 16,641$  possible symbols, including `aa`, `ab`, `b5`, `xa`, `XY`, `YY`, `b_1`, `x_1`, and `y_1`. Figure 5.14 demonstrates the idea. In this example, the head  $M$  is at cell 2, scanning (`c`, `v`). This is simulated by  $M'$ , whose head is also at cell 2, scanning the single symbol `cv`. If, for example,  $M$  now erases the `c` on tape  $A$  and replaces it with a `y`,  $M'$  simulates this move by erasing the `cv` and replacing it with `yv`. In general,  $M'$  performs the same operations as  $M$ , but translates each two-tape operation into the corresponding single-tape operation on the larger alphabet. (A completely formal proof would give a careful specification of the transition functions for  $M'$ , but these details are omitted. Hopefully, the description of  $M'$  is already convincing enough.) □

## Two-way infinite tapes

Since the JFLAP software uses Turing machines with two-way infinite tapes, it's worth briefly mentioning how these can be simulated using our own one-way variant. This relies on a simple trick: we use a two-tape machine, where tape  $A$



**Figure 5.14:** Simulation of a two-tape, single-head Turing machine  $M$  (*top*) by a single-tape Turing machine  $M'$  (*bottom*) with a larger alphabet.

stores the content of the left-hand half of the two-way infinite tape, and tape  $B$  stores the content of the right-hand half. Note that the content of tape  $A$  is actually a “reflected” version of the two-way infinite tape’s left half, storing cells  $-1, -2, -3, \dots$ . Tape  $B$  stores the nonnegative cells in the usual order:  $0, 1, 2, \dots$ .

### Multi-tape, single-head Turing machines

The previous claim applied to two tapes, but a very similar proof shows that a standard Turing machine can simulate a  $k$ -tape, single-head machine, for any positive integer  $k$ . For example, to simulate a five-tape machine, we use an alphabet whose symbols consist of groups of five ASCII characters or blanks (like “ $x\_\_2yz$ ”). So, in terms of computability, a standard machine is just as powerful as any multi-tape, single-head machine.

At first glance, these proofs based on large alphabets might seem extreme and unrealistic. For example, the single-tape simulator of the five-tape machine mentioned above requires  $129^5 \approx 36$  billion symbols in its alphabet. Mathematically, the proof is correct. But the vast size of this alphabet might lead you to believe that it is wildly impractical. In fact, the opposite is true: this process of amplifying the alphabet mirrors the history of computer architecture over the second half of the 20th century. Many early computers used a so-called *16-bit architecture*. This means that the computer spends most of its time reading and writing quantities that are 16 bits long. Roughly speaking, the alphabet of such a computer therefore consists of all 16-bit binary strings—and there are  $2^{16} \approx 66$  thousand such strings. In the later decades of the 20th century, 32-bit architectures became common, and these computers have an alphabet size of  $2^{32} \approx 4.3$  billion binary strings. In the 21st century, this book was written using a 64-bit computer, whose alphabet size is  $2^{64} \approx 1.8 \times 10^{19}$ . So proofs that employ large alphabet sizes are both mathematically correct and physically plausible.

In addition, it’s worth noting that there are also reasonably efficient ways of simulating  $k$ -tape machines on a single-tape machine *with the same alphabet*. Michael Sipser’s book gives elegant descriptions of these approaches; we won’t investigate them any further.

One additional technical detail will be important later. In any multi-tape machine, one of the tapes is designated as the *input/output tape*. This is the tape on which the input is written before the machine begins running, and the content of this same tape is defined to be the output of the machine when it halts.

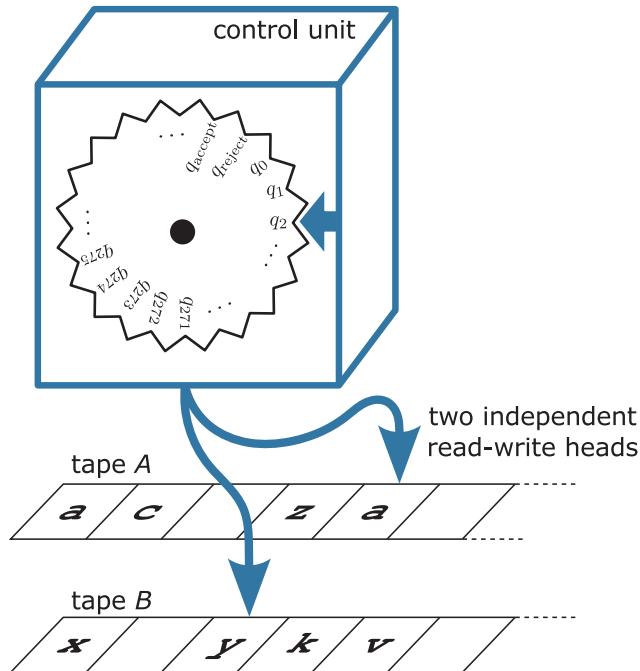


Figure 5.15: A two-tape, two-head Turing machine.

## Two-tape, two-head Turing machines

The requirement to scan the same cell of all tapes means that programming a multi-tape, single-head machine is still rather laborious. It would be much easier if we could allow the machine to have multiple independent read-write heads, as in figure 5.15. The key new feature is demonstrated in this figure, since the head for tape A is scanning cell 4, while the head for tape B is scanning cell 2.

Once again, it turns out that this new feature improves efficiency and ease of programming, but adds nothing in terms of computability: a standard machine can still compute anything that a two-tape, two-head machine can compute. To prove this, we don't need to go all the way back to the standard single-tape machine. We just need to prove the next link in our chain of simulations (as discussed on page 87). Specifically, we prove the following claim, showing that a multi-tape, single-head machine can simulate a multi-tape, multi-head machine.

**Claim 5.2.** Let  $M$  be a multi-tape, multi-head Turing machine. Then there exists a multi-tape, single-head Turing machine  $M'$  that computes the same function as  $M$ .

*Proof of the claim.* Initially, assume  $M$  has only two tapes,  $A$  and  $B$  (the generalization to more tapes will be obvious). Machine  $M'$  will have four tapes:  $A_1$ ,  $A_2$ ,  $B_1$ ,  $B_2$ , as shown in figure 5.16. Tapes  $A_1$  and  $B_1$  in  $M'$  perform exactly the same roles as tapes  $A$  and  $B$  in  $M$ . Tapes  $A_2$  and  $B_2$  are used to keep track of where the head of  $M$  is on each of its tapes. Specifically,  $A_2$  and  $B_2$  are always

$M$	tape $A$ :	a	b	c	$\leftarrow$	d
	tape $B$ :	u	$\leftarrow$	v	$\leftarrow$	w
$M'$	tape $A_1$ :	a	b	c	$\leftarrow$	d
	tape $A_2$ :	$\leftarrow$	$\leftarrow$	$\leftarrow$	$\leftarrow$	x
	tape $B_1$ :	u	$\leftarrow$	v	$\leftarrow$	w
	tape $B_2$ :	$\leftarrow$	$\leftarrow$	x	$\leftarrow$	$\leftarrow$

**Figure 5.16:** Simulation of a two-tape, two-head Turing machine  $M$  (*top*) by a four-tape, single-head Turing machine  $M'$  (*bottom*). Machine  $M'$  simulates the position of the heads of  $M$  using “x” markers on tapes  $A_2$  and  $B_2$ .

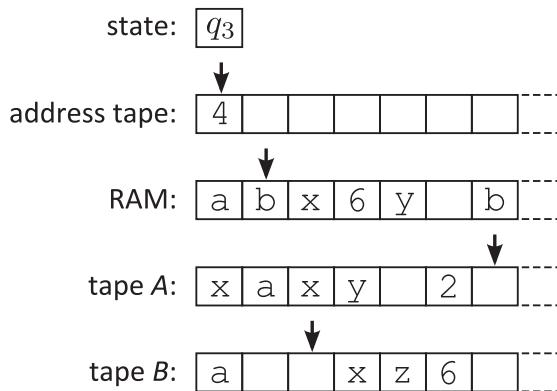
completely blank except for a single “x” symbol in the cell corresponding to the head position of  $M$ . In figure 5.16, for example,  $M$  is configured with head  $A$  at cell 4, and with head  $B$  at cell 2. Machine  $M'$  is simulating this configuration with an “x” at cell 4 on tape  $A_2$  and at cell 2 on tape  $B_2$ . The *real* head of machine  $M'$  happens to be scanning cell 0. To simulate a step of  $M$ , the real head of  $M'$  must sweep to the right from cell 0, searching for the x’s on tapes  $A_2$  and  $B_2$ , and then altering tapes  $A_1$  and  $B_1$  in the same way that  $M$  alters  $A$  and  $B$ . Again, we omit the detailed specification of the transition functions for  $M'$ .  $\square$

It’s worth noting that the simulation in this proof is rather inefficient. Suppose  $M$  is currently “using” its first  $n$  cells (where “using  $n$  cells” means the last non-blank occurs in cell  $n-1$ ). Then  $M'$ , while searching for x-markers, may need to scan all cells from 0 to  $n-1$ , just to simulate a single move of  $M$ . After  $M$  has made  $n$  moves,  $M'$  could have made as many as  $n^2$  moves just to find the x-markers, and ignoring any moves required to update the tapes. This analysis can be made more rigorous, and the conclusion is that if  $M$  finishes its computation after  $t$  steps, then  $M'$  terminates after  $ct^2$  steps, for some constant  $c$  that depends only on  $M$  and not on its input. Whether you consider this quadratic slowdown (from  $t$  to  $ct^2$ ) to be woefully inefficient or surprisingly good, depends on your point of view. We will return to this discussion, examining both points of view, in section 10.5.

Now that we know all these models have equivalent computational power, the distinction between single and multiple heads becomes mostly irrelevant. But for concreteness, let’s agree that from now on the term *multi-tape Turing machine* refers to a machine with multiple tapes *and* multiple heads.

## 5.4 FROM MULTI-TAPE TURING MACHINES TO PYTHON PROGRAMS AND BEYOND

Here’s a summary of what we know so far: standard Turing machines can simulate multi-tape, single-head machines, which can simulate multi-tape, multi-head machines. These two simulation proofs are just the first steps in a chain of proofs that can be used to show how a standard Turing machine can simulate any real computer or programming language. To examine the entire chain in detail would take us too long (and, quite frankly, would be too tedious). Instead, we examine brief sketches of the remaining links in the chain of simulations.



**Figure 5.17: A random-access Turing machine.** This machine can read or write the cell referenced by its address tape in a single step. In the configuration shown here, the machine has immediate access to the  $y$  at  $\text{RAM}[4]$ .

### Multi-tape Turing machine → random-access Turing machine

A *random-access Turing machine* has, in addition to the usual tapes and heads, two special tapes that are initially blank: an *address tape* and a *random-access tape*. We refer to the random-access tape as the *RAM*, in analogy with the random-access memory (RAM) used by real computers. Given a nonnegative integer  $n$ , the notation  $\text{RAM}[n]$  represents the symbol stored at cell  $n$  of the RAM. Figure 5.17 shows an example where  $\text{RAM}[0] = a$  and  $\text{RAM}[5] = \perp$ . For technical reasons that aren't discussed further here, it's important that the RAM is not used as the input/output tape; one of the other "ordinary" tapes must be the input/output tape.

The address tape typically stores a nonnegative integer  $n$  (in, say, decimal notation). The symbols comprising  $n$  can be read and written by the machine in the same way as symbols on any other tape, so the value of  $n$  typically changes while the machine is running. The value of  $n$  is interpreted as an *address* in the RAM. That is, we think of the address tape as storing a reference to  $\text{RAM}[n]$ . The special ability of the random-access Turing machine is that it can access  $\text{RAM}[n]$  in a single step, regardless of the current head position. In figure 5.17, for example, the address tape specifies  $n = 4$ , so the machine's next step depends on the " $y$ " at  $\text{RAM}[4]$ , in addition to the other scanned symbols. In addition, the " $y$ " could be replaced by another symbol in a single computation step.

It's not hard to simulate a random-access Turing machine with a standard multi-tape machine. Each random access is simulated by running a small subprogram that counts the relevant number of cells from the left end of the RAM.

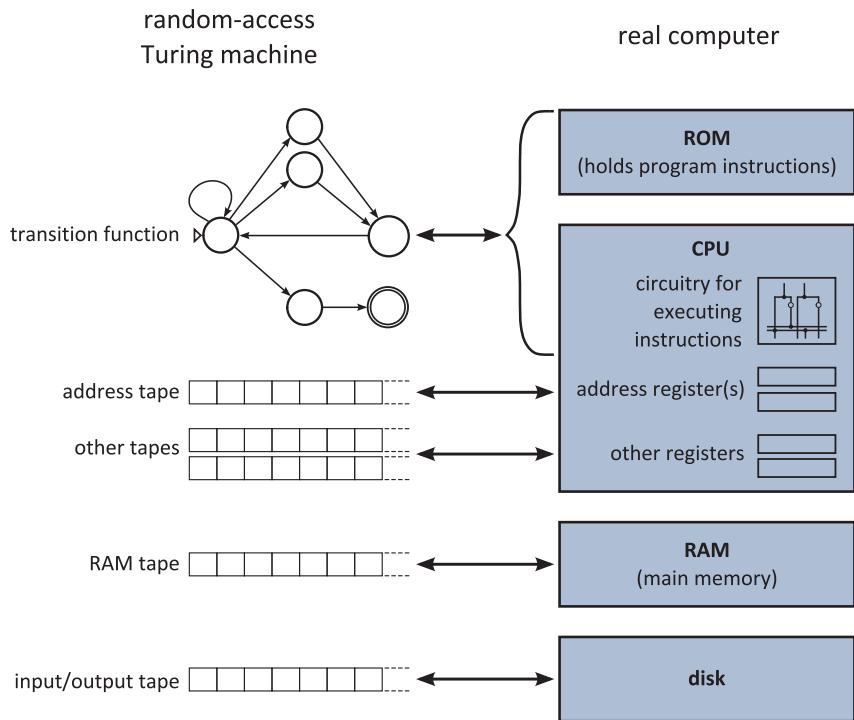
### Random-access Turing machine → real computer

From the random-access Turing machine, we can leap straight to simulating a real computer.

The description of this simulation requires knowledge of some elementary computer architecture. Some essential terms are summarized in the following list and in figure 5.18; you can skip over these terms if you are already familiar with them:

- **Register.** A register is a piece of circuitry built directly into the CPU, used for storing exactly one item of information. Registers have a fixed size, which is typically 64 bits on modern computers. So, a single register can store a string of 64 zeros and ones (e.g., “11010011 . . . 00111”). This string can of course be interpreted in many different ways. For example, it could represent a 64-bit integer, or a string of 8 ASCII characters (since ASCII characters are usually encoded using 8 bits each), or a 64-bit machine instruction (see below for more details about instructions). Modern computers usually have dozens of registers.
- **RAM.** Random-access memory (RAM) is the computer hardware that stores information in active use by the computer. The phrase “random access” refers to the fact that each cell of the memory can be directly accessed (i.e., read or written) based on its numeric address. For example, the action “store the value 37 at address 568998” can be accomplished with a single STORE instruction (see below for more details about instructions). Similarly, the action “load the value at address 4638001 into register 5” can be accomplished with a single LOAD instruction.
- **ROM.** Read-only memory (ROM) is a piece of specialized memory hardware for storing information that can be read, but not written. Typically, the ROM stores a fixed program that tells the computer how to boot when it is first switched on.
- **Instruction set:** Every CPU has a relatively small, fixed set of actions it can perform. Each of these actions is called an *instruction*, and the collection of all possible actions is called the *instruction set*. Modern CPU architectures vary widely in terms of the size and nature of their instruction sets. For concreteness, let’s imagine a CPU with about 100 different instructions in the instruction set—this is typical of many existing CPU architectures. The instruction set usually includes instructions such as LOAD and STORE (for transferring data to and from RAM); ADD and MULTIPLY (for doing arithmetic on values stored in registers); BRANCH (for jumping to a different part of the program); and AND, OR, and NOT (for performing logical operations on register values). The detailed meaning of individual instructions is irrelevant for this book. The important point is that any program, regardless of which programming language it is written in, is eventually converted into CPU instructions before being executed. Therefore, to simulate a modern computer, we need only show how to simulate each instruction in the instruction set—and there is a relatively small number of these (say, 100).

Real computers have sophisticated ways of reading input and producing output, via disks, the network, keyboards, monitors, and many other devices. But we ignore these here. Instead, let’s think of a computer that operates as follows. The program to be executed is stored in ROM as a sequence of CPU instructions. The input to the program is stored on a disk or other external device, which plays



**Figure 5.18:** Simulating a real computer via a random-access Turing machine.

the role of the input/output tape. The content of this disk after the program halts is defined to be the output. Real computers do not use blank symbols, so the end of the input and output must be marked in some problem-specific manner (e.g., surrounded by x's, as in some of our earlier Turing machine examples). Also, real computers do not have explicit accept and reject states, so we switch over to the convention that the outputs "yes" and "no" correspond to accept and reject respectively.

When we adopt these conventions, it's relatively easy to sketch the simulation of a real computer  $C$  by a random-access Turing machine  $R$ , as shown in figure 5.18. The address tape of  $R$  mimics any register of  $C$  that is used for storing memory addresses. The general-purpose registers of  $C$  are each simulated by a separate tape in  $R$ , and  $C$ 's disk is simulated by  $R$ 's input/output tape. Also,  $C$ 's RAM is simulated by  $R$ 's RAM,  $C$ 's ROM program is built into  $R$ 's transition function, and  $R$  has a state for each instruction in the ROM program, and various other states that help with implementing these instructions. In fact, each instruction type could be implemented with a small "building block" Turing machine, similar to those in figure 5.12 (page 85). As mentioned above, modern computer architectures have a modest number of instruction types—say, 100 or so. Thus,  $R$  might contain 100 building blocks, each implementing a different instruction type.

Given a particular, real computer architecture (say, Intel's x64 architecture), we could go through the instruction types one by one to check that building

block Turing machines can be constructed for each instruction type. This is too tedious for us to do in practice, but given the very simple nature of CPU instructions, it is nevertheless a reasonable claim. We will not attempt any more persuasive proof here. However, it is worth mentioning that undergraduate computer organization and architecture classes are sometimes taught with this “bootstrapping” concept as an underlying theme. The idea is that by starting with a few simple components (like AND and OR gates), we can build successively more complex components until we have a complete modern computer and operating system. My favorite demonstration of this is the course From NAND to Tetris, based on the book *The Elements of Computing Systems* by Noam Nisan and Shimon Schocken.

There is one important caveat in simulating a fully fledged computer with a random-access Turing machine. Modern computers typically have multiple *CPU cores*, enabling them to run multiple programs simultaneously. Although we will see in chapter 8 that Turing machines can simulate multi-core computers, nothing in the above description explains how to do this. Therefore, at this point, we are considering computers with only a single core.

### Modern computer → Python program

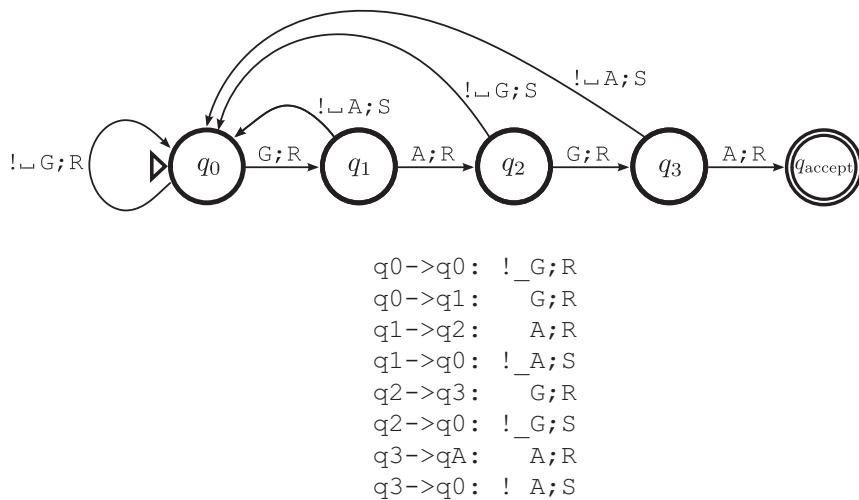
Once we can simulate a modern single-core computer, we can simulate any software too, and this includes Python programs. One way to see this is to appeal to the well-known equivalence between hardware and software. But an alternative is to imagine configuring your computer so that it automatically runs some particular Python program on startup. Once it is set up in this way, your computer can be regarded as a single, fixed piece of computer hardware that simulates the Python program.

This completes the chain of simulations outlined on page 87: we started with a standard single-tape Turing machine, and eventually were able to simulate a Python program. Many formal details were omitted. Nevertheless, if we took the time and effort needed to upgrade these arguments to rigorous mathematical proofs, then we would have proved the following:

**Claim 5.3.** Let  $P$  be a Python program. Then there exists a standard, single-tape Turing machine  $M$  that computes the same function as  $P$ .

## 5.5 GOING BACK THE OTHER WAY: SIMULATING A TURING MACHINE WITH PYTHON

In this book, Python programs are the main model of computation. But most theoretical computer science is done in terms of Turing machines. Therefore, it’s important for us to realize that Python programs and Turing machines are precisely equivalent, in terms of what computations they are capable of performing. In the previous section, we saw that Turing machines can simulate Python programs. But what about the other direction? Can Python simulate a Turing machine? If so, this would complete the proof that the two models are equivalent.



**Figure 5.19: Example of a Turing machine description.** *Top:* The containsGAGA Turing machine. This particular version of containsGAGA is identical to the bottom panel of figure 5.6. *Bottom:* An example of desc(containsGAGA)—a *description* of the containsGAGA machine above, using ASCII text. This machine description is available in the book’s resources as a text file, containsGAGA.tm. For some more complex examples, see the other .tm files in the book resources.

Fortunately, it’s not hard to implement a Turing machine in Python. We first need to agree on notation for describing Turing machines. One simple approach is demonstrated in figure 5.19, which gives an ASCII representation of the containsGAGA Turing machine. We call this a *description* of containsGAGA, written desc(containsGAGA). Note that desc(containsGAGA) is provided with the book resources, as the file containsGAGA.tm. So you can inspect, edit, and experiment with this description; some suggestions for how to do this are given below.

Given a Turing machine  $M$ , desc( $M$ ) is not unique. We could easily define desc( ) more carefully to ensure uniqueness, but it will be easier to abuse notation and let desc( $M$ ) stand for any valid description of  $M$ .

The notation in figure 5.19 is self-explanatory, and we won’t bother to give a formal specification of how to describe Turing machines in ASCII—hopefully, it’s immediately obvious that this can be done. In the simple scheme adopted by figure 5.19, we have reserved some ASCII characters for special purposes (including `_!` ; : ), so these characters cannot be in the machine alphabet. Once we have agreed notation for Turing machines, it’s a fun exercise to write a Python program that reads in the description of a Turing machine and its input, and then simulates the execution of the machine. This is a highly recommended exercise, but the book resources also provide an implementation if needed. The initial fragment of this implementation is shown as simulateTM.py in figure 5.20.

Whether or not you write your own version of simulateTM.py, it’s essential that you experiment with using it. Simulating Turing machines (and, eventually, Python programs) is a fundamental technique used in the remainder of the book.

```

1 def simulateTM(tmString, inString):
2     # simulate Turing machine described by tmString, with input inString
3     # ...

```

**Figure 5.20:** The Python program `simulateTM.py`. The body of the program is omitted from this listing. This program simulates a Turing machine described by `tmString`, on input `inString`, and returns its output. The description `tmString` is given in the format of figure 5.19.

Therefore, make sure you can do some simulations in IDLE. For example,

```
>>> simulateTM(rf('containsGAGA.tm'), 'TTGAGATT')
```

returns “yes”. Experiment with other input strings to `containsGAGA`, then move on to simulating some other Turing machines, such as

```
>>> simulateTM(rf('binaryIncrementer.tm'), 'x101x')
```

All of the Turing machines in this chapter are available as `.tm` files with the book resources.

### A serious caveat: Memory limitations and other technicalities

There is one severe limitation in using Python programs to simulate Turing machines. Turing machines have an infinite tape, whereas real computers have a finite amount of memory and other storage. Therefore, given any real computer system, we can create a Turing machine that exceeds the storage limitations of that computer system.

On this point, we must openly admit defeat. This book attempts to use a “practical” approach to “theoretical” computer science, including the use of real Python programs instead of Turing machines whenever possible. However, real Python programs run on real computers, which are finite. Turing machines are mathematical abstractions that are infinite. So the “practical” approach cannot succeed perfectly. Instead, we make the following concession to the theoretical view: we imagine our Python programs are running on a computer that has as much memory as required. Any program that eventually halts does use only a finite amount of memory. Therefore, we can—in principle—simulate any *halting* Turing machine on a real computer. In practice, even this may be impossible. For example, the number of bits of memory required may exceed the number of particles in the universe, meaning that no plausible physical device could actually perform the computation. But at this point, we’re deviating into the realm of philosophy. For the purposes of this book, it’s sufficient to assume our chosen computer system *C* (which is used in the definition of a Python program—see page 23) has enough memory to simulate a given, halting Turing machine.

Even if we permit unbounded memory, Python programs have certain other technical limitations. For example, some Python implementations impose a bound on the length of a sequence. Perhaps we could imagine an idealized version of Python in which restrictions of this kind are removed. Ultimately,

however, the only completely rigorous approach is to think of the Python programs in this book as a kind of shorthand notation for Turing machines.

## 5.6 CLASSICAL COMPUTERS CAN SIMULATE QUANTUM COMPUTERS

At the time of writing (late 2010s), almost all computers in active use are *classical* computers. This means that they rely on the laws of classical physics. In particular, each piece of circuitry is in exactly one of two possible states (e.g., electrical current is flowing, or no electrical current is flowing). In contrast, it is possible to design and build a *quantum* computer, which relies on the laws of quantum mechanics. In a quantum computer, each component has infinitely many possible states. This enables quantum computers to solve certain types of problems much more efficiently than classical computers.

The state of a quantum computer evolves according to known physical laws. Therefore, even though the state of a quantum computer may be immensely complex, it can always be simulated to any desired accuracy on a classical computer. This is done by numerically solving the equations that govern quantum mechanics. Hence, any problem that can be solved by a quantum computer can also be solved by a classical computer. Quantum computers can also simulate classical computers. We therefore conclude that quantum and classical computers are equivalent in terms of which problems they can solve.

## 5.7 ALL KNOWN COMPUTERS ARE TURING EQUIVALENT

We have now seen numerous computational models, including the following six models: (i) standard Turing machines, (ii) multi-tape Turing machines, (iii) random-access Turing machines, (iv) Python programs, (v) classical computers, and (vi) quantum computers. The usual caveat applies to the last three models: they are always provided with enough memory to complete a given, halting computation. Two computational models  $A$  and  $B$  are called *Turing equivalent* if they can solve the same set of problems. We say that  $A$  is *weaker* than  $B$  if the set of problems  $A$  can solve is a strict subset of the problems  $B$  can solve. Thus, we can summarize the results of this chapter by saying that all six computational models listed above are Turing equivalent.

This may already be surprising, but much more is true. Let's call a model “physically realistic” if it can be implemented by a device that operates according to the known laws of physics. Then the following is an empirical fact: all physically realistic models of computation that have ever been invented, discovered, or proposed are either Turing equivalent to the above six models, or weaker than the above six models. Therefore, it's widely believed that existing computers can solve all problems solvable by any other kind of physically realistic computer—past, present, or future. This claim is closely related to an idea known as the *Church–Turing thesis*, which is discussed in more detail in chapter 15.

We will see examples of weaker models, such as finite automata, in chapter 9. And there do exist stronger models of computation, but they are not physically realistic. For example, we can imagine a computer  $Z$  that is a standard modern computer augmented with a special black box that can solve YESONSTRING.

Computer  $Z$  is not Turing equivalent to a standard Turing machine or to a classical computer. Theoretical computer scientists study machines like  $Z$  to gain a deeper understanding of the properties of computation. (For example, it's easy to show that  $Z$  can solve the halting problem, which is an undecidable problem defined in chapter 7.) But we believe machines like  $Z$  can never be built, so they're studied only for their theoretical properties.

## EXERCISES

**5.1** As suggested on page 79, use JFLAP to implement a Turing machine swapCandG, that changes every “C” to a “G”, and every “G” to a “C”. Test your machine on multiple inputs.

**5.2** Use JFLAP to create a binaryDecrementer machine, whose behavior is similar to the binary incrementer of figure 5.10 except that it decrements binary numbers rather than incrementing them. Your machine's output is permitted to contain leading zeros. For example, “ $x10x$ ” becomes “ $x01x$ ”. You may assume the input represents a positive integer.

**5.3** Create an accepter Turing machine with the following properties: the machine accepts strings containing at least five G's and at most three T's; all other strings are rejected.

**5.4** Consider the Turing machine  $M$  with the following description:

```

q0->q1: 123456789;R
q0->q3: H;R
q1->q1: 0123456789;R
q1->q2: _;0,R
q2->qH: _;5,S
q3->q4: A;R
q4->q5: L;R
q5->q5: T;S

```

The alphabet of  $M$  consists of the alphanumeric ASCII characters 0–9, a–z, and A–Z.

- (a) Describe the set of strings for which  $M$  halts.
- (b) Describe the set of strings for which  $M$  halts in  $q_{\text{halt}}$ .
- (c) Describe the set of strings rejected by  $M$ .
- (d) Considered as a transducer mapping positive integers to positive integers, what mathematical function does  $M$  compute?

**5.5** Create a Turing machine with the following properties: the machine is a transducer, and its alphabet is the ASCII alphabet augmented with the blank symbol. The input is guaranteed to contain only characters from the genetic alphabet {C, A, G, T}—so your machine need only work correctly on genetic string inputs. If the input contains no G's, the output is the empty string. If the input contains a G, then the output is the same as the input except that a Z is inserted before the first G. For example, on input “CCAAGTGT” the output is “CCAAZGTGT”.

**5.6** Construct a Turing machine that reverses its input. You may assume that the input is a genetic string demarcated by “x” symbols. Feel free to use any building blocks defined in this chapter.

**5.7** Construct a Turing machine that deletes all G’s from its input. For example, “TTGAGGT” becomes “TTAT”. You may assume the input is a genetic string.

**5.8** Suppose we wish to construct a Turing machine noMoreCsThanGs, which accepts genetic strings whose number of C’s is no more than the number of G’s. This can be achieved by making some simple changes to the moreCsThanGs machine of figure 5.9. Describe the changes that would be required. (There is no need to draw a diagram of the resulting machine.)

**5.9** The proof on page 88 was written for the specific case of a machine  $M$  with two tapes and 129 symbols in its alphabet. The resulting simulator  $M'$  used a larger alphabet. In general, how many symbols would be in the alphabet of  $M'$  when the same kind of simulation is used for an  $M$  with  $k$  tapes and an alphabet of  $s$  symbols?

**5.10** Let  $M$  be a six-tape, single-head Turing machine that employs an alphabet of 10 symbols. Suppose we use a technique similar to the proof on page 88 to show that  $M$  can be simulated by a two-tape, single-head Turing machine  $M'$ . How many symbols are in the alphabet of  $M'$ ? Explain your answer. (Note:  $M'$  has *two* tapes, which differs from the proof on page 88. The intention of this question is that the simulation should use both of these tapes, approximately equally. You could, of course, use exactly the same simulation as the proof on page 88, by employing only one of the tapes on  $M'$ . But that would be no fun.)

**5.11** The proof on page 90 was written for the specific case of a machine  $M$  with two tapes and two independent heads. Suppose instead that  $M$  has  $k$  tapes,  $k$  independent heads, and  $s$  symbols in its alphabet. How many tapes and symbols are required by the corresponding single-head simulator  $M'$ ?

**5.12** Suppose  $M$  is a standard (single-tape) Turing machine using the five-symbol alphabet  $\Sigma = \{a, b, c, d, e\}$ . Give a high-level description of how to simulate  $M$  using a standard (single-tape) Turing machine  $M'$  that employs the four-symbol alphabet  $\Sigma' = \{a, b, c, d\}$ . Your simulation need only work correctly for input and output strings that contain no “e” characters. More precisely,  $M'$  must have the property that if  $I \in \Sigma^*$  and  $M(I) \in \Sigma^*$ , then  $M'(I) = M(I)$ .

**5.13** On page 95, it was stated that any CPU instruction could be implemented with a Turing machine building block. This question asks you to create such building blocks for two common CPU instructions. In each case, you may assume the input is correctly formatted and that any binary strings are nonempty.

- Construct a Turing machine that implements the SHIFT instruction. The input is a binary string flanked by x characters, and the output is the binary string shifted right by one bit. The rightmost bit is deleted, the string is padded on the left with a 0, and the final result is flanked by x characters. Example: “x11001x” becomes “x01100x”.
- Construct a Turing machine that implements the logical AND instruction. The input is two nonempty binary strings separated and flanked by

$x$  characters. The output is “ $x1x$ ” if both binary strings were nonzero, and “ $x0x$ ” otherwise. Examples: “ $x101x0100x$ ” becomes “ $x1x$ ”; “ $x00x010x$ ” becomes “ $x0x$ ”.

- 5.14** Suppose we are simulating a three-tape, three-head machine  $M$  using a multi-tape machine  $M'$  with a single head. The simulation follows the same methodology as claim 5.2 (page 90). Suppose  $M$ 's tapes are in the following configuration:

tape A:	a	b	c	a	d
tape B:	✉	✉	a	c	b
tape C:	✉	c	b	✉	✉

Give the corresponding configuration for  $M'$ .

- 5.15** Write your own version of `simulateTM.py` (see figure 5.20), without consulting the version provided with the book materials.

- 5.16** Consider the `binaryDecrementer` Turing machine from exercise 5.2. Give an ASCII description `desc(binaryDecrementer)` of this machine, using the construction suggested by figure 5.19.

- 5.17** Suppose only the following three resources are available to you: the Python program `simulateTM.py`, the Turing machine description `contains-GAGA.tn`, and the `rf` utility for reading files. Let  $S$  be an ASCII string. Give a Python command using only the three resources listed above, that will determine whether  $S$  contains the substring “GAGA”.

- 5.18** Let  $M$  be a random-access Turing machine as defined in this chapter. Recall that  $M$ 's address tape and RAM tape are initially blank, by definition.

- (a) Suppose  $M$  has executed  $t$  steps. What is the largest possible index  $k$  of a non-blank cell on the RAM tape? (Hint: For  $t = 2$ , the answer is  $k = 9$ , because the machine can write a “9” on the address tape in the first step, and then write a non-blank symbol to  $\text{RAM}[9]$  in the second step.)
- (b) Suppose  $M$  has executed  $t$  steps. What is the largest possible number  $K$  of non-blank cells on the RAM tape?
- (c) Suppose we wish to simulate the RAM tape using an ordinary tape  $T$ . The ordinary tape  $T$  records the RAM's contents using an extremely simple data structure: a flat list of address–content pairs separated by some appropriate choice of punctuation. For example, one specific convention would be that if  $T = "4:a;57:b;899:x"$ , this corresponds to  $\text{RAM}[4] = a$ ,  $\text{RAM}[57] = b$ ,  $\text{RAM}[899] = x$ .

Now suppose  $M$  has executed  $t$  steps, at which point we freeze the machine and reconstruct the ordinary tape  $T$  from the current state of the RAM tape. Give an estimate of the maximum number of non-blank cells on  $T$ . (Note: This estimate is used later, in figure 10.16.)

- 5.19** A quantum algorithm is an algorithm (i.e., a computer program) that runs on a quantum computer. Is it likely that someone will invent a quantum algorithm that solves the problem YESONSTRING? Why or why not?

**5.20** In chapter 1 we discussed an intractable problem known as *multiple sequence alignment* (see page 5). Let's now refer to this problem as MULTSEQALIGN. Suppose we augment a standard modern computer with a black box that can instantaneously solve any instance of MULTSEQALIGN, and equip the computer with as much memory as necessary. Is this type of augmented computer Turing equivalent to the class of multi-tape Turing machines? Explain your answer.

# 6



## UNIVERSAL COMPUTER PROGRAMS: PROGRAMS THAT CAN DO ANYTHING

We have perfected the general-purpose computer for scientific computations. Now the same device with little alteration we attempt to employ in a great variety of different situations for which it was never intended in the first place. Under these circumstances, if it should ever turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the basic logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered.

— Howard Aiken, *The Future of Automatic Computing Machinery* (1956)

In chapter 3, we already saw that computer programs can analyze other computer programs. For example, `countLines.py` (page 32) can analyze `containsGAGA.py` (page 16) to determine how many lines of source code are in the program `containsGAGA.py`. This kind of interplay between computer programs can be pushed even further: it's actually possible for one computer program to *execute* another computer program. To modern computer users, this is not surprising at all. In fact, many of the everyday operations we perform on a computer involve one program executing another program, as the following examples demonstrate:

- **Running a Python program.** Suppose you have opened `someProgram.py` in IDLE, and hit F5 to run the program. Behind the scenes, IDLE is using one program to execute another program. Specifically, IDLE launches the Python interpreter (which on my Windows computer happens to be the program `C:\Python34\python.exe`) and gives it as input the program `someProgram.py`. Thus, `python.exe` is executing `someProgram.py`.
- **Launching Microsoft Word.** Suppose you double-click on a Microsoft Word document, thus launching the program Microsoft Word (and also opening the document). In this case, you have used one program

(your operating system, such as Windows or OS X) to launch another program (Microsoft Word).

These examples might initially appear so mundane and obvious that it is not worth discussing them. But in fact, these everyday examples have an almost magical aspect to them: Isn't it rather wonderful that the Python interpreter seems to be capable of executing *any* legal Python program? Similarly, the fact that your operating system can launch and run any correctly written application program is immensely powerful. Arguably, this capability lies at the heart of the desktop computer revolution that began in the 1980s and continued into the 21st century with smartphones, tablets, and other computing devices. Perhaps the most wonderful thing about computers is that they can run any program you want—not just a fixed program or programs preinstalled by the manufacturer.

## 6.1 UNIVERSAL PYTHON PROGRAMS

Let's push a bit harder on this concept of programs executing other programs. In both examples above, we have a specific type of program (e.g., the Python interpreter, `python.exe`) executing a specific *different* type of other programs (e.g., source code written in the Python language, `someProgram.py`). Can we come up with an example of a program of a particular type, that executes other programs of that *same* type? The answer is yes. In fact, it turns out that essentially any programming language can be used to create a program that executes other programs written in that same language. So it's possible, for example, to write a Java program that can execute other Java programs. But some languages—including Lisp and Python—make this particularly easy, by building in a self-execution facility. In Python, for example, there is an `exec()` function that does exactly this. Thus, in IDLE we can enter

```
>>> command = "print('abc', 5+2)"
>>> exec(command)
```

When executed, this code produces the output “abc 7”. (Try it now, with a few different values for `command`.) Take careful note of the fact that the parameter `5+2` was *evaluated*, producing the value 7, rather than the string “`5+2`”.

Of course, `exec` can be used on arbitrarily complex strings, including entire Python programs. This leads us to our first example of what is called a *universal* program: `universal.py`, shown in figure 6.1.

Before we examine this code, experiment with it: run `universal.py`, then at the IDLE prompt, enter

```
>>> universal(rf('containsGAGA.py'), 'GTTGAGA')
>>> universal(rf('containsGAGA.py'), 'GTTAA')
```

Recall that `rf('containsGAGA.py')` reads the contents of the file `containsGAGA.py` into a string. We've used the program `containsGAGA.py` many times before; but for easy reference, its source code is repeated here in figure 6.2. Thus, the two Python shell commands above should produce the output “yes” and “no” respectively, correctly indicating whether the strings `GTTGAGA` and `GTTAA` contain `GAGA`.

```

1  def universal(progString, inString):
2      # Execute the definition of the function in progString. This defines
# the function, but doesn't invoke it.
3      exec(progString)
4      # Now that the function is defined, we can extract a reference to it.
5      progFunction = utils.extractMainFunction(progString)
6      # Invoke the desired function with the desired input string.
7      return progFunction(inString)
8

```

**Figure 6.1:** The Python program `universal.py`.

```

1  import utils; from utils import rf
2  def containsGAGA(inString):
3      if 'GAGA' in inString:
4          return 'yes'
5      else:
6          return 'no'

```

**Figure 6.2:** The Python program `containsGAGA.py`.

Now let's go back and understand the code for `universal.py` (figure 6.1). As a concrete example, assume we have issued the first of the two Python shell commands above, so that `progString` is a string containing the Python source code for `containsGAGA.py` (figure 6.2), and `inString` is “GTTGAGA”. At line 4 of `universal.py`, all lines of `containsGAGA.py` are executed. Note that this does not mean the function `containsGAGA()` is invoked; `containsGAGA()` is merely defined via the `def` keyword at line 2 of figure 6.2. This is a potentially confusing point, so let's be clear about its meaning. Before line 4 of `universal.py` (figure 6.1), the Python interpreter has never heard of the function `containsGAGA()`, and certainly can't invoke it. After line 4, the function `containsGAGA()` has been defined via the `def` keyword, so the Python interpreter has inserted `containsGAGA()` into the list of available functions, and subsequent lines in the program can invoke it.

Actually invoking this newly available function requires some slightly obscure Python, which we don't examine in detail here. The `utils` package provides `extractMainFunction()`, which scans its input and returns a reference to the given program's main function at line 6. See the comments in `utils.py` for details. Finally, line 8 invokes the function with the desired input (`inString`, which is “GTTGAGA” in our example), and returns the result.

To summarize, `universal.py` is a *universal* Python program because it can simulate any other Python program.

## 6.2 UNIVERSAL TURING MACHINES

So, it's clear that universal Python programs exist. What about Turing machines? It will come as no surprise that it's also possible to construct universal Turing

machines. In fact, Alan Turing constructed a universal Turing machine in his 1936 paper that first introduced the concept of (what we now call) Turing machines. But Turing's construction is notoriously difficult to understand, and even contained a few mistakes.

Fortunately, we don't need to understand the details of this construction, or any of the universal Turing machines that were subsequently developed by other scholars. There is an easier way, based on the equivalence of Python programs and Turing machines. We exploit this in proving the existence of a universal Turing machine  $U$  in the claim below.

But first, let's nail down a precise definition of a universal Turing machine. We would like our machine  $U$  to receive two parameters: (i) the description of some other Turing machine  $M$ , and (ii) the input string  $I$  which should be given to  $M$ . Then,  $U$  should mimic the behavior of  $M$  on input  $I$ . One problem is that Turing machines are defined to receive only a single string as input. So, we must first convert  $M$  into an ASCII string, then combine the result with  $I$ . The conversion of  $M$  is easy: we just produce  $\text{desc}(M)$ , a *description* of  $M$ , using the technique on page 96. Then  $\text{desc}(M)$  is combined with  $I$  using the encode-as-single-string function  $\text{ESS}()$ , introduced on page 61. The final result is denoted  $\text{str}(M, I)$ —the *string representation* of  $(M, I)$ . Formally, then, the input to  $U$  will consist of

$$\text{str}(M, I) = \text{ESS}(\text{desc}(M), I).$$

As an example, suppose  $M$  is the `containsGAGA` machine in the bottom panel of figure 5.6, and  $I$  is “ATTGACT”. Then  $\text{desc}(M)$  could be the string in figure 5.19, which is 120 characters long, and the input to  $U$  would be encoded as

$$\text{str}(M, I) = "120 \text{ q0->q0: !\_G; R\_\_q0->q1...q3->q0: !\_A; S\_\_q0->q1 ATTGACT"$$

(Recall that “ $\_\_$ ” represents a newline character.)

Now we can formally define universal Turing machines. A Turing machine  $U$  is *universal* if, for all Turing machines  $M$  and strings  $I$ ,

- $U$  accepts  $\text{str}(M, I)$  if and only if  $M$  accepts  $I$  (and similarly for rejects and halts);
- if  $M(I)$  is defined, then  $U(\text{str}(M, I)) = M(I)$ .

With this definition in hand, we can also prove the existence of universal Turing machines:

**Claim 6.1.** There exists a universal Turing machine  $U$ .

*Proof of the claim.* Recall that we have a program, `simulateTM.py`, which simulates Turing machines (see page 97). This program receives two parameters, but we can easily create the equivalent one-parameter program `simulateTM1.py` shown in figure 6.3. This uses the `DESS()` function to unpack two string parameters from a single string encoded using `ESS()`—see page 61.

Recall claim 5.3 on page 95: given any Python program, there exists a standard Turing machine that performs the same computation. Applying this claim to the Python program `simulateTM1.py`, we obtain a standard Turing machine  $U$  that performs the same computation as `simulateTM1.py`—specifically,  $U$  simulates its input. That is, given  $\text{str}(M, I)$ , machine  $U$  simulates  $M$  with input  $I$ .

```

1 from simulateTM import simulateTM
2 def simulateTM1(inString):
3     (progString, newInString) = utils.DESS(inString)
4     return simulateTM(progString, newInString)

```

**Figure 6.3:** The Python program `simulateTM1.py`.

It's easy to check that  $U$  satisfies the formal definition of a universal Turing machine given above.  $\square$

It was probably overkill to give a formal proof of this claim. By this point in the book, most readers will have established an instinctive belief that Turing machines and Python programs are equivalent forms of computation. Therefore, the mere existence of a universal Python program should give us complete confidence that universal Turing machines also exist.

## 6.3 UNIVERSAL COMPUTATION IN THE REAL WORLD

The existence and successful implementation of universal computers is arguably one of the biggest scientific surprises of the 20th century. It was certainly a surprise to the eminent computer pioneer Howard Aiken, whose quote appears at the start of this chapter. Aiken understood the abstract theory of universal programs, but from his viewpoint in the 1950s he could not foresee the emergence of today's multipurpose—indeed, *universal*—real-world computers.

Universal computers and programs are all around us. We exhibited one particular universal program `simulateTM.py`, and observed that it has an equivalent Turing machine  $U$ . But any modern computer has universality built into it at several layers, as described in figure 6.4. The hardware is universal, because it could be configured to load and run any Turing machine converted into machine code. The operating system is universal, because it can run application programs, and we can compile application programs that mimic any Turing machine. Some of the applications on your computer, such as the Python interpreter and the Java virtual machine (JVM), are themselves universal. The Python interpreter (e.g., `python.exe` on Windows computers) can run any Turing machine encoded as a Python program; the JVM can run any Turing machine encoded as a compiled Java program. And we can write a *single* Python program (`simulateTM.py`) that is also universal, because it can run any Turing machine  $M$  encoded as  $\text{desc}(M)$ . Note that all of this “real-world” universality is subject to the caveat discussed on page 97: we must add as much storage as needed to run any given program.

The examples of figure 6.4 show the surprising variety of universal computation, but they are misleading in one respect: they require considerable complexity (e.g., billions of hardware transistors and/or megabytes of software). In contrast, it is possible to construct rather simple universal Turing machines. The original universal machine, published by Turing in 1936, contained dozens of states and used about 20 symbols in its alphabet. Some 30 years later, the artificial

Layer	How this layer simulates <code>containsGAGA</code>
Hardware	<code>containsGAGA</code> booted as machine code
Operating system	compiled application <code>containsGAGA.exe</code>
Python interpreter	Python program <code>containsGAGA.py</code>
JVM	compiled Java program <code>containsGAGA.class</code>
Python program	<code>simulateTM.py</code> on contents of <code>containsGAGA.tm</code>

Figure 6.4: A modern computer is capable of universal computation at many layers.

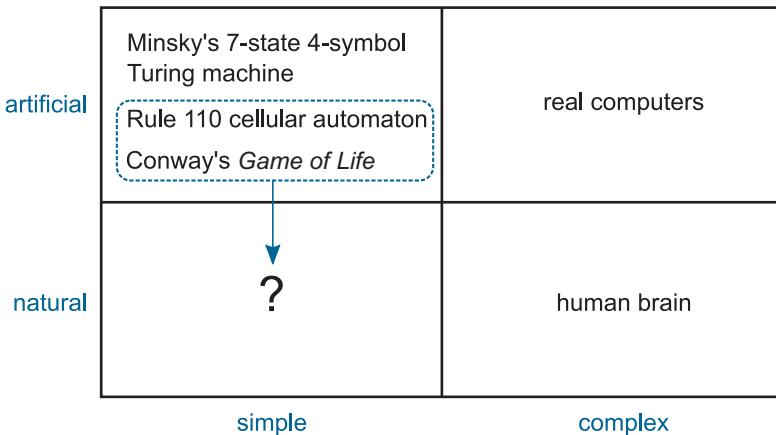


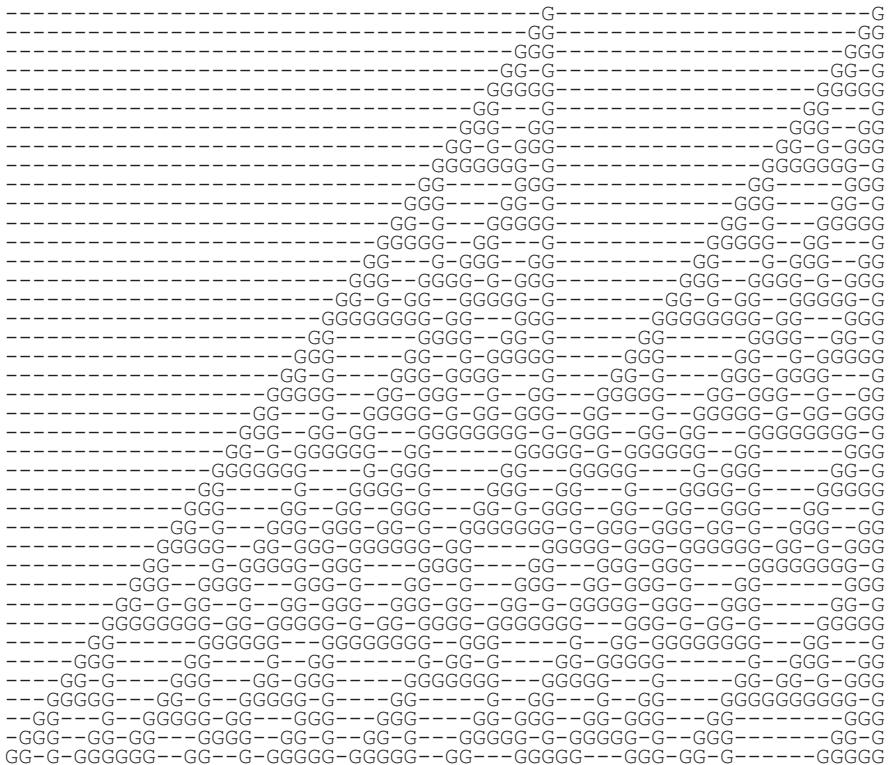
Figure 6.5: Examples of universal computation in the real world.

intelligence researcher Marvin Minsky discovered a universal Turing machine that uses only 7 states and 4 symbols. Today, even smaller universal machines are known. Hence, it's clear that universality does not require a great deal of complexity. (However, there is a trade-off: the small universal machines require long machine descriptions, and their simulations can be inefficient.)

Figure 6.5 summarizes this discussion, classifying the examples of real-world universal computation according to whether they are simple or complex, artificial or natural. So far, we've discussed only the top row of this  $2 \times 2$  matrix: universal programs that are "artificial" because they are built or described by humans and do not occur naturally in nature.

Let's now think about the bottom row of the matrix, which is concerned with the natural world. Are universal computers always constructed by humans, or can they arise spontaneously in nature? Well, the human brain is a universal computer, since a human can easily simulate a given Turing machine (perhaps with the help of pencil and paper). So, if you consider the human brain to be part of "nature," this is an immediate proof that natural universal computers exist. The brain contains billions of neurons, so this example of universality employs great complexity, placing it in the lower-right quadrant of figure 6.5.

But are there examples of universal computation that are both simple and "natural," thus placing them in the bottom left quadrant? There is no straightforward answer to this question. One potential example is a type of system called



**Figure 6.6: Example output of the “rule 110” cellular automaton, which can function as a universal computer.** The initial state of the automaton is given by the first row.

Each subsequent row represents one computational step, in which each symbol is based on a fixed rule depending only on the three immediate neighbors above. This particular automaton uses the 2-symbol alphabet {"G", "-"}, and is initialized using a string containing two "G"s. See `rule110.py` for details.

a *cellular automaton*. A cellular automaton consists of a one-dimensional row of cells that evolve according to simple rules that depend only on immediately neighboring cells. For example, cells could have four possible symbols C, G, A, T, and evolve according to rules like “if both neighbors are C, become G; if the left neighbor is T, become C.”

There is a famous cellular automaton called the “rule 110” automaton, which is particularly interesting for this discussion. The rule 110 automaton uses only two symbols and eight rules. We don’t describe the rules in detail here, but a sample output of the rule 110 automaton is shown in figure 6.6. You can investigate the details of how it works by consulting `rule110.py` in the book resources.

For us, the key point is that certain extremely simple cellular automata—including the rule 110 automaton, which requires only eight rules—can function as universal programs. There are important technicalities about how the program and its input are encoded, but the universal nature of rule 110 has been proved

Type of abstract analysis	Program	File or string parameters
analyze a string	containsGAGA.py	"ATGAG"
analyze another program	countLines.py	containsGAGA.py
simulate another program	universal.py	(containsGAGA.py, "ATGAG")
simulate a newly created program	universal.py	Altered version of repeatCAorGA.py, via alterGAGAtoTATA.py

**Figure 6.7:** The increasing levels of abstract analysis demonstrated by programs studied in this book. The highest level of sophistication is to first alter an existing program, then execute the altered version.

rigorously by mathematicians. Another famous example of a simple cellular automaton that can function as a universal computer is John Conway's *Game of Life*. As far as we know, the rule 110 automaton and the Game of Life do not occur in nature, so they probably belong in the top left quadrant of figure 6.5. On the other hand, the rules for these automata are so simple that we can easily imagine similar interactions occurring in nature, perhaps by manipulating string-like molecules such as DNA. So the arrow and question mark in the bottom left quadrant of figure 6.5 indicate that at least in principle, simple universal computation is possible in naturally occurring physical systems. Philosophers and physicists are still absorbing the implications of universal computation. Perhaps the future will reveal further connections between the nature of computation, and computations in nature.

Hopefully, you are now convinced that universal computation is a surprisingly widespread phenomenon. This idea will be reinforced further in section 7.7, where we meet two more universal systems: Diophantine equations, and Post tile systems.

## 6.4 PROGRAMS THAT ALTER OTHER PROGRAMS

So far in the book, and especially in this chapter, we have studied computer programs that employ increasing levels of abstract analysis. This progression is summarized in figure 6.7. In chapter 2, we started off with programs that analyze strings, such as

```
>>> containsGAGA.py('ATGAG')
```

Then in chapter 3, we moved on to programs that analyze other programs, such as

```
>>> countLines.py(rf('containsGAGA.py'))
```

```

1 def repeatCAorGA(inString):
2     if inString == 'CA':
3         return 'CACA'
4     elif inString == 'GA':
5         return 'GAGA'
6     else:
7         return 'unknown'

```

```

1 from universal import universal
2 def alterGAGAtoTATA(inString):
3     (progString, newInString) = utils.DESS(inString)
4     val = universal(progString, newInString)
5     if val == 'GAGA':
6         return 'TATA'
7     else:
8         return val

```

**Figure 6.8:** The Python programs `repeatCAorGA.py` and `alterGAGAtoTATA.py`.

Program `alterGAGAtoTATA.py` simulates an altered version of its input. Program `repeatCAorGA.py` is just one example of a program that could be altered in this way.

In the current chapter, we have seen programs that execute other programs via simulation, using `universal.py`:

```
>>> universal(rf('containsGAGA.py'), 'GTTGAGA')
```

The next level of abstraction is to execute a program that was created by the simulator itself. Typically, the newly created program will be an altered version of one that already existed.

As a concrete example, we will show how to simulate a newly created, altered version of a trivial program called `repeatCAorGA.py`. The original version of `repeatCAorGA.py` is shown in the top panel of figure 6.8. Notice how `repeatCAorGA.py` always returns one of the values CACA, GAGA, or unknown.

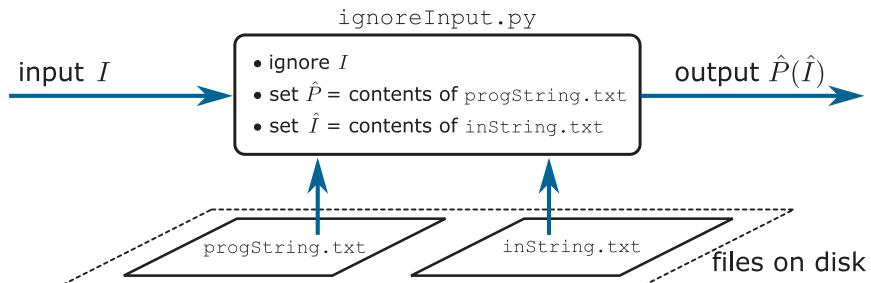
In the bottom panel of figure 6.8, we see `alterGAGAtoTATA.py`. This is our “program-altering program.” It takes as input a single string, which encodes two parameters in a single string according to the scheme on page 61. These parameters are unpacked (line 3), then simulated (line 4). Then, the crucial change is made (lines 5–8): if the simulated program returned “GAGA”, the newly created program returns “TATA”. Otherwise the return value is unchanged. This trick of changing the return value is simple but profound, and will have critical consequences in some of the proofs later in the book.

You may be wondering why `alterGAGAtoTATA.py` takes only a single parameter. Wouldn’t it be more logical for it to take two parameters ( $P, I$ ), representing the program  $P$  and input  $I$  whose return value  $P(I)$  should be altered? Yes, it would be more logical. But when we use our “program-altering programs” in proofs later on, we will need them to receive only a single parameter. That’s

```

1 def ignoreInput(inString):
2     progString = rf('progString.txt')
3     newInString = rf('inString.txt')
4     return universal(progString, newInString)

```



**Figure 6.9:** The program `ignoreInput.py`. Top: Source code. Bottom: Schematic representation. This strange but useful program ignores its input  $I$ , instead performing a preset computation specified by two files stored on disk.

why we presented `alterGAGAtoTATA.py` as a one-parameter program that unpacks  $P$  and  $I$  using the provided `utils.DESS()` function.

At this stage, the ability to simulate altered versions of programs may seem somewhat pointless. After all, it would be extremely easy to edit `repeatCAorGA.py`, directly changing the return of “GAGA” to “TATA”. Why jump through all the hoops of simulating an altered version instead? The reason is that our `alterGAGAtoTATA.py` simulation is much more powerful: it works on *any* program, not just `repeatCAorGA.py`. We will frequently use this universal simulate-and-alter technique in proofs of uncomputability.

### Ignoring the input and performing a fixed calculation instead

In chapter 7, we will need a simulation trick that initially seems slightly absurd: a program called `ignoreInput.py`, shown in figure 6.9. This program has a single input parameter, `inString`. But `inString` is ignored, so its value does not affect the output of the program. Instead, `ignoreInput.py` creates two new strings by reading the contents of the two files `progString.txt` and `inString.txt` (lines 2–3). The contents of these files are interpreted as a program and input string respectively, which are then simulated and the resulting value is returned (line 4).

The overall effect is that `ignoreInput.py` ignores its input, instead simulating the contents of `progString.txt`, with the input being the contents of `inString.txt`. Hence, by saving a particular program  $\hat{P}$  and input  $\hat{I}$  into `progString.txt` and `inString.txt` beforehand, we can arrange that `ignoreInput.py` always returns  $\hat{P}(\hat{I})$ , regardless of its own input. Check your understanding by deciding what the output of the following

```

1 from universal import universal
2 def recYesOnString(inString):
3     (progString, newInString) = utils.DESS(inString)
4     val = universal(progString, newInString)
5     if val == 'yes':
6         return 'yes'
7     else:
8         return 'no'

```

**Figure 6.10:** The Python program `recYesOnString.py`. This program recognizes, but does not decide,  $L_{YESONSTRING}$ .

commands will be:

```

>>> utils.writeFile('progString.txt', rf('containsGAGA.py'))
>>> utils.writeFile('inString.txt', 'GGGGGGGTTT')
>>> ignoreInput('GAGAGA')

```

Exercise 6.5 provides further practice for understanding `ignoreInput.py`.

## 6.5 PROBLEMS THAT ARE UNDECIDABLE BUT RECOGNIZABLE

Universal programs and the power of simulation let us understand the difference between recognizable and decidable languages more clearly. Recall from page 65 that, roughly speaking, a language or decision problem is recognizable if some program correctly decides all *positive* instances. In contrast, a language or decision problem is decidable if some program correctly decides *all* instances, whether positive or negative. We stated on page 66 that YESONSTRING and CRASHONSTRING are recognizable even though they’re undecidable. Now we are ready to prove that statement:

**Claim 6.2.** YESONSTRING and CRASHONSTRING are recognizable.

*Proof of the claim.* We claim the Python program `recYesOnString.py`, shown in figure 6.10, recognizes YESONSTRING. This follows because the simulation at line 4 always terminates for positive instances, returning the value “yes”. For negative instances, the simulation may terminate, in which case `recYesOnString.py` will correctly return “no” at line 8. It is also possible for a negative instance to cause an infinite loop at line 4, but this doesn’t matter for the definition of recognizability.

The proof for CRASHONSTRING is very similar, and we omit it.  $\square$

We’ve known since chapter 3 that YESONSTRING is undecidable. But a casual reading of the above proof appears to show that YESONSTRING is actually decidable. After all, doesn’t the program `recYesOnString.py` (figure 6.10) decide YESONSTRING? The answer is emphatically no, because `recYesOnString.py` may not terminate on some negative instances. Specifically, it can enter an infinite loop in the simulation at line 4.

We'll return to the idea of recognizability in section 8.8, where we will see more examples of unrecognizable languages.

## EXERCISES

**6.1** Conduct all the experiments suggested in this chapter, including the following ones:

- (a) Experiment with using `exec()` on various snippets of Python.
- (b) Use `universal.py` to determine the output of `countLines.py` on several different inputs.
- (c) Experiment with simulating altered versions of programs by trying various inputs to `alterGAGAtoTATA.py` and `alterGAGAtoTATA2.py`.

**6.2** Let  $U$  be a universal Turing machine. Suppose we wish to use  $U$  to simulate the operation of the `binaryIncrementer` machine (page 82) on input “`x1001x`”. Describe the input string  $I$  that should be given to  $U$  in order to achieve this simulation.

**6.3** Write a Python program `repeat.py` that takes as input a single string parameter  $S$ . The parameter  $S$  encodes two strings  $P$  and  $I$  in the usual way:  $S = \text{ESS}(P, I)$ , where  $P$  is expected to be a Python program. The output of `repeat.py` is the concatenation of  $P(I)$  with itself.

**6.4** Write a Python program `applyBothTwice.py` that takes as input a single string parameter  $S$ . The parameter  $S$  encodes three strings  $P, Q, I$  in the following way:  $S = \text{ESS}(\text{ESS}(P, Q), I)$ , where  $P$  and  $Q$  are expected to be Python programs. The output of `applyBothTwice.py` is  $Q(P(Q(P(I))))$ .

**6.5** Here is some essential practice for `ignoreInput.py`:

- (a) Give a sequence of IDLE commands, similar to those on page 113, that use `ignoreInput.py` to compute the result of `longestWord.py` on input “`a bb ccc`”.
- (b) Assuming the file system remains unchanged after you have executed the commands from part (a), what is the output of the following command?

```
>>> ignoreInput('dddd')
```

- (c) Again assuming an unchanged file system, and abbreviating `ignoreInput()` to `ig()`, determine whether the following statements are true or false: (i) For all strings  $I$ ,  $\text{len}(\text{ig}(I)) = 3$ ; (ii) There exists some string  $I$  such that  $\text{ig}(I) = "abcde"$ .

**6.6** Write a Python program that recognizes, but does not decide, the language of genetic strings containing “GAGA”.

**6.7** Prove that if  $L_1$  and  $L_2$  are recognizable languages, then  $L_1 \cap L_2$  is also a recognizable language.

**6.8** Complete the proof of claim 6.2 (page 113), by giving a detailed and rigorous proof that  $L_{\text{CRASHONSTRING}}$  is recognizable.

**6.9** Consider the problem THREEZSONEMPTY, defined as follows: the input is a Python program  $P$ , and the solution is “yes” if  $P(\epsilon)$  is a string containing at least three z’s. Prove that  $L_{\text{THREEZSONEMPTY}}$  is recognizable.

**6.10** Give an example of a computational problem not mentioned in this book that is recognizable but not decidable. Explain why your example is recognizable but not decidable.

# 7



## REDUCTIONS: HOW TO PROVE A PROBLEM IS HARD

Add the wine and boil it down rapidly, scraping up coagulated roasting juices and reducing the liquid to 2 or 3 tablespoons.

— Julia Child, *Mastering the Art of French Cooking* (1961)

**M**athematicians and computer scientists often *reduce* one problem to another problem. A vague but useful definition of this is as follows. Suppose we are given two computational problems,  $F$  and  $G$ . Then the phrase “ $F$  reduces to  $G$ ” means “ $F$  can be solved if  $G$  can be solved.” In other words, reductions are all about using the solution to one problem ( $G$ ) to solve another problem ( $F$ ). For this chapter, you’ll need an excellent understanding of the technical definition of “computational problem.” This might be a good time to revisit the definition on page 53.

It’s important to realize right from the outset that theoretical computer scientists use reductions in a particular way that is completely different to the way reductions are used by mathematicians, engineers, and applied computer scientists. In a nutshell, mathematicians mostly use reductions to show that a problem is *easy*, whereas theoretical computer scientists mostly use reductions to show that a problem is *hard*. The next two sections give examples of these two different approaches to reductions, which we call *reductions for easiness* and *reductions for hardness*.

### 7.1 A REDUCTION FOR EASINESS

Our first example will be a reduction “for easiness”—this is the kind we will mostly *not* be using in the rest of the book, but it will help set the stage nicely. The example uses the two decision problems described formally in figure 7.1: `ISODD` and `LASTDIGITISEVEN`. Both problems take an integer  $M$  as input. `ISODD` accepts if  $M$  is odd; `LASTDIGITISEVEN` accepts if the last digit of  $M$  is even.

**PROBLEM ISODD**

- **Input:** An integer  $M$ , as an ASCII string in decimal notation.
- **Solution:** “yes” if  $M$  is odd, and “no” otherwise.

**PROBLEM LASTDIGITISEVEN**

- **Input:** An integer  $M$ , as an ASCII string in decimal notation.
- **Solution:** “yes” if the last digit of  $M$  is even, and “no” otherwise.

**Figure 7.1:** The computational problems ISODD and LASTDIGITISEVEN. There is an obvious reduction from ISODD to LASTDIGITISEVEN.

```
1 def isOdd(inString):
2     if int(inString) % 2 == 1:
3         return 'yes'
4     else:
5         return 'no'
```

```
1 def lastDigitIsEven(inString):
2     lastDigit = inString[-1] # “[−1]” is Python notation for last element
3     if lastDigit in '02468':
4         return 'yes'
5     else:
6         return 'no'
```

```
1 from lastDigitIsEven import lastDigitIsEven
2 def isOddViaReduction(inString):
3     inStringPlusOne = int(inString) + 1
4     return lastDigitIsEven(str(inStringPlusOne))
```

**Figure 7.2:** Three Python programs that demonstrate a reduction: `isOdd.py`, `lastDigitIsEven.py`, and `isOddViaReduction.py`. The first two programs solve ISODD and LASTDIGITISEVEN problems directly. The third program, `isOddViaReduction.py`, solves ISODD by reducing it to LASTDIGITISEVEN.

Both problems are easily solved, and the first two programs in figure 7.2 give direct implementations. However, it’s also possible to reduce ISODD to LASTDIGITISEVEN. To do this, we first observe that an integer  $M$  is odd if and only if the last digit of  $M+1$  is even. So, given an integer  $M$ , we can first add 1 to it, then ask whether the result’s last digit is even. That’s exactly what is done by the program `isOddViaReduction.py`, in the bottom panel of figure 7.2.

In this simple example, the reduction seems silly, because the direct implementation (`isOdd.py`) is so easy to write. But suspend your disbelief for a moment. Imagine you are a beginner programmer who doesn’t know how to directly check whether an integer is odd, but you have been provided with the program

**PROBLEM YESONSTRING**

- **Input:** A program  $P$  and string  $I$ .
- **Solution:** “yes” if  $P(I) = \text{“yes”}$ , and “no” otherwise.

**PROBLEM GAGAONSTRING**

- **Input:** A program  $P$  and string  $I$ .
- **Solution:** “yes” if  $P(I) = \text{“GAGA”}$ , and “no” otherwise.

**Figure 7.3: The computational problems GAGAONSTRING and YESONSTRING.**

YESONSTRING is undecidable (see claim 4.1, page 62). By showing that YESONSTRING reduces to GAGAONSTRING, we will conclude that GAGAONSTRING is also undecidable.

`lastDigitIsEven.py`. Then you can still solve the problem ISODD via the reduction to LASTDIGITISEVEN. In this sense, you have used a reduction to make ISODD into an easier problem—it’s been reduced to a problem that has already been solved.

This is the way that mathematicians, engineers, and applied computer scientists use reductions. We are given a problem  $F$  that we don’t know how to solve directly, but we know of some other problem  $G$  that *can* be solved. To solve  $F$ , we reduce it to  $G$  by giving a Python program that invokes the program for  $G$  whenever necessary. Thus,  $F$  is “easy” because we already know how to solve  $G$ .

## 7.2 A REDUCTION FOR HARDNESS

In contrast to most mathematicians, theoretical computer scientists often want to show that a problem is *hard*, rather than easy. In fact, in this chapter we will often want to show that a problem is so hard, it’s uncomputable. In this scenario, we are given a problem  $G$ , and we don’t know whether  $G$  is uncomputable. But we suspect  $G$  is uncomputable and would like to prove that it’s uncomputable. We also have, in our toolkit, some other problem  $F$  that has already been proved uncomputable. The plan is as follows: (a) show that  $F$  reduces to  $G$ ; (b) conclude that  $G$  is uncomputable. Why does this conclusion hold? We can prove it by contradiction:

1. Assume  $G$  is computable.
2. This means  $F$  is also computable, since we have a reduction from  $F$  to  $G$ .
3. This contradicts the fact that  $F$  is uncomputable.
4. Therefore, our initial assumption was incorrect. We conclude that  $G$  is uncomputable.

For our first concrete example of this, we will reduce the problem  $F = \text{YESONSTRING}$  to  $G = \text{GAGAONSTRING}$ . Both problems are described in figure 7.3. We have seen YESONSTRING before, on page 63. YESONSTRING is a decision problem taking two inputs  $(P, I)$ ; it returns “yes” if  $P(I) = \text{yes}$ , and

```

1  def alterYesToGAGA(inString):
2      (progString, newInString) = utils.DESS(inString)
3      val = universal(progString, newInString)
4      if val == 'yes':
5          return 'GAGA'
6      else:
7          return 'no'

from GAGAOnString import GAGAOnString # oracle function
2 def yesViaGAGA(progString, inString):
3     singleString = utils.ESS(progString, inString)
4     return GAGAOnString(rf('alterYesToGAGA.py'), singleString)

```

**Figure 7.4:** Two Python programs that demonstrate a reduction from YESONSTRING to GAGAONSTRING: `alterYesToGAGA.py` and `yesViaGAGA.py`.

“no” otherwise. From claim 4.1 on page 63, we already know that YESONSTRING is undecidable. Most of chapter 3 was devoted to proving this, via an elaborate proof that is summarized in figure 3.8 (page 38). We can take advantage of that hard work, and obtain a simple proof that a new problem is also undecidable, via a reduction from YESONSTRING. The new problem here is GAGAONSTRING, described in the bottom panel of figure 7.3. As you can see, this is a decision problem that asks whether a given program, with a given input, will produce the output “GAGA”. (Of course, there is nothing special about the string “GAGA”. The reduction below works equally well for any other fixed output string.)

We will later discover that GAGAONSTRING is undecidable. But for the moment, its status is unknown. Let’s put that issue on the back burner, and concentrate on finding a reduction from YESONSTRING to GAGAONSTRING.

Therefore, we assume that a program called `GAGAOnString.py` is available and solves GAGAONSTRING. We need to write a program that can solve YESONSTRING, using `GAGAOnString()` as a subroutine. To remind us that our new program employs this mythical `GAGAOnString()`, we will call our program `yesViaGAGA.py`. The basic idea is simple: `yesViaGAGA()` will receive two parameters  $(P, I)$ . As usual, these describe a program  $P$  and its input string  $I$ . We will use the ideas from section 6.4 to alter and simulate a new version of  $P$ . Specifically, we create a new program  $P'$  that returns “GAGA” if and only if  $P$  returns “yes”. Then, we can pass the combination  $(P', I)$  to `GAGAOnString()`. By construction, `GAGAOnString()` returns “yes” on  $(P', I)$  exactly when  $P$  returns “yes” on  $I$ . So we have a program that solves YESONSTRING.

The details of this reduction are shown in figure 7.4. The top program here, `alterYesToGAGA.py`, is extremely similar to the program `alterGAGA-toTATA.py` of figure 6.8 (page 111). It would be a good idea to reread section 6.4 now, acquiring a fresh understanding of `alterGAGAtoTATA.py`. With that understanding, it is almost trivial to understand the reduction of figure 7.4. The program `alterYesToGAGA.py` simulates its input, returning “GAGA” if the input produced “yes”. The reduction itself is performed in

`yesViaGAGA.py`, which invokes `GAGAOnString()` on the altered version of the input.

To summarize, `yesViaGAGA.py` is a reduction from YESONSTRING to GAGAONSTRING. Therefore, if we can solve GAGAONSTRING, we can also solve YESONSTRING. What does this tell us about the decidability of GAGAONSTRING? In fact, we can conclude that GAGAONSTRING is undecidable. To see this, we just rerun the proof by contradiction given earlier (page 118), this time using YESONSTRING and GAGAONSTRING instead of  $F$  and  $G$ :

1. Assume GAGAONSTRING is decidable.
2. This means YESONSTRING is also decidable, since we found a reduction from YESONSTRING to GAGAONSTRING.
3. This contradicts the fact that YESONSTRING is undecidable.
4. Therefore, our initial assumption was incorrect and we conclude that GAGAONSTRING is undecidable.

We will see more concrete examples of uncomputability proofs after taking care of some formal definitions. And don't be concerned if the above reduction seemed long and elaborate: section 7.9 will revisit this issue, providing several techniques for producing brief, yet rigorous, reductions.

### 7.3 FORMAL DEFINITION OF TURING REDUCTION

The reductions we've examined so far are technically known as “Turing reductions,” and are formally defined as follows.

**Definition of a Turing reduction.** Let  $F$  and  $G$  be computational problems. We say that  $F$  has a *Turing reduction* to  $G$  if we can write a Python program solving  $F$ , after first assuming the existence of a program solving  $G$ .

**Notation for Turing reductions.** If  $F$  has a Turing reduction to  $G$ , we write

$$F \leq_T G.$$

This definition is more subtle than it might seem at first glance; the next four subsections investigate some of the subtleties in more detail.

#### Why “Turing” reduction?

The phrase “Turing reduction” distinguishes this type of reduction from other types of reductions (especially polynomial-time mapping reductions, which we will meet in chapter 13). But for the time being, Turing reductions are the only type of reduction that we know about. Therefore, until chapter 13, we'll be deliberately sloppy: “Turing reduction” and “reduction” will be used interchangeably.

## Oracle programs

In the definition of a Turing reduction, what does it mean to assume the “existence of a program solving  $G$ ”? The answer is that we assume there is a Python program, say `G.py`, that computes  $G$ . At the top of our program for  $F$ , say `F.py`, we can include the code “`from G import G`”—see line 1 of `yesViaGAGA.py` (figure 7.4, bottom panel) for an example of this. Anywhere else in `F.py`, we can invoke `G()`, and assume that it terminates on any input, returning a correct solution. This is done, for example, in line 4 of `yesViaGAGA.py`.

The program  $G$  is called an *oracle program*, or *oracle function*. (This has nothing to do with the computer company Oracle. The word “oracle” here is meant to remind us of the ancient Greek oracles, who could supposedly answer any question.) Note that we are allowed to assume the existence of an oracle program for  $G$  even if  $G$  is uncomputable—for the purposes of this definition, we temporarily pretend that  $G$  is computable, regardless of whether it actually is.

Most textbooks define Turing reductions in terms of Turing machines, rather than Python programs. To do this, they first define an *oracle Turing machine*, which has the magical ability to solve a given computational problem. Our definition of oracle programs is analogous to oracle Turing machines, and leads to an equivalent definition of Turing reduction.

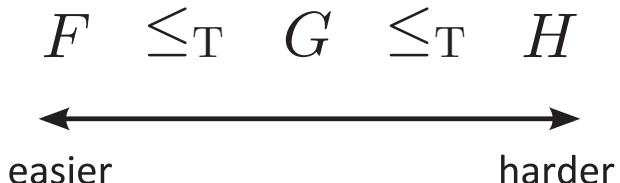
## Why is $\leq_T$ used to denote a Turing reduction?

The “T” in  $F \leq_T G$  reminds us that this is a *Turing reduction*. But that’s the easy part. Why is “ $\leq$ ” relevant here?

A short answer is that the “ $\leq$ ” in  $F \leq_T G$  reminds us that  $F$  is a “lesser” problem than  $G$ . Problem  $F$  is, in some sense, as “easy” as  $G$ —and perhaps even easier. This notion of “easy” can be counterintuitive when you first see it. We won’t be giving a formal definition of “easy,” but let’s try to gain some intuition about it. Suppose we know  $F \leq_T G$ , but we know nothing else about the difficulty of solving  $F$  or  $G$ . If, at some future time, we discover a method of solving  $G$ , then we immediately know how to solve  $F$  too (because  $F \leq_T G$ ). Thus, we will never be stuck in a situation where we can solve  $G$  but not  $F$ . On the other hand, it’s possible we could discover a method of solving  $F$  in the future, and that would tell us nothing about how to solve  $G$ . Hence, we could be stuck in a situation where  $F$  can be solved but  $G$  cannot. This is what it means for  $F$  to be as “easy” as  $G$ , and perhaps even easier. Equivalently, we can say that  $F$  is “no harder than”  $G$ . Figure 7.5 demonstrates this idea graphically.

## Beware the true meaning of “reduction”

The word “reduction” can be dangerously misleading, especially when combined with the  $\leq_T$  notation. Even experienced computer science researchers get confused by this, so watch out! The source of confusion is that, in ordinary English, when we “reduce” something, that thing gets smaller. For example, if we start with the number  $F = 27$ , and *reduce* it by 6, we get  $G = F - 6 = 21$ , so  $G \leq F$ . But



**Figure 7.5:** Informally,  $F \leq_T G$  means that  $F$  is “easier” or “no harder than”  $G$ . Therefore, we can think of the  $\leq_T$  symbol as ordering problems according to increasing hardness, from left to right.

if we start with a computational problem  $F$ , and (*Turing*) *reduce* it to  $G$ , we get  $F \leq_T G$ . This is the opposite of what we would get if  $F$  and  $G$  were numbers. There is no easy way out of this mess: as a computer science student, you just need to practice working with the definition until your brain instinctively knows that the following statements are all equivalent:

- $F$  reduces to  $G$ .
- $F \leq_T G$ .
- $F$  is no harder than  $G$ .
- $F$  can be solved, assuming  $G$  can be solved.

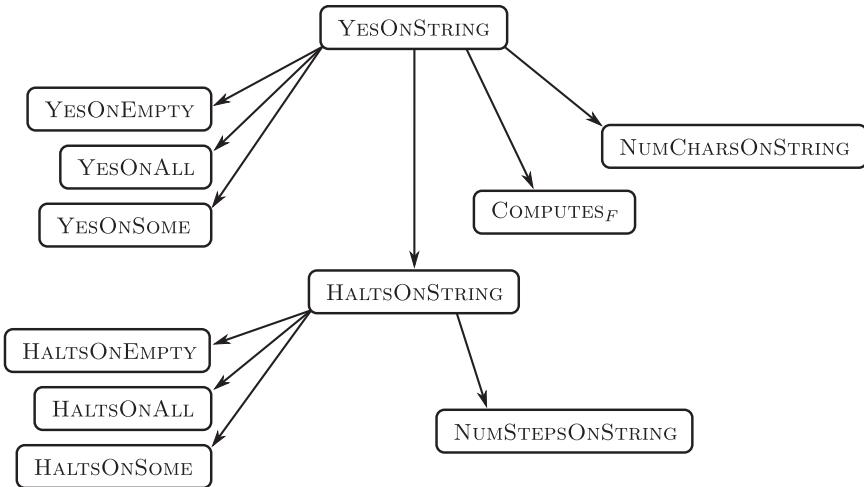
As one minor consolation for the confusion over “reduce,” it’s worth pointing out that our technical meaning of “reduce” in computer science has a counterpart in cooking, as you can see from the Julia Child quotation at the start of this chapter. Suppose you have a pot full of thin, watery soup boiling on the stove. If you leave this soup boiling for long enough, most of the water in the soup will evaporate and you could be left with a thick, tasty sauce. A chef would say you have “reduced” the watery soup to a tasty sauce. In some sense, the sauce is the essence of the soup. Similarly, if  $F \leq_T G$ , then  $G$  is the essence of  $F$ . In other words,  $G$  is the problem that is left over if we “boil down” or “reduce”  $F$ .

## 7.4 PROPERTIES OF TURING REDUCTIONS

Let’s now prove some of the elementary properties of Turing reductions that will be needed later. First, *transitivity* will let us build up chains of reductions. This idea is intuitively obvious from figure 7.5, but let’s also work through a rigorous statement and proof:

**Claim 7.1.** Turing-reducibility is transitive. That is, if  $F \leq_T G$  and  $G \leq_T H$ , then  $F \leq_T H$ .

*Proof of the claim.* This follows almost immediately from the definition of a Turing reduction: “ $F \leq_T G$ ” means we have a program `F.py` that solves  $F$ , assuming the existence of `G.py` that solves  $G$ . And “ $G \leq_T H$ ” means we have `G.py`, assuming the existence of `H.py` that solves  $H$ . Hence, assuming only the existence of `H.py`, we can obtain first `G.py` and then `F.py`, proving that  $F \leq_T H$ .  $\square$



**Figure 7.6: A summary of the reductions proved in this chapter.** Each arrow signifies a reduction from one computational problem to another. Because the root of this tree (YESONSTRING) is uncomputable, we will conclude that all the other problems in the tree are also uncomputable.

Next, we state a formal claim that justifies the use of  $\leq_T$  to represent the relative “easiness” of solving problems, at least in terms of computability. Again, the intuition is clear from figure 7.5, but we need a formal proof too:

**Claim 7.2.** (Propagation of computability). Suppose  $F \leq_T G$ . Then,

- (a) if  $G$  is computable, then  $F$  is computable;
- (b) if  $F$  is uncomputable, then  $G$  is uncomputable.

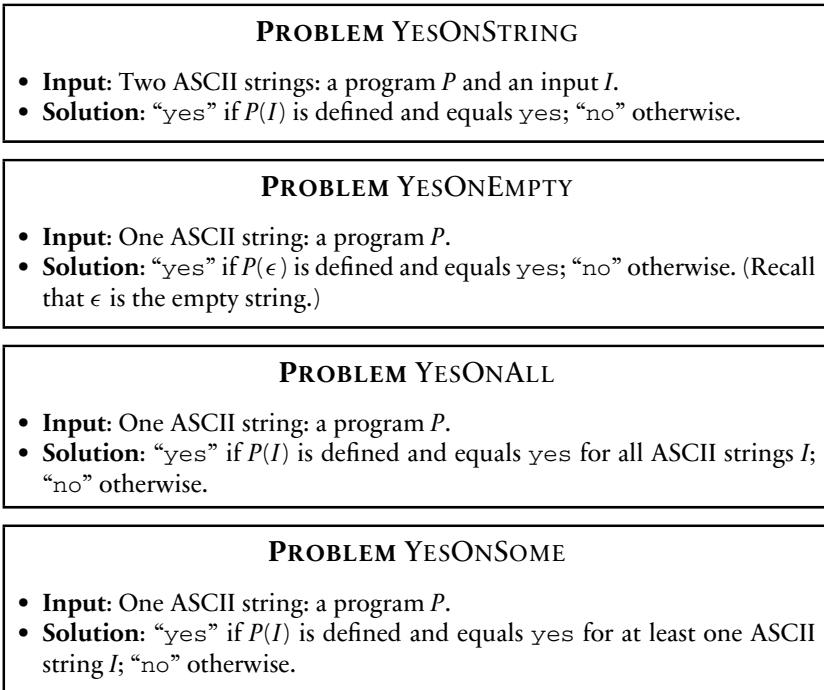
*Proof of the claim.* Part (a) is essentially a restatement of the definition of a Turing reduction. Part (b) has been proved twice already (pages 118 and 120), but let’s do it one more time: Assume  $G$  is computable; then  $F$  is computable. This contradicts the fact that  $F$  is uncomputable; hence  $G$  is uncomputable.  $\square$

## 7.5 AN ABUNDANCE OF UNCOMPUTABLE PROBLEMS

In the previous chapter, we saw that universal computation is surprisingly common. In this chapter, we are about to discover that uncomputability is also surprisingly common. In fact, the rest of this chapter will consist of a long laundry list of uncomputable problems, followed by a theorem (Rice’s theorem) that describes an infinite set of important undecidable problems. Figure 7.6 summarizes our approach.

### The variants of YESONSTRING

The first three new entries on our laundry list of uncomputable problems are variants of YESONSTRING. These computational problems are described



**Figure 7.7: Four variants of the computational problem YESONSTRING.** All four are undecidable.

in figure 7.7, along with the original YESONSTRING for easy comparison. Whereas YESONSTRING takes two parameters  $(P, I)$ , its three variants take only the single parameter  $P$ , describing the program. YESONEMPTY asks whether  $P$  outputs “yes” when given the empty string as input. YESONALL asks whether  $P$  outputs “yes” on all possible input strings. And YESONSOME asks whether  $P$  outputs “yes” on at least one input string.

Each of the three new problems can be proved undecidable by reducing from YESONSTRING. The key is `ignoreInput.py`, discussed on page 112. Reread that section now, and make sure you can complete exercise 6.5 (page 114) before continuing. The source code of `ignoreInput.py` is presented again for easy reference in the top panel of figure 7.8. The fundamental idea is that `ignoreInput.py` performs a preset computation based on the contents of the two special files `progString.txt` and `inString.txt`. The input string is irrelevant: for any given contents of the special files `progString.txt` and `inString.txt`, the program `ignoreInput.py` produces the same output on all inputs.

This yields easy reductions from YESONSTRING to YESONEMPTY, YESONALL, and YESONSOME, as shown in the bottom three panels of figure 7.8. Let’s examine just the last one. We are given an oracle that solves YESONSOME, and now we wish to know whether  $\widehat{P}(\widehat{I}) = \text{“yes”}$  for some particular  $\widehat{P}$  and  $\widehat{I}$ . As suggested above,  $\widehat{P}$  and  $\widehat{I}$  are stored in `progString.txt` and `inString.txt`

```

1 def ignoreInput(inString):
2     progString = rf('progString.txt')
3     newInString = rf('inString.txt')
4     return universal(progString, newInString)

from yesOnEmpty import yesOnEmpty # oracle function
1 def yesViaEmpty(progString, inString):
2     utils.writeFile('progString.txt', progString)
3     utils.writeFile('inString.txt', inString)
4     return yesOnEmpty(rf('ignoreInput.py'))

from yesOnAll import yesOnAll # oracle function
1 def yesViaAll(progString, inString):
2     utils.writeFile('progString.txt', progString)
3     utils.writeFile('inString.txt', inString)
4     return yesOnAll(rf('ignoreInput.py'))

from yesOnSome import yesOnSome # oracle function
1 def yesViaSome(progString, inString):
2     utils.writeFile('progString.txt', progString)
3     utils.writeFile('inString.txt', inString)
4     return yesOnSome(rf('ignoreInput.py'))

```

**Figure 7.8:** The reductions from YESONSTRING to YESONEMPTY, YESONALL, and YESONSOME. All three reductions rely on `ignoreInput.py` (*top panel*), which ignores its input string. The reduction programs themselves (`yesViaEmpty.py`, `yesViaAll.py`, `yesViaSome.py`) are identical except for the oracle function used.

respectively. Now we are in a situation where `ignoreInput.py` returns exactly the same thing on all inputs—the returned value is always  $\hat{P}(\hat{I})$ . In particular, if `ignoreInput.py` returns “yes” on some input, then  $\hat{P}(\hat{I}) = \text{“yes”}$ . So we can ask our oracle whether `ignoreInput.py` returns “yes” on some input. The answer is also the solution to our original question of whether  $\hat{P}(\hat{I}) = \text{“yes”}$ .

An almost identical argument works for the reductions to YESONEMPTY and YESONALL, since the output of `ignoreInput.py` is the same for the empty string and for all strings. Hence, we have the following conclusion:

**Claim 7.3.** YESONSTRING, YESONEMPTY, YESONALL, and YESONSOME are undecidable.

*Proof of the claim.* We already know YESONSTRING is undecidable. The discussion above, together with the Python programs in figure 7.8, proves that YESONSTRING  $\leq_T$  YESONEMPTY, YESONSTRING  $\leq_T$  YESONALL, and YESONSTRING  $\leq_T$  YESONSOME. Applying the propagation of computability claim (claim 7.2, page 123) immediately yields the result.  $\square$

It's worth noting that there is nothing special about the value "yes" in these problems. Similar questions about other output values are also undecidable. For example, we have already proved that GAGAONSTRING is undecidable (page 119). Other problems that can easily be proved undecidable by similar techniques include GAGAONEMPTY, YESONGAGA, NOONSTRING, NOONSOME, EMPTYONSTRING, EMPTYONALL, and EMPTYONEMPTY. The formal definitions of these problems should be obvious, and are omitted.

## The halting problem and its variants

Given any program  $P$  and input  $I$ , we can ask an interesting and important question: Does  $P$  terminate (or *halt*) on input  $I$ ? This question, and various questions related to it, are known as the *halting problem*. There are two main reasons for studying the halting problem. First, it is of practical importance. When we write or use a piece of software, we would often like to know that it will finish any job that it starts. This is not always true: some types of software (e.g., an operating system or a Web server) are intended to run indefinitely. But other types of software (e.g., a payroll computation that generates paychecks for a company's employees) should always terminate. Therefore, it would be beneficial to solve the halting problem for many kinds of software.

The second reason for studying the halting problem is its historical significance. In his seminal 1936 "On computable numbers" paper, Alan Turing focused on a variant of the halting problem. Thus, questions about whether computer programs halt or loop are intimately related to the birth of theoretical computer science. (For more details, see chapter 15.)

On the other hand, it's worth pointing out that the significance of the halting problem is usually overstated. The halting problem is just one of many, many problems that are all undecidable and reducible to each other. In this sense, the halting problem is no more or less significant than other fundamental problems such as YESONSTRING and YESONALL.

Before discussing the halting problem in detail, we need a careful definition of what it means for a program to halt. For Turing machines, the definition is obvious: a machine *halts* if it enters one of the halting states  $q_{\text{accept}}$ ,  $q_{\text{reject}}$ ,  $q_{\text{halt}}$ . For Python programs, we say program  $P$  *halts* if the output  $P(I)$  is defined, according to the formal definition on page 24. Note that a program could fail to halt for several different reasons: it could enter an infinite loop, but it could also throw an exception or return an object that isn't an ASCII string. These latter two cases may seem counterintuitive, since the program does stop running when it throws an exception or returns a non-string. In these two cases, it's best to think of the program being "stuck": it is no longer continuing to execute, but it is unable to proceed and terminate successfully. In this sense, the program has not "halted."

Let's proceed with a detailed investigation of the halting problem. Figure 7.9 gives four interesting variants of the problem: HALTSONSTRING, HALTSONEMPTY, HALTSONALL, and HALTSONSOME. The definitions should be obvious, but for complete clarity figure 7.9 gives all the details. Note that there is no widely agreed definition of *the* halting problem. Many textbooks define one of HALTSONSTRING or HALTSONEMPTY as the halting problem. Here, we won't

**PROBLEM HALTSONSTRING**

- **Input:** Two ASCII strings: a program  $P$  and an input  $I$ .
- **Solution:** “yes” if  $P$  halts on input  $I$ , and “no” otherwise.

**PROBLEM HALTSONEMPTY**

- **Input:** One ASCII string: a program  $P$ .
- **Solution:** “yes” if  $P$  halts on empty input  $\epsilon$ , and “no” otherwise.

**PROBLEM HALTSONALL**

- **Input:** One ASCII string: a program  $P$ .
- **Solution:** “yes” if  $P$  halts on all inputs  $I$ , and “no” otherwise.

**PROBLEM HALTSONSOME**

- **Input:** One ASCII string: a program  $P$ .
- **Solution:** “yes” if  $P$  halts on at least one input  $I$ , and “no” otherwise.

**Figure 7.9:** Four variants of the halting problem. All four are undecidable.

```

from universal import universal
2 def alterYesToHalt(inString):
    (progString, newInString) = utils.DESS(inString)
4    val = universal(progString, newInString)
    if val == 'yes':
6        # return value is irrelevant, since returning any string halts
        return 'halted'
8    else:
        # deliberately enter infinite loop
10       utils.loop()

```

```

from haltsOnString import haltsOnString # oracle function
2 def yesViaHalts(progString, inString):
    singleStr = utils.ESS(progString, inString)
4    return haltsOnString(rf('alterYesToHalt.py'), singleStr)

```

**Figure 7.10:** Two Python programs that demonstrate a reduction from YESONSTRING to HALTSONSTRING: `alterYesToHalt.py` and `yesViaHalts.py`.

single out any particular variant, so the phrase “halting problem” will remain deliberately vague, referring to any or all of these four problems.

Figure 7.10 shows how to reduce YESONSTRING to HALTSONSTRING. The key idea is in the top panel, `alterYesToHalt.py`. Note the similarities and differences of this program compared to `alterYesToGAGA.py` (figure 7.4, page 119). In the earlier reduction, we transformed a program  $P$  returning yes/not-yes into a program  $P'$  returning GAGA/not-GAGA. In this reduction,

we transform  $P$  returning yes/not-yes into a program  $P'$  that halts/doesn't-halt. Specifically,

- if  $P$  returns “yes” on input  $I$ , then  $P'$  halts on input  $I$ : this is implemented at line 7 of `alterYesToHalt.py`—note that the actual value returned (“halted”) is irrelevant, since all we need is for the program to terminate successfully;
- if  $P$  doesn't return “yes” on input  $I$ , then  $P'$  loops on input  $I$ : this is implemented at line 10 of `alterYesToHalt.py`.

Program `yesViaHalts.py` (bottom panel of figure 7.10) implements the reduction itself, using the now-familiar technique of passing the altered program on to the `haltsOnString()` oracle function. Thus we can conclude that  $\text{YESONSTRING} \leq_T \text{HALTSONSTRING}$ . The undecidability of  $\text{HALTSONSTRING}$  follows immediately. And as the following claim shows, the other variants of the halting problem are also undecidable.

**Claim 7.4.** All four variants of the halting problem are undecidable. (The four variants are  $\text{HALTSONSTRING}$ ,  $\text{HALTSONEMPTY}$ ,  $\text{HALTSONALL}$ , and  $\text{HALTSONSOME}$ —see figure 7.9, page 127.)

*Proof of the claim.* The immediately preceding discussion proved that  $\text{YESONSTRING} \leq_T \text{HALTSONSTRING}$ . We already know  $\text{YESONSTRING}$  is undecidable. Applying the propagation of computability claim (claim 7.2, page 123), we thus have that  $\text{HALTSONSTRING}$  is undecidable. We can then propagate the undecidability of  $\text{HALTSONSTRING}$  to the other three variants using reductions from  $\text{HALTSONSTRING}$ . These three reductions are essentially identical to the reductions from  $\text{YESONSTRING}$  to its three variants (see figure 7.8, page 125). The only difference is that new oracle functions must be used (e.g., the reduction to  $\text{HALTSONEMPTY}$  uses the `haltsOnEmpty()` oracle function).  $\square$

## Uncomputable problems that aren't decision problems

It's been mentioned in a few places that the theory of computer science tends to concentrate on decision problems, whereas in practice, computer programs often solve nondecision problems. So far in this chapter, we have followed the typical theoretical approach, proving the uncomputability of decision problems such as  $\text{YESONSTRING}$  and  $\text{HALTSONEMPTY}$ . Let us now return to some more practical territory, and prove the uncomputability of two *nondecision* problems:  $\text{NUMCHARSONSTRING}$  and  $\text{NUMSTEPSONSTRING}$ .

$\text{NUMCHARSONSTRING}$  is described formally in figure 7.11. Informally, it tells us the number of characters (i.e., the *length*) of a program's output. The solution is a number (e.g., “4325”), so this is indeed a nondecision problem. But we can still prove it uncomputable by reducing *from* a decision problem. In particular, let's reduce  $\text{YESONSTRING}$  to  $\text{NUMCHARSONSTRING}$ . Given  $(P, I)$ , we alter  $P$  to a new program  $P'$  that returns a string of length 3 if and only if  $P$  returns “yes”. (There is nothing special about the number 3 here; we choose it only because the

### PROBLEM NUMCHARSONSTRING

- **Input:** Two ASCII strings: a program  $P$  and an input  $I$ .
- **Solution:** The length of  $P(I)$ , if  $P(I)$  is defined, or “no” otherwise.

**Figure 7.11:** A first example of an uncomputable problem that is not a decision problem: NUMCHARSONSTRING.

```

1 from universal import universal
2 def alterYesToNumChars(inString):
3     (progString, newInString) = utils.DESS(inString)
4     val = universal(progString, newInString)
5     if val == 'yes':
6         # return a string with three characters
7         return 'xxx'
8     else:
9         # return a string with two characters
10        return 'xx'
```

```

1 from numCharsOnString import numCharsOnString # oracle function
2 def yesViaNumChars(progString, inString):
3     singleString = utils.ESS(progString, inString)
4     val = numCharsOnString(rf('alterYesToNumChars.py'), \
5                           singleString)
6     if val == '3':
7         return 'yes'
8     else:
9         return 'no'
```

**Figure 7.12:** Two Python programs that demonstrate a reduction from YESONSTRING to NUMCHARSONSTRING: alterYesToNumChars.py and yesViaNumChars.py.

string “yes” has 3 characters.) Figure 7.12 gives details of the reduction, which should look very familiar. We won’t bother giving any more formal proof than this.

Our next example of an uncomputable, nondecision problem is interesting and important, because it provides a link to the second half of this book. The problem NUMSTEPSONSTRING, described formally in figure 7.13, asks about the *running time* of a program. That is, given program  $P$  and input  $I$ , how long will it take to compute  $P(I)$ ? The second half of this book (chapter 10 onwards) is devoted to this question, and investigates how to classify many programs according to their running times. However, we now show that there is no systematic method of solving this problem that works for all programs  $P$  on all inputs  $I$ , because NUMSTEPSONSTRING is uncomputable.

The running time of a program is measured in “computational steps.” For Turing machines, the definition of “computational step” is obvious: each application of the transition function comprises one step. But for Python programs,

### PROBLEM NUMSTEPSONSTRING

- **Input:** Two ASCII strings: a program  $P$  and an input  $I$ .
- **Solution:** The number of computational steps used in the computation of  $P(I)$ , if  $P(I)$  is defined, or “no” otherwise.

(Note: The notion of “computational step” is not defined formally until chapter 10. For now, assume the number of steps is proportional to the actual time needed to run the program.)

**Figure 7.13:** Our second example of an uncomputable problem that is not a decision problem: NUMSTEPSONSTRING.

```

1 from numStepsOnString import numStepsOnString # oracle function
2 def haltsViaNumSteps(progString, inString):
3     val = numStepsOnString(progString, inString)
4     if val == 'no':
5         return 'no'
6     else:
7         return 'yes'
```

**Figure 7.14:** Python program `haltsViaNumSteps.py`, which performs a reduction from HALTSSTRING to NUMSTEPSSTRING.

the definition is less clear. In chapter 10, we will investigate the definition of computational steps more carefully. For now, you can take it on faith that running times can be defined for Python programs, and that they correspond roughly to the amount of actual time required to execute the program on the reference computer system.

There is a simple reduction from HALTSSTRING to NUMSTEPSSTRING. Notice that NUMSTEPSSTRING returns “no” whenever  $P(I)$  is undefined, so we don’t even need to alter the input program  $P$ . We can send  $(P, I)$  directly to NUMSTEPSSTRING, and conclude that  $P$  halts if and only if the result isn’t “no”. Program `haltsViaNumSteps.py`, in figure 7.14, gives the details.

## 7.6 EVEN MORE UNCOMPUTABLE PROBLEMS

Our laundry list of uncomputable problems has grown rather long already, but it’s about to get much longer: we are about to discover an infinite family of problems that are all uncomputable. This family is easiest to explain with a concrete example.

So, consider the problem IS EVEN shown in figure 7.15. This problem decides whether a given integer  $M$  is even. We can write a one-line Python program to solve this, so IS EVEN is decidable. However, let’s now consider a much more abstract counterpart to this problem: COMPUTESISEVEN takes a program  $P$  and tells us whether or not  $P$  computes IS EVEN.

**PROBLEM IS EVEN**

- **Input:** An integer  $M$ , as an ASCII string in decimal notation.
- **Solution:** “yes” if  $M$  is even, and “no” otherwise.

**PROBLEM COMPUTEIS EVEN**

- **Input:** A program  $P$ , as an ASCII string.
- **Solution:** “yes” if  $P$  computes IS EVEN, and “no” otherwise.

**Figure 7.15:** The computational problems IS EVEN and COMPUTEIS EVEN. IS EVEN is decidable, and COMPUTEIS EVEN is undecidable. This is a particular example of a general pattern: if  $F$  is a computable problem, then COMPUTE $_F$  is uncomputable.

```

1  def P1(inString):
2      n = int(inString)
3      if n%2==0: return 'yes'
4      else: return 'no'
5
6  def P2(inString):
7      if inString[-1] in '02468': return 'yes'
8      else: return 'no'
9
10 def P3(inString):
11     n = int(inString)
12     if (2*n+1)%4==1: return 'yes'
13     else: return 'no'
14
15 def P4(inString):
16     n = int(inString)
17     if (3*n+1)%5==1: return 'yes'
18     else: return 'no'
```

**Figure 7.16:** Four instances of the problem COMPUTEIS EVEN. The first three are positive instances; the last is a negative instance.

This problem may sound easy, but it’s not. Consider the four instances `P1.py`, `P2.py`, `P3.py`, `P4.py` shown in figure 7.16. Which ones are the positive instances? Spend a few minutes trying to solve this for yourself before reading on.

The answer is that the first three are positive instances, and the last one is a negative instance. And of course, we could make much more complicated examples. The important point here is that even very simple problems (like IS EVEN) can be solved in arbitrarily complicated ways. So, it turns out that COMPUTEIS EVEN is undecidable. More generally, it turns out that there is no way to determine whether a given program computes a given function.

### PROBLEM COMPUTES<sub>F</sub>

This problem is defined with respect to a computational problem  $F$ .

- **Input:** A program  $P$ .
- **Solution:** “yes” if  $P$  computes  $F$ , and “no” otherwise.

**Figure 7.17:** The computational problem COMPUTES<sub>F</sub>.

### The computational problem COMPUTES<sub>F</sub>

We can formalize this by first defining the problem COMPUTES<sub>F</sub> (see figure 7.17). This is not really a single problem, but rather an infinite *family* of problems, parameterized by  $F$ . Here,  $F$  is itself a computational problem, such as IS EVEN in the example above. COMPUTES<sub>F</sub> takes a program  $P$  as input, and tells us whether or not  $P$  computes  $F$ . It’s important to realize that  $F$  is not part of the input for COMPUTES<sub>F</sub>. Instead,  $F$  is part of the definition of any particular problem in this family. For example, when  $F = \text{IS EVEN}$ , COMPUTES<sub>F</sub> = COMPUTEIS EVEN.

The next claim uses a reduction to prove that COMPUTES<sub>F</sub> is uncomputable whenever  $F$  is computable.

**Claim 7.5.** (Uncomputability of COMPUTES<sub>F</sub>). Let  $F$  be a computable problem. Then COMPUTES<sub>F</sub> is uncomputable.

*Proof of the claim.* Figure 7.18 shows the details of a reduction from YESONSTRING to COMPUTES<sub>F</sub>. The top panel, `alterYesToComputesF.py`, shows how to transform an instance of YESONSTRING into an instance of COMPUTES<sub>F</sub>. This program makes use of both the specified computable problem  $F$ , and any other computable problem  $G$ . The problem  $G$  must be different to  $F$ , but that is the only requirement. It could be, say, a constant function that always returns the empty string. As with several of our earlier reductions (see page 125, for example), the instance  $(P, I)$  of YESONSTRING is saved into some text files. So we think of  $(P, I)$  as a fixed part of the `alterYesToComputesF.py` program, read in at lines 5–6. At line 7, the value of  $P(I)$  is obtained via simulation. If this value is “yes” (so  $(P, I)$  was a positive instance of YESONSTRING), the program computes  $F$  at line 9. Otherwise the program computes  $G$  (line 11). To summarize, `alterYesToComputesF.py` computes  $F$  for positive instances, and  $G$  otherwise.

The reduction is completed by `yesViaComputesF.py`, in the bottom panel of figure 7.18. After writing out the instance  $(P, I)$  to the relevant files (lines 3–4), we ask whether or not `alterYesToComputesF.py` computes  $F$  (line 5). By construction, the answer is “yes” if and only if  $(P, I)$  is a positive instance of YESONSTRING. Hence, the reduction is complete.  $\square$

This is a very powerful result, since it gives us a feeling for the seething mass of uncomputable problems that lurk in the computational universe. Specifically, for every computable problem  $F$ , there’s a corresponding uncomputable problem

```

1  from universal import universal
2  from F import F
3  from G import G # G is any computable function different to F
4  def alterYesToComputesF(inString):
5      progString = rf('progString.txt')
6      newInString = rf('inString.txt')
7      val = universal(progString, newInString)
8      if val == 'yes':
9          return F(inString)
10     else:
11         return G(inString)

from computesF import computesF # oracle function
2 def yesViaComputesF(progString, inString):
3     utils.writeFile('progString.txt', progString)
4     utils.writeFile('inString.txt', inString)
5     val = computesF(rf('alterYesToComputesF.py'))
6     if val == 'yes':
7         return 'yes'
8     else:
9         return 'no'

```

**Figure 7.18:** Two Python programs that demonstrate a reduction from YESONSTRING to COMPUTES<sub>F</sub>: alterYesToComputesF.py and yesViaComputesF.py.

COMPUTES<sub>F</sub>. Given the importance of this result, it's worth making a few more observations about it:

- Is it really necessary for the problem  $F$  to be computable? Wouldn't COMPUTES<sub>F</sub> be even more difficult to solve if  $F$  were uncomputable? In fact, this is a trick question. If  $F$  is uncomputable, then the solution to COMPUTES<sub>F</sub> is always “no”—because we know that no program can compute  $F$ . So COMPUTES<sub>F</sub> is the constant function “no”, which is certainly computable. Therefore, the restriction that  $F$  be computable is definitely required.
- The claim remains true when  $F$  is replaced by a set  $S$  of problems. We define the problem COMPUTESONEOF<sub>S</sub> to answer the question, does the input program  $P$  compute one of the problems in the set  $S$ ? Then it turns out that COMPUTESONEOF<sub>S</sub> is also undecidable, provided that one technical requirement is met. The requirement is that  $S$  should contain at least one computable problem  $F$ , and should exclude at least one computable problem  $G$ . When this holds, it's easy to check that the same proof given above works for reducing YESONSTRING to COMPUTESONEOF<sub>S</sub>.
- Now consider restricting this generalization to decision problems. We get another problem, DECIDESONEOF<sub>S</sub>, that is also undecidable (provided  $S$  contains a decidable language, and excludes a decidable language). This result is usually known as *Rice's theorem*. Another variant uses languages

that are recognizable rather than decidable. This gives us the problem  $\text{RECOGNIZESONEOF}_S$ , which is again undecidable (provided  $S$  contains a recognizable language, and excludes a recognizable language).

### Rice's theorem

We summarize the above results in the following claim.

**Claim 7.6.** (Variants and generalizations of Rice's theorem). Let  $F$  be a computable problem. Let  $S$  be a set of problems that contains at least one computable problem and excludes at least one computable problem. Similarly, let  $S'$  be a set of decision problems that contains at least one decidable problem and excludes at least one decidable problem. Then the following are all undecidable:  $\text{COMPUTES}_F$ ,  $\text{COMPUTESONEOF}_S$ ,  $\text{DECIDESONEOF}_{S'}$ ,  $\text{RECOGNIZESONEOF}_{S'}$ .

## 7.7 UNCOMPUTABLE PROBLEMS THAT AREN'T ABOUT PROGRAMS

As promised, we now have our infinite laundry list of uncomputable problems. So hopefully you're convinced by now that uncomputability is not a rare or obscure phenomenon. In contrast, the “uncomputability of  $\text{COMPUTES}_F$ ” (claim 7.5, page 132) and its generalizations suggest that essentially any question about what a program outputs is uncomputable.

Nevertheless, all of our uncomputable problems are *questions about programs*. Specifically, every one of our uncomputable problems so far takes as input a program  $P$  (and sometimes an additional string  $I$ ). But is this an essential aspect of uncomputability? Are there uncomputable problems whose inputs are *not* computer programs? There is no simple answer to this question, as we will now see.

On the surface, the answer appears to be yes—there are uncomputable problems whose inputs aren't programs. We won't study the details of these problems, but two of the most famous ones are described briefly below. Don't worry if you're not familiar with the terminology in these descriptions. The objective is not to understand these problems in detail, but rather to appreciate that their definitions do not involve computer programs directly.

- **Diophantine equations.** The input is a list of polynomial equations with integer coefficients. (Example:  $2x^2 + 5xy^3 = 7$ ,  $x - x^2y = 0$ ,  $xyz^2 + 2y^5z^4 = 3$ .) The output is “yes” if there is an integer solution, and “no” otherwise. (A solution to the previous example is  $x=1$ ,  $y=1$ ,  $z=-1$ .) This problem is known as “Hilbert's 10th problem,” because it was number 10 on a list of important unsolved mathematical problems proposed by the great mathematician David Hilbert in 1900. The problem was not proved undecidable until 70 years later.
- **Post correspondence problem.** The input is a set of *tiles*. Each tile has two ASCII strings written on it, one above the other. For example, 

abc
yz

 is a

single tile. So, an example of the problem input would be

$$\left\{ \begin{array}{|c|} \hline YZ \\ \hline Y \\ \hline \end{array}, \begin{array}{|c|} \hline X \\ \hline ZX \\ \hline \end{array}, \begin{array}{|c|} \hline ZZ \\ \hline Z \\ \hline \end{array}, \begin{array}{|c|} \hline WW \\ \hline W \\ \hline \end{array} \right\}.$$

We imagine that we have an infinite supply of each kind of tile in the set. The output of the problem is “yes” if some tiles from the supply can be arranged in a sequence (with repetitions permitted) so that the strings on the top and bottom of the sequence are identical. If no such arrangement is possible, the output is “no”. The example above is a positive instance, because the following sequence produces the string “yzxzzxyzx” on the top and bottom:

$$\begin{array}{|c|c|c|c|c|c|} \hline YZ & X & ZZ & X & YZ & X \\ \hline Y & ZX & Z & ZX & Y & ZX \\ \hline \end{array}.$$

Notice how the tile-type  $\begin{array}{|c|c|} \hline X \\ \hline ZX \\ \hline \end{array}$  was used three times, whereas  $\begin{array}{|c|c|} \hline WW \\ \hline W \\ \hline \end{array}$  was not used at all. In contrast, the following input is a negative instance:

$$\left\{ \begin{array}{|c|c|} \hline YZ \\ \hline X \\ \hline \end{array}, \begin{array}{|c|c|} \hline Y \\ \hline ZX \\ \hline \end{array} \right\}.$$

This instance is negative because the top line always contains at least one  $y$ , but no tile has a  $y$  on the bottom line.

These two examples were easy instances, but it turns out that the Post correspondence problem is, in general, undecidable.

So, at first glance, the two problems above are examples of undecidable problems that don’t involve computer programs. If we look more closely, however, the situation is less clear. It turns out that there are computer programs hiding inside the inputs to the above problems—but we won’t try to understand how or why. If you’re curious for more details, chapter 7 of Moore and Mertens is strongly recommended. But for now let’s just accept that, given a decision program  $P$ , one can devise a Diophantine equation that simulates  $P$ . Similarly, one can devise a set of tiles that simulates  $P$ . In other words, Diophantine equations and tile sets are further examples of universal computation. We can add these two examples to our list of surprising sources of universality, discussed in section 6.3 (page 107).

## 7.8 NOT EVERY QUESTION ABOUT PROGRAMS IS UNCOMPUTABLE

You might be tempted to think, at this point, that any problem that takes a program as input is uncomputable. However, this is not true. There are

**PROBLEM HALTSBEFORE100**

- **Input:** A Turing machine  $M$ .
- **Solution:** “yes” if there exists some input  $I$  for which  $M$  halts before completing 100 steps; and “no” otherwise.

**Figure 7.19:** The computational problem HALTSBEFORE100, which is decidable despite its apparent similarity to HALTSONSOME.

many questions about a Python program  $P$  that are obviously computable, such as

- How many lines are in  $P$ ?
- Does  $P$  contain a variable called `loopCounter`?
- Does  $P$  contain any calls to the `print` function?

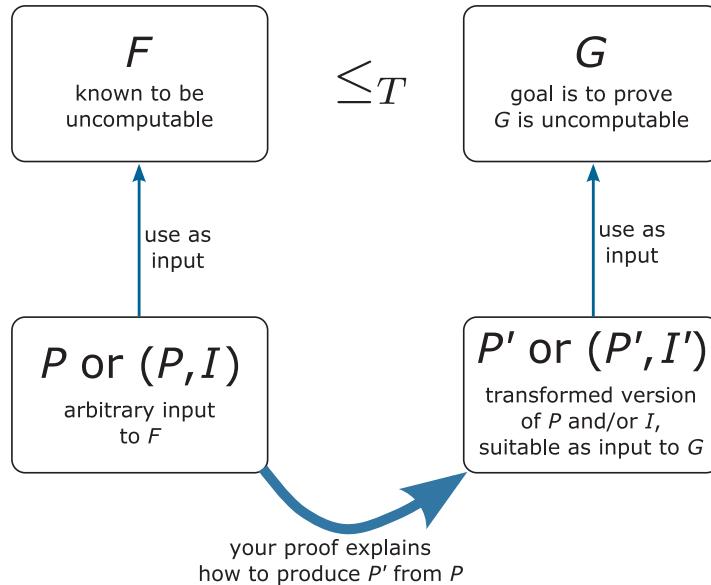
Given a Turing machine  $M$ , obviously computable questions include

- How many states does  $M$  have?
- Does the transition function include any transitions that change the tape contents? (Note: It’s undecidable to tell whether the tape contents will be changed, but we can easily examine the transition function to determine whether such a transition exists.)
- Does the state diagram contain any cycles?

In addition, there are some more subtle computable questions about programs. For example, HALTSBEFORE100 (defined in figure 7.19) asks whether there is some input for which a given Turing machine halts before its 100th step. This seems very similar to the undecidable problem HALTSONSOME, but in fact HALTSBEFORE100 is decidable. This is because the head of the Turing machine can move at most one cell per step, so it examines at most the first 100 cells of the tape in its first 100 steps. Therefore, although there are infinitely many possible inputs  $I$ , we need to consider only a finite number of possibilities for the first 100 steps. Specifically, if there are  $K$  symbols in the alphabet, there are  $K^{100}$  possibilities to be considered for the first 100 symbols on the tape. We can simulate the machine for 100 steps, for each of these  $K^{100}$  possibilities, to determine whether or not it halts before the 100th step. (Naturally, there’s nothing special about the number 100 in this example; HALTSBEFORE $N$  is decidable for any fixed  $N$ .)

## 7.9 PROOF TECHNIQUES FOR UNCOMPUTABILITY

The ability to construct Turing reductions and/or prove that problems are uncomputable is one of the key skills you should acquire from this book. Here we discuss three techniques for proving uncomputability; you need to understand all three. In each case, we assume your task is “prove that  $G$  is uncomputable,” for some computational problem  $G$ .



**Figure 7.20: The reduction recipe for proving Turing reductions.** The most crucial part of the proof is explaining how one could automatically transform  $P$  into  $P'$ , as shown by the bold arrow.

### Technique 1: The reduction recipe

Proving uncomputability by constructing Turing reductions can be daunting at first. Fortunately, there is a reasonably straightforward recipe for designing the reduction and writing out a concise mathematical proof. This reduction recipe is summarized in figure 7.20. The detailed steps in the recipe are as follows:

1. Choose a problem  $F$  that is already known to be uncomputable. We will plan to show  $F \leq_T G$ , thus proving  $G$  is uncomputable. Which  $F$  should you use? Often, YESONSTRING or YESONEMPTY are good choices. Another good strategy is to choose an  $F$  that is similar to  $G$ , since that might make the reduction simpler.
2. Consider an arbitrary input for  $F$ . Usually, the input will be a program  $P$ , possibly combined with an input string as  $(P, I)$ .
3. Describe how to construct  $P'$ , an altered version of  $P$ . Usually, you construct  $P'$  so that  $G(P')$  produces the same output as  $F(P)$ . In more complex proofs,  $G(P')$  is not identical to  $F(P)$ , but you can easily calculate  $F(P)$  from  $G(P')$ . Of course, there may be input strings involved so that we are instead dealing with  $G(P', I)$  and  $F(P, I)$ . Typically,  $P'$  follows this pattern:
  - (a) Compute  $v = P(I)$ , or something similar such as  $v = P(\epsilon)$  or  $v = P(I')$ , where  $I'$  is an altered version of  $I$ .
  - (b) Return a value that depends on  $v$  and somehow translates the properties of  $F$  into the properties of  $G$ .

4. Observe that, by construction,  $G(P')$  yields the solution  $F(P)$ , so the reduction is complete and we conclude that  $G$  is uncomputable.

Now let's look at an example of this recipe in action. Consider the problem `CONTAINSZONALL`, which takes as input a program  $P$  and asks the question, does  $P$  return a string containing the character z on all inputs?

**Claim 7.7.** `CONTAINSZONALL` is uncomputable.

*Proof of the claim.* We will show  $\text{YESONALL} \leq_T \text{CONTAINSZONALL}$ , which will prove that `CONTAINSZONALL` is uncomputable because `YESONALL` is already known to be uncomputable. Let  $P$  be an arbitrary input to `YESONALL`. It's easy to construct  $P'$ , an altered version of  $P$ , that first computes  $v = P(I)$  and then returns a string containing z if and only if  $v = \text{"yes"}$ . By construction,  $\text{CONTAINSZONALL}(P') = \text{"yes"}$  if and only if  $\text{YESONALL}(P) = \text{"yes"}$ . Thus,  $\text{YESONALL}(P)$  can be computed via  $\text{CONTAINSZONALL}(P')$ , and the reduction is complete.  $\square$

Why did we choose `YESONALL` for the reduction here? Because it seems similar to `CONTAINSZONALL`. In fact, there are also simple reductions from `YESONSTRING` and `YESONEMPTY`.

For our second example, consider the problem `SHORTESTOUTPUT`, which takes as input a program  $P$  and returns the shortest possible output string that  $P$  can produce (or "no" if the output is undefined for all inputs).

**Claim 7.8.** `SHORTESTOUTPUT` is uncomputable.

*Proof of the claim.* We will show  $\text{YESONEMPTY} \leq_T \text{SHORTESTOUTPUT}$ , which will prove that `SHORTESTOUTPUT` is uncomputable because `YESONEMPTY` is already known to be uncomputable. Let  $P$  be an arbitrary input to `YESONEMPTY`. It's easy to construct  $P'$ , an altered version of  $P$ , that works as follows:  $P'$  computes  $v = P(\epsilon)$ . If  $v = \text{"yes"}$ ,  $P'$  returns "x", and otherwise returns "xx". By construction,  $\text{SHORTESTOUTPUT}(P') = \text{"x"}$  if and only if  $\text{YESONEMPTY}(P) = \text{"yes"}$ . Thus,  $\text{YESONEMPTY}(P)$  can be computed via  $\text{SHORTESTOUTPUT}(P')$ , and the reduction is complete.  $\square$

Why did we choose `YESONEMPTY` for the reduction here? Because it produced the simplest proof after experimenting with two or three possibilities. After practicing a few reductions, you will find that reducing from `YESONEMPTY` often works well.

## Technique 2: Reduction with explicit Python programs

This technique has already been used frequently in this book, for example in figure 7.12. Recall that we are trying to prove  $G$  is uncomputable. As with the cookbook recipe in Technique 1, we first choose an uncomputable problem  $F$ , planning to reduce  $F$  to  $G$ . Then, we write a Python program `alterFToG.py`, explicitly implementing the program  $P'$  described above in the cookbook recipe (see figure 7.12 for a useful example, namely `alterYesToNumChars.py`). Next, we write a Python program `FViaG.py`, explicitly demonstrating that  $F$  can be computed if  $G$  can be computed (e.g., see `yesViaNumChars.py` in figure 7.12).

Finally, we observe that the two programs yield a reduction from  $F$  to  $G$ , proving that  $G$  is uncomputable.

Theoretical computer scientists don't usually write proofs using complete program listings, in Python or any other language. Instead they use concise abstract proofs as in Technique 1, sometimes augmented with pseudocode. But Technique 2 provides a deep understanding of the proof and often helps eliminate errors, so it's well worth mastering both approaches.

### Technique 3: Apply Rice's theorem

You can often prove that a problem is uncomputable by applying one of the variants of Rice's theorem, as described in claim 7.6 on page 134. For example, we can re-prove our claim about `CONTAINSZONALL` very easily this way:

**Claim 7.9.** `CONTAINSZONALL` is uncomputable.

*Proof of the claim.* Let  $S$  be the set of functions that map ASCII strings to ASCII strings and return strings containing a “z” on all inputs. Then `CONTAINSZONALL` is precisely the problem `COMPUTESONEOFS`. Note that  $S$  includes at least one computable function (e.g., the constant function returning “z”) and excludes at least one computable function (e.g., the constant function returning “y”). Hence the conditions of Rice's theorem are fulfilled and we conclude that `CONTAINSZONALL` is undecidable.  $\square$

Note that there is also a fourth technique for proving uncomputability: go back to first principles, and produce a self-contradictory program. This is how we proved our very first uncomputability results in chapter 3 (see claim 3.4 on page 38, for example). But usually, this technique is overkill and we don't discuss it further here.

Finally, we address two common points of confusion regarding proofs about Turing reductions:

**Common Turing reduction misunderstanding #1: execution of impossible programs.** Look back to figure 7.20, which explains the roles of programs  $P$  and  $P'$  in Technique 1 for proving Turing reductions. It's a common mistake to think that  $P$  and/or  $P'$  are executed during the reduction process, but in fact the opposite is true: *neither P nor P' is executed when a reduction is carried out*. Instead, we need only describe the construction of  $P'$  from  $P$ . Regardless of whether  $P$  and  $P'$  are “impossible” programs like `yesOnString.py`, the Turing reduction involves a proof that starts with  $P$  and constructs (or perhaps merely demonstrates the existence of) the corresponding  $P'$ .

In the language of Technique 2, this is equivalent to saying that `alterFToG.py` is never executed. It is the fact that we can show there *exists* a suitable `alterFToG.py`, without having to execute it, that permits the Turing reduction to be performed.

**Common Turing reduction misunderstanding #2: “This proof would work for any problem.”** At first glance, it might seem that Technique 1 can be adapted to prove that any computational problem

reduces to any other computational problem. Therefore, it's useful to think about exactly where the technique would break down in the case that no Turing reduction exists. For example, imagine using Technique 1 to attempt a proof that YESONEMPTY reduces to CONTAINSGAGA. We consider an arbitrary input  $P$  for YESONEMPTY. We need to transform  $P$  into a new string  $P'$  such that  $P'$  contains "GAGA" if and only if  $P(\epsilon) = \text{"yes"}$ . But now we are stuck. How do we construct this  $P'$ ? It seems we would first need to know whether or not  $P(\epsilon) = \text{"yes"}$ . But that is exactly the problem we were trying to solve in the first place, via a reduction.

## EXERCISES

**7.1** The program `isOddViaReduction.py` in figure 7.2 reduces ISODD to LASTDIGITISEVEN. Write a similar program that does the reverse, reducing LASTDIGITISEVEN to ISODD.

**7.2** Consider the decision problem INTONSTRING, which takes two strings  $P, I$  as input and outputs "yes" if  $P(I)$  is a nonnegative integer (i.e., a string containing only digits), and "no" otherwise. Using the "explicit Python program" approach (i.e., Technique 2, page 138), write two Python programs that together demonstrate a reduction from YESONSTRING to INTONSTRING.

**7.3** Suppose  $F_1, F_2, \dots, F_{10}$  are computational problems such that each problem reduces to the next one:  $F_1$  reduces to  $F_2$ ,  $F_2$  reduces to  $F_3$ , and so on. Suppose also that  $F_3$  is computable and  $F_5$  is uncomputable. Which of the other problems are therefore computable? Which are uncomputable? Which could be either computable or uncomputable?

**7.4** Suppose  $F$  and  $G$  are general computational problems, and  $D$  is a decision problem. Also suppose  $F$  reduces to  $G$  and  $D$  reduces to  $G$ . What, if anything, can we conclude if

- (a)  $D$  is undecidable?
- (b)  $G$  is computable?
- (c)  $F$  is computable?

**7.5** Prove that each of the following problems is undecidable. Each problem receives a single program  $P$  as input.

- (a) EMPTYONEMPTY: Does  $P(\epsilon) = \epsilon$ ?
- (b) GAGAONEMPTY: Does  $P(\epsilon) = \text{"GAGA"}$ ?
- (c) NOONSOME: Does  $P(I) = \text{"no"}$  for some  $I$ ?
- (d) YESONGAGA: Does  $P(\text{"GAGA"}) = \text{"yes"}$ ?
- (e) LONGERTHAN3ONALL: Is  $|P(I)| > 3$  for all  $I$ ?

**7.6** Consider the decision problem INTONALLINTS. This problem takes a single string  $P$  as input, and outputs "yes" if  $P(I)$  is a nonnegative integer for all input strings  $I$  which are themselves nonnegative integers. In other words, it answers the following question: Does  $P$  output a nonnegative integer for all nonnegative integer inputs? Prove that INTONALLINTS is undecidable. You may use any of

```

1 from yesOnString import yesOnString
2 def yesOnPosIntsViaYoS(progString):
3     i = 1
4     while True:
5         if not yesOnString(progString, str(i))=='yes':
6             return 'no'
7         i += 1

```

**Figure 7.21:** The Python program `yesOnPosIntsViaYoS.py`, which purports to be a reduction from YESONPOSINTS to YESONSTRING.

the three techniques suggested in section 7.9 (or even better, practice using all three techniques).

**7.7** The decision problem YESONPOSINTS receives a single program parameter  $P$ , and the solution is “yes” if and only if  $P(I)$  = “yes” for all strings  $I$  that represent positive integers. The program `yesOnPosIntsViaYoS.py`, shown in figure 7.21, is an attempt at reducing YESONPOSINTS to YESONSTRING. Explain why this is not a correct Turing reduction.

**7.8** In mathematics, a *fixed point* of a function  $f$  is a value  $x$  such that  $f(x) = x$ . For a program  $P$ , we can similarly define a *fixed string* as a string  $I$  such that  $P(I) = I$ . We can then define the decision problem HASFIXEDSTRING, which takes an input  $P$  and outputs “yes” if and only if  $P$  has a fixed string. Prove that HASFIXEDSTRING is undecidable.

**7.9** Define the computational problem NUMLONGER as follows. The input is a program  $P$ . The solution is the number of distinct strings  $I$  for which  $P(I)$  is defined and is longer than  $I$ . In other words, the solution is the cardinality of the set

$$\{I \in \text{ASCII}^* \text{ such that } |P(I)| > |I|\}.$$

Of course, this set could be infinite, in which case the solution is the string “infinite”. Is NUMLONGER computable? Give a proof of your answer.

**7.10** Our proof that all four variants of the halting problem are undecidable (page 128) omitted some details. Fill in one of these missing details as follows: using the “explicit Python program” approach (i.e., Technique 2, page 138), write Python programs demonstrating a reduction from HALTSSTRING to HALTSONSOME.

**7.11** Consider our proof that NUMCHARSONSTRING is uncomputable. In particular, examine the program `alterYesToNumChars.py` (top of figure 7.12). Line 10 of this program returns the string “xx”, but the proof could remain valid even if we had used a different return value here. Which return values can be used at line 10? In particular, for which of the following return values would the proof remain valid:  $\epsilon$ , “a”, “aa”, “aaa”, “aaaa”? Explain your answer.

**7.12** Define the decision problem SLOWERTHANINPUTLENGTH as follows. The input is a program  $P$ . The solution is “yes” if and only if the number of steps before terminating on input  $I$  is more than  $|I|$ , for all  $I$ . In other words, SLOWERTHANINPUTLENGTH answers the question, is the running time of  $P$  always more than the length of its input? (Note: We define the running time as

infinite whenever  $P(I)$  is undefined.) Prove that SLOWERTHANINPUTLENGTH is undecidable.

**7.13** Let  $P = \text{"def P(x): return str(5*len(x))"}.$

- (a) Give an example of a computational problem  $F$  such that  $P$  is a positive instance of COMPUTES $_F$ .
- (b) Give an example of a computational problem  $G$  such that  $P$  is a negative instance of COMPUTES $_G$ .

**7.14** Use one of the variants of Rice's theorem (i.e., Technique 3, page 139) to prove that each of the following problems is uncomputable:

- (a) COMPUTESENGTH: Input is program  $P$ , solution is “yes” if and only if  $P(I) = |I|$  for all  $I$ .
- (b) SEARCHESFORSUBSTRING: Input is program  $P$ , solution is “yes” if and only if  $P$  searches for some fixed substring. That is, there exists a string  $s$  such that  $P(I) = \text{“yes”}$  if and only if  $s$  is a substring of  $I$ .

**7.15** Consider the computational problem ECHOESFIRSTCHARACTER, abbreviated to EFC, defined as follows. The input to EFC is an ASCII string  $P$ . If  $P$  is not a SISO Python program, the solution is “no”. Otherwise we may assume  $P$  is a SISO Python program and the solutions are defined as follows: If there exists some nonempty string  $I$  such that the first symbols of  $I$  and  $P(I)$  are identical, then the first symbol of  $I$  is a solution. For example, if  $P(\text{“banana”}) = \text{“breakfast”}$ , then “b” is a solution. (In other words,  $P$  “echoes” the first character of  $I$  in this case.) If there is no string  $I$  with this property, the instance is negative and the solution is “no”.

Prove that EFC is uncomputable.

**7.16** Consider the decision problem TMTENSTEPSONALL, defined as follows: The input is an ASCII description of a Turing machine  $M$ . The solution is “yes” if  $M$  executes at least ten steps on all inputs; otherwise the solution is “no”. Is TMTENSTEPSONALL decidable? Justify your answer.

**7.17** Consider the search problem TRIPLEONSOME defined as follows. The input is a SISO Python program  $P$ . If there exists a positive integer  $M$  for which  $P(M) = 3M$ , then  $M$  is a solution. If no such  $M$  exists, the solution is “no”. Prove that TRIPLEONSOME is uncomputable.

(Note: To keep our notation simple, the above definition ignored the distinction between integers and strings that represent integers. A more accurate description would state that  $\text{str}(M)$  is a solution if  $P(\text{str}(M)) = \text{str}(3M)$ .)

**7.18** State whether each of the following instances of the Post correspondence problem is positive or negative. Explain your answers.

(a)  $\left\{ \begin{array}{c} \boxed{C} \\ \boxed{A} \end{array}, \begin{array}{c} \boxed{G} \\ \boxed{T} \end{array}, \begin{array}{c} \boxed{A} \\ \boxed{C} \end{array}, \begin{array}{c} \boxed{T} \\ \boxed{G} \end{array} \right\}$

(b)  $\left\{ \begin{array}{c} \boxed{C} \\ \boxed{CC} \end{array}, \begin{array}{c} \boxed{A} \\ \boxed{AA} \end{array} \right\}$

(c)  $\left\{ \begin{array}{c} \boxed{G} \\ \boxed{CG} \end{array}, \begin{array}{c} \boxed{C} \\ \boxed{CG} \end{array}, \begin{array}{c} \boxed{CG} \\ \boxed{T} \end{array}, \begin{array}{c} \boxed{GAT} \\ \boxed{A} \end{array} \right\}$

# 8



## NONDETERMINISM: MAGIC OR REALITY?

A system which is part of a deterministic system I shall call “determined”; one which is not part of any such system I shall call “capricious.”

— Bertrand Russell, *On the Notion of Cause* (1912)

One of the interesting things about computers is that they seem to be able to do many things at once. For example, it’s possible to watch a video clip in one window on your monitor, while simultaneously editing a document in another window and also receiving pop-up notifications from e-mail and social network programs. This kind of multitasking is very easy for modern computers, because modern computers often have multiple CPUs, and each CPU typically has multiple *cores*. Each of the cores in a computer is capable of simultaneously running a different program. So modern computers can literally do many things at once.

How does this kind of multitasking fit with the types of computational models we have encountered in the book so far? By now, we are very familiar with the fact that—in terms of computability, but not necessarily efficiency—all real computers are equivalent to a standard Turing machine. Yet a standard Turing machine can’t multitask: it has only a single head, and it can read and write only a single symbol at a time. So, we have an apparent paradox: modern, multitasking computers are equivalent to Turing machines that can’t multitask.

The resolution of the paradox lies in the definition of “multitask.” It turns out that we can make a Turing machine *appear* to multitask simply by running it extremely fast, and making it rapidly switch between several different tasks. In fact, until approximately the year 2000, most desktop computers had only one CPU with a single core. So in a literal sense, these 20th-century PCs could not multitask—at any one instant, these single-core machines could work on only one task. But even in the 1990s, desktop computers gave the *appearance* of multitasking. The operating system would permit dozens of programs to be loaded simultaneously into memory, and would switch between programs every

few milliseconds. The switching was fast enough that, to a human observer, the computer appeared to be doing many things at the same time.

Two words that describe the multitasking ability of computers are *parallelism* and *nondeterminism*. These words emphasize different aspects of the same idea. “Parallelism” emphasizes the idea that tasks are done *in parallel*—that is, at the same time. “Nondeterminism” emphasizes the idea that at any one instant, the next step performed by a multitasking computer could be chosen in a nondeterministic way—that is, depending on the design of the operating system and on unpredictable events such as keypresses and the arrival of network packets, the next step could advance any one of the tasks currently being performed. This contrasts with the *deterministic* operation of a single task, in which the next step of the computation is always uniquely determined.

Note that some books draw the following important distinction between parallelism and nondeterminism: in a parallel computation, the simultaneous computations can communicate with each other; in a nondeterministic computation, each simultaneous computation operates completely independently. In this book, we ignore the possibility that simultaneous computations could communicate with each other, and focus only on nondeterministic computations that each act independently.

The title of this chapter asks us to consider whether nondeterminism is “magic or reality.” In one sense, nondeterminism is clearly real: in the 21st century, it is common for scientists and engineers to run a single computation distributed over a vast number of CPU cores located throughout one or more data centers. This allows the computation to terminate many orders of magnitude faster than if it were executed on a single core. On the other hand, we will soon see that the formal definition of nondeterminism allows an *unbounded* number of computations to run in parallel. This would be the equivalent, in real life, of assuming your computation can use as many CPU cores as desired—even if the number of cores exceeds the number of atoms in the universe. Clearly, this is not a physically realistic assumption. Hence, the type of nondeterminism employed by theoreticians is a little “magical.” Nevertheless, the theoretical study of nondeterminism is extremely fruitful, leading to important practical results about the computability of problems and the efficiency of algorithms.

## 8.1 NONDETERMINISTIC PYTHON PROGRAMS

In modern computer operating systems, the basic unit of parallelism is called a *thread*. Typically, computers execute dozens (and sometimes hundreds or thousands) of threads simultaneously. This simultaneous execution can be genuinely simultaneous (on separate cores), or it can be faked by rapidly switching between threads on the same core. As computer programmers, we don’t need to know whether the multitasking will be real or fake. We can just create new threads, start them running, and let the operating system take care of the details.

In Python, this is done by importing the `threading` module, then using `Thread()` to create a thread object, the `start()` method to run the thread, and the `join()` method to wait for the thread to finish. As an example, consider the problem of examining an input string to see whether it contains one of the

### PROBLEM CONTAINSNANA

- **Input:** A string.
- **Solution:** “yes” if the input contains one of the substrings “CACA”, “GAGA”, “TATA”, or “AAAA”; and “no” otherwise.

**Figure 8.1:** Description of the computational problem CONTAINSNANA.

```

1  def containsNANA(inString):
2      strings = ['CACA', 'GAGA', 'TATA', 'AAAA']
3      for string in strings:
4          if string in inString:
5              return 'yes'
6      return 'no'

```

**Figure 8.2:** The Python program `containsNANA.py`.

substrings “CACA”, “GAGA”, “TATA”, or “AAAA”. Molecular biologists sometimes use the symbol N to represent any one nucleotide (i.e., C, A, G, or T)—think of N as representing the “n” in “any.” So we can concisely describe this problem as a search for “NANA”. Figure 8.1 gives a formal definition of the problem, which we will call CONTAINSNANA. One simple method of solving CONTAINSNANA is to search the input four times, in sequence, looking for a different string each time. The program `containsNANA.py`, shown in figure 8.2, demonstrates this simple and effective approach.

Alternatively, we might try a slightly more complex approach to the same problem: we still search the input separately, four times, looking for a different string each time—but instead of waiting for each scan of the input to complete before starting the next scan, we perform all four scans at the same time. This requires us to use Python’s `threading` module to launch each of the scans in a separate thread. The program `ndContainsNANA.py`, shown in figure 8.3, does exactly this. Note that the “nd” in `ndContainsNANA.py` is an abbreviation of “nondeterministic.” We’ll often use the prefix “nd” to signify nondeterminism.

Let’s take a look at this multithreaded code, starting with the helper function `findString(string, text, ndSoln)` defined near the end (line 36). This is the function that will be launched in four separate threads, with four different values for the `string` parameter. But the `text` parameter, which represents the text to be searched for an occurrence of `string`, will be the same in each of the four threads. Naïvely, we might expect `findString` to return “yes” if it finds an occurrence of `string` in `text`, and “no” otherwise. However, because `findString` will be running in a separate thread, there is no simple way of returning a value to the main thread of the program. To help with this, the `utils` package provides a special class called `NonDetSolution`. `NonDetSolution` objects provide the facilities for storing and returning the solution of a nondeterministic computation. In particular, we see at line 36 that the third parameter of `findString` is `ndSoln`, an instance of the `NonDetSolution` class. So, instead

```

1   from threading import Thread
2
3   # function that searches the given input for four different strings, in parallel
4   def ndContainsNANA(inString):
5       # the list of strings to look for
6       strings = ['CACA', 'GAGA', 'TATA', 'AAAA']
7
8       # create an empty list that will store our threads
9       threads = []
10
11      # create an object that can store a nondeterministic solution computed by
12      # other threads
13      ndSoln = utils.NonDetSolution()
14
15      # create the threads that will perform the nondeterministic computation
16      for s in strings:
17          # create a thread that will execute findString(s, inString, ndSoln)
18          # when started; findString is a helper function defined below
19          t = Thread(target=findString,
20                      args = (s, inString, ndSoln))
21          # append the newly created thread to our list of threads
22          threads.append(t)
23
24      # Perform the nondeterministic computation. By definition, this means
25      # that each thread is started, and we get a return value if either (a)
26      # any thread reports a positive solution, or (b) all threads report
27      # negative solutions.
28      solution = utils.waitForOnePosOrAllNeg(threads, ndSoln)
29
30      return solution
31
32      # findString is a helper function that sets the nondeterministic solution to the
33      # value "yes" if the given string is found in the given text. This function is
34      # intended to be executed in a separate thread as part of a nondeterministic
35      # computation.
36      def findString(string, text, ndSoln):
37          if string in text:
38              ndSoln.setSolution('yes')

```

**Figure 8.3:** The Python program `ndContainsNANA.py`.

of returning a value, `findString` will report any occurrences of `string` using the `ndSoln` object. This is exactly what happens at line 38, where the solution buried inside the `ndSoln` object gets set to “yes”:

```
ndSoln.setSolution('yes')
```

Note that if `findString` does not find any occurrences of `string`, it does *not* set the solution to “no”. This is a pattern we will see repeatedly in this chapter: `ndSoln` objects are always initialized with a solution equal to “no”, and the solution is later altered only if a positive solution is found by one or more threads.

Now let's take a look at the main `ndContainsNANA()` function, at line 4. After defining the list of strings we want to search for (line 6), two important objects are created. The first (at line 9) is an empty list of threads, which will be used to store the threads created later. The second (at line 13) is the special `ndSoln` helper object from the `utils` package, which will be passed to all of the `findString` threads as described above.

The threads themselves are created in the loop at line 16. The syntax for creating threads in Python can seem obscure the first time you see it, so it's worth going over this in some detail. Here's a simple example:

```
myThread = Thread(target=myFunction, args=(val1, val2))
myThread.start()
```

The first line of this code creates a new thread whose name is `myThread`. This line uses a Python feature known as *keyword arguments*. For example, `target=myFunction` means that the parameter `target` receives the value `myFunction`. If desired, you can read about keyword arguments in an online tutorial, but we will rarely use them in this book. In the particular case above, the effect is as follows: when the thread begins executing, it will execute the function `myFunction`, using the arguments `(val1, val2)`. The second line of the code above instructs the thread to begin execution. In other words, it is exactly equivalent to

```
myFunction(val1, val2)
```

except that `myFunction` will run in a separate thread.

With this background, we can now understand line 19 of `ndContainsNANA.py`:

```
t = Thread(target=findString, args = (s, inString, ndSoln))
```

This creates a thread that, once it is started, will perform the same job as

```
findString(s, inString, ndSoln)
```

So, the loop at line 16 creates four separate threads that will search the same text for four different strings. However, none of the threads is started yet. They are merely created, then appended to a list of threads to be started later (line 22).

The threads are actually run at line 28, using the `waitForOnePosOrAllNeg()` function from the `utils` package. If you're interested, you can look at the implementation of `waitForOnePosOrAllNeg()`, but it's not necessary to do so. You can just trust that it does the following:

1. Start each thread in the `threads` list.
2. Wait until either
  - (a) *any* of the threads find a positive solution; or
  - (b) *all* of the threads terminate (implicitly, without changing the default negative value of the solution).
3. Return the solution.

Thus, when `ndContainsNANA.py` calls `waitForOnePosOrAllNeg()` at line 28, the effect is that “yes” is returned as soon as any one of the threads

### PROBLEM FINDNANA

- **Input:** A string.
- **Solution:** If the input contains the substring “CACA”, then “CACA” is a solution. If the input contains “GAGA”, then “GAGA” is a solution. And similarly for the substrings “TATA”, and “AAAA”. If the input contains none of the four desired substrings, the solution is “no”.

**Figure 8.4: Description of the computational problem FINDNANA.** Note that this problem is very similar to the decision problem CONTAINSNANA (figure 8.1, page 145). The only difference is that FINDNANA outputs a desired substring if it’s found, rather than outputting “yes”.

finds the string it is looking for. If all threads terminate without finding one of the desired strings, “no” is returned.

Note that multithreaded programming is notoriously challenging. In this book, we make no attempt to address the important skills required for general-purpose multithreaded programming. Instead, we can gain insight into nondeterministic computation by using the provided utilities such as `NonDetSolution` and `waitForOnePosOrAllNeg()`. Because there is no easy way to kill specific threads in Python, another useful tool is the `utils.killAllThreadsAndExit()` function. If your multithreaded Python programs are not terminating in the way you expect, call this function to guarantee termination. See the `utils` package for some examples of `killAllThreadsAndExit()` in action.

This would be a very good time to get some practical experience with writing nondeterministic Python programs: try to complete exercise 8.1 before reading on.

## 8.2 NONDETERMINISTIC PROGRAMS FOR NONDECISION PROBLEMS

Recall from chapter 4 that computational problems can be decision problems, or general problems. Decision problems have “yes” and “no” as their only possible solutions, and these outputs are also known as *positive* and *negative* solutions respectively. General problems have arbitrary solution sets; the solution “no” is still regarded as a negative solution, and any other solution is a positive solution. Examples of positive solutions include “534”, “GAGA”, and “yes”.

Our first example of a nondeterministic program (`ndContainsNANA.py`) was used to solve a decision problem (CONTAINSNANA). Let’s now investigate how a nondeterministic program can solve a general, nondecision problem. We will use a simple variant of CONTAINSNANA, called FINDNANA. FINDNANA is described formally in figure 8.4. Just as with CONTAINSNANA, the basic idea is to search an input string for one of the four desired substrings (“CACA”, “GAGA”, “TATA”, or “AAAA”). But this time, if one of the substrings is found, that substring is returned as a solution, instead of “yes”. For example, given input “CCCTATACCCGAGACCC”, the solution set consists of the two elements “TATA” and “GAGA”.

```

# The main body of this program is essentially identical to containsNANA.py,
2 # but this helper function returns different information.
def findString(string, text, nonDetSolution):
4     if string in text:
         nonDetSolution.setSolution(string) # note this difference

```

**Figure 8.5:** The Python program `ndFindNANA.py`. This program is essentially identical to `ndContainsNANA.py` (figure 8.3, page 146). The only difference is at line 5, which sets the solution to `string` rather than “yes”.

Note that there is nothing strange or new about this situation: in chapter 4, we saw plenty of computational problems that have solution sets with more than one element. By definition, a computer program solves a problem if it outputs any element of the solution set. The new feature in this chapter is that our programs might choose, nondeterministically, between the elements of the solution set. Thus, a program might produce different (but still correct) outputs when run multiple times on the same input. As a specific example of this, a deterministic `FindNANA.py` program might be written so that it always outputs “GAGA” on input “CCCTATACCGAGACCC”. In contrast, the nondeterministic `ndFindNANA.py` discussed next behaves differently: in principle, on input “CCCTATACCGAGACCC”, this program can output either “TATA” or “GAGA”, depending on how the operating system chooses to interleave the threads of the program.

The nondeterministic `ndFindNANA.py` is shown in figure 8.5. In fact, only a tiny change to `ndContainsNANA.py` is needed to produce the new program: at line 5 of figure 8.5, we use `setSolution(string)` instead of `setSolution('yes')`. The rest of the program is essentially identical to `ndContainsNANA.py`.

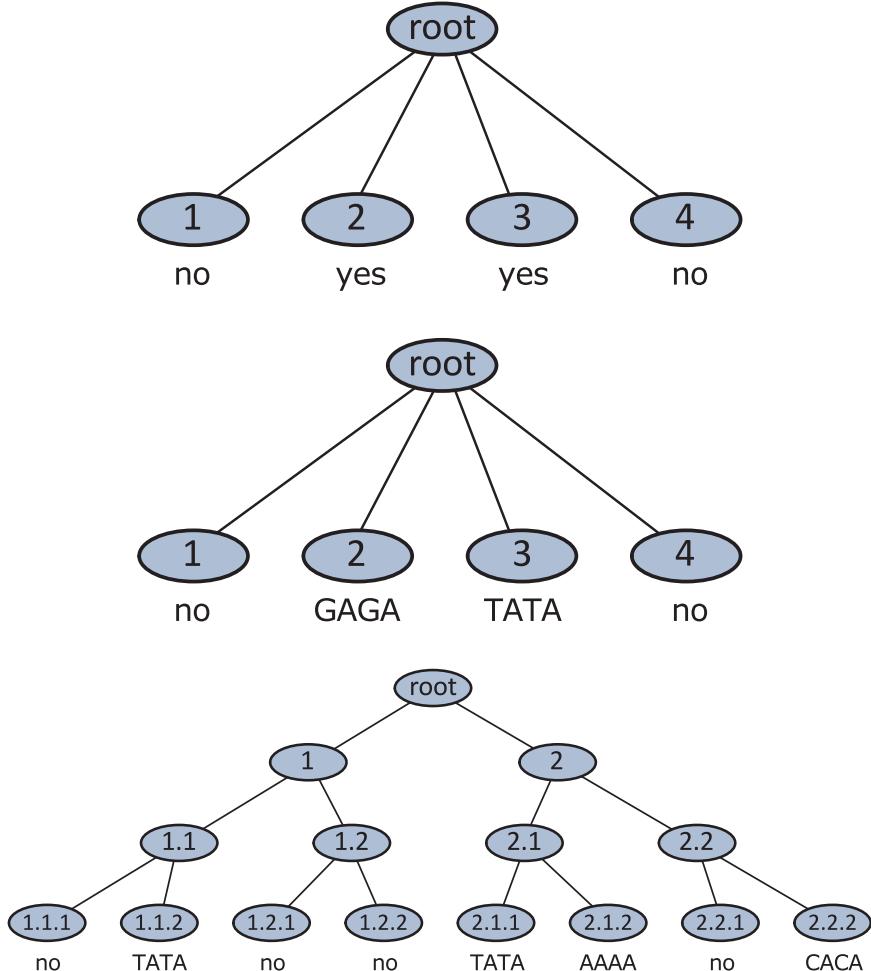
For a good understanding of nondeterminism with nondecision problems, it is strongly recommended to complete exercise 8.2 before continuing.

## 8.3 COMPUTATION TREES

A *computation tree* is a type of labeled graph that helps us visualize the effect of a nondeterministic program. Figure 8.6 shows three examples of computation trees. Each thread in a computation is represented by a node of the tree. The initial thread of the program is called the “root” thread, and it is also the root node of the computation tree.

Threads spawned by the root thread are numbered sequentially as 1, 2, 3, …, and are shown as child nodes of the root. Child threads can spawn their own children, and so on. The names of these children are derived by appending “.1”, “.2”, “.3”, … to the name of the parent. For example, node 3.2 represents the second child of the third child of the root. The bottom tree in figure 8.6 shows an example with three levels of children.

Leaf nodes typically return some string value, which is a potential solution to the computational problem. This return value is noted on the tree (e.g., “no”,



**Figure 8.6: Computation trees.** Top: `ndContainsNANA.py` on input “CCCTATACCGAGACCC”. The output is “yes”. Middle: `ndFindGAGA.py` on input “CCCTATACCGAGACCC”. The output is “TATA” or “GAGA”. Bottom: A divide-and-conquer program solves FINDNANA, given a long string input that happens to contain two “TATA” substrings, one “AAAA” substring, and one “CACA” substring. The output is “TATA”, “AAAA”, or “CACA”.

“yes”, “GAGA”). There are exactly three types of leaves:

- **negative leaves:** leaves whose thread terminates and returns “no”
- **positive leaves:** leaves whose thread terminates and returns anything other than “no”
- **nonterminating leaves:** leaves whose thread does not terminate

In the simple examples of figure 8.6, all threads eventually terminate, so there are no nonterminating leaves. We will encounter more complex examples later.

It is crucial to understand how outputs of computation trees (and hence, of nondeterministic programs) are defined. We use the same “one-positive-or-all-negative” approach discussed earlier on page 147. This is easiest to explain when the tree is finite and all threads terminate, so let’s make that simplifying assumption for now:

**Definition of the output of a terminating nondeterministic computation.**

Let  $T$  be a computation tree representing a nondeterministic computation. For this definition, we assume  $T$  is a finite tree and all threads terminate. Therefore  $T$  has a finite number of leaves and every leaf is either positive or negative. If all leaves of  $T$  are negative, the output of the computation is “no”. Otherwise any one of the positive return values is a possible output.

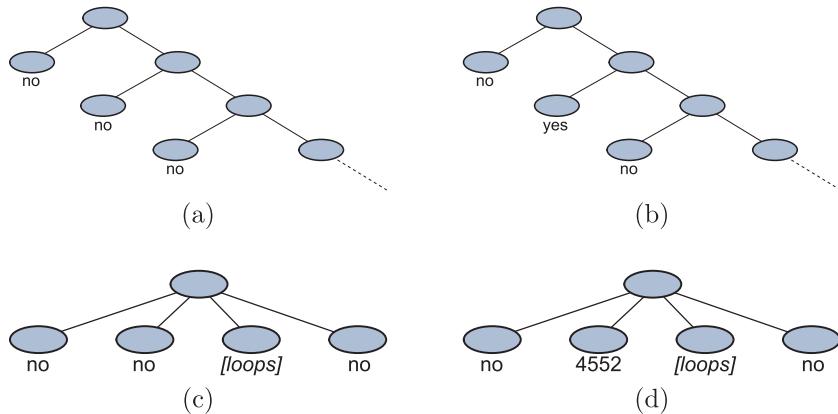
The examples of figure 8.6 illustrate this definition. The top tree shows the effect of `ndContainsNANA.py` on input “CCCTATACCCGAGACCC”. If all four leaf threads had returned “no”, the output would be “no”. But in fact, two of the leaves returned “yes”, so the output is “yes”.

The middle tree shows the effect of `ndFindNANA.py` on input “CCCTATA-CCCGAGACCC”. There are two positive leaves, with return values “GAGA” and “TATA”. Hence, any run of the program on this input will produce the output “GAGA” or “TATA”. It is impossible to determine which of these two possibilities will be chosen: we say the program makes a *nondeterministic choice* between “GAGA” and “TATA”.

The bottom tree shows the effect of another program that solves FINDNANA. The source code of this program isn’t listed here, because the details would be too distracting. (If you’re interested, however, check out `ndFindNANADivConq.py` in the book resources.) The program uses a divide-and-conquer strategy to search for a NANA substring. Specifically, it splits the original input into two halves, and launches two new threads to search for NANA substrings in each half. This process is applied recursively—two additional times, in the particular example shown here. Each resulting leaf thread searches deterministically for a NANA substring in the small part of the initial input it has been assigned. In this particular example, the output is a nondeterministic choice between “TATA”, “AAAA”, and “CACA”.

It’s well worth experimenting with the provided `ndFindNANADivConq.py`. In particular, execute the test function `testNdFindNANADivConq()` a few times. Depending on which Python implementation and hardware you are using, you may see different outputs on different runs of this program. So here you can see that the program is quite literally nondeterministic: it produces different outputs on different runs, depending on which thread happens to find a positive solution first.

The analysis of computation trees gets more complicated when we consider computations that don’t terminate. Figure 8.7 shows some of the possibilities: the computation tree can become infinitely deep, as in (a) and (b); or some of the leaf threads can enter infinite loops, as in (c) and (d). Even in these cases, however, any positive return value represents a successful computation. If there are any positive leaves, then the program output is selected nondeterministically from the positive return values—this is what happens in trees (b) and (d) of figure 8.7.



**Figure 8.7: Nonterminating computation trees.** The outputs are (a) undefined; (b) “yes”; (c) undefined; (d) “4552”.

To return a negative output, however, we require that the tree is finite and all threads terminate with a negative output. Otherwise the output is undefined—as in trees (a) and (c).

This definition might initially seem strange, but it makes sense if we consider an observer waiting for an output from the program. Suppose that at some instant, a large number of threads have terminated and returned “no”, but some threads are still running. It is of course possible that one of the running threads will eventually terminate with a positive output, so we cannot yet conclude that the output will be “no”. This leads to the following formal definition:

**Definition of the output of a nondeterministic computation.** Let  $T$  be a computation tree representing a nondeterministic computation. If  $T$  has any positive leaves, any one of the positive return values is a possible output. If  $T$  is finite and all leaves are negative, the output of the computation is “no”. Otherwise the output is undefined.

### *How fast can a computation tree grow?*

This subsection covers some technicalities that will be needed in later chapters but can be skipped at first reading. Therefore, feel free to skip ahead to section 8.4.

Python’s `Thread.start()` method can be used to launch only one new thread at a time. Strictly speaking, therefore, the computation trees of Python programs are always binary trees: each internal node has two children, one executing the new thread and the other continuing the existing thread. Sometimes, however, we model the computation more loosely and consider several threads to be launched simultaneously, as in the top two trees of figure 8.6. Similarly, when we examine nondeterministic Turing machines in section 8.5, we will discover that these machines can launch multiple clones in a single step. Nevertheless, for any given nondeterministic Python program or Turing machine, there is some

constant  $C$  bounding the number of threads that can sprout in a single step. And it turns out that by adopting a more careful model, we can always take  $C = 2$  if desired.

To summarize, computation trees can grow exponentially fast, but no more than that. We may always assume that there are at most  $C^k$  nodes at level  $k$ , and we can take  $C = 2$  if desired.

## 8.4 NONDETERMINISM DOESN'T CHANGE WHAT IS COMPUTABLE

In chapter 5, we defined a hierarchy of computational models, starting with the standard Turing machine and progressing through multi-tape machines, random-access machines, and finally a fully fledged, single-core modern computer. We saw that—provided we ignore issues of efficiency, and consider computability only—each of these models is exactly equal in terms of computational power. But none of these models had the ability to do multiple things at once. So it's natural to ask, if we give our computers the ability to do genuine multitasking on multiple cores (and not just the fake “rapid-switching” multitasking described earlier), will they be able to solve some additional types of problems? The answer is no. It turns out that multi-core computers can solve exactly the same set of problems as single-core computers. This is formalized in the following claim:

**Claim 8.1.** Any problem that can be computed by a nondeterministic Python program can also be computed by a deterministic Python program.

*Sketch proof of the claim.* The basic idea of this proof is that you can always simulate nondeterminism in a deterministic way. The simulation works as follows. Every time a new thread is created, add it to a list of tasks the program is working on, and make sure to switch between all tasks in the list periodically. If the computation tree contains any positive leaves, the deterministic simulation is guaranteed to eventually reach one of them and return that value. If the computation tree is finite and contains only negative leaves, the deterministic simulation is guaranteed to eventually simulate the entire tree and correctly return “no”. If the computation tree has no positive leaves, and is infinite or contains nonterminating leaves, the deterministic simulation won't terminate either—but that's fine, since the output of the nondeterministic computation isn't defined. Hence, the deterministic simulation always produces a correct output, and the claim is proved.  $\square$

This proof has an interesting connection to real-world computers. Recall that, as discussed on page 143, desktop computers in the 1990s typically had only a single core. But their operating systems could still run multithreaded code by switching between threads periodically. In fact, these computers were using a practical implementation of the above proof.

It's extremely important to note that the “equivalence” of deterministic and nondeterministic programs in the above claim relates only to computability and not to efficiency. The deterministic and nondeterministic versions of a program could be vastly different in terms of efficiency.

### PROBLEM GTHENONET

- **Input:** A genetic string bracketed by x's. (Example: xCTCGTGATTGx.)
- **Solution:** "yes" if the input contains a "G" character with exactly one "T" between the "G" and either end of the string. Otherwise the solution is "no". (Solution for above example: "yes", because the first G is preceded by exactly one T.)

Further examples: "xATGTTx", "xATTGTGCCTACx", and "xTGGGTTx" are all accepted; "xATTGx", "xCTTGAGTGTATx", and "xCTCTx" are all rejected.

**Figure 8.8:** Description of the computational problem GTHENONET.

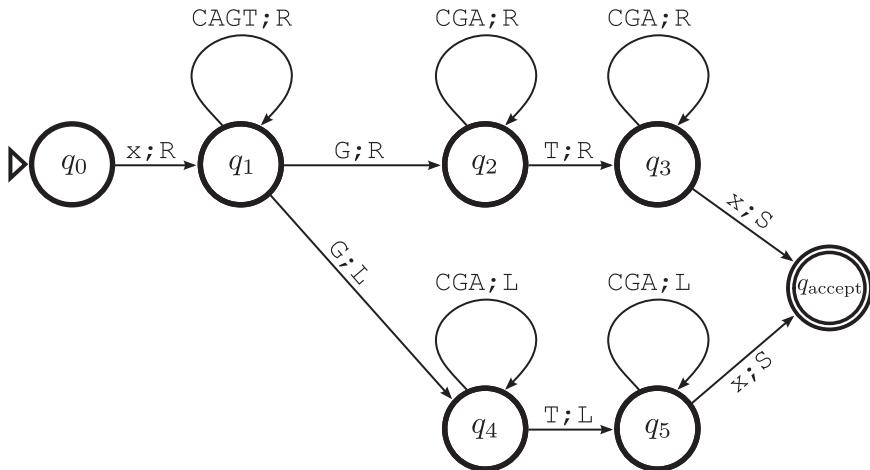
## 8.5 NONDETERMINISTIC TURING MACHINES

Nondeterministic Python programs are very helpful for understanding nondeterministic computations in practice. But they are often too unwieldy for rigorous mathematical proofs about computation. The usual way to formalize the notion of nondeterministic computing is to define a *nondeterministic Turing machine*. This is a standard Turing machine that can make clones of itself whenever necessary. Here, a "clone" of a machine is an exact copy of that machine, in the same state and with the same tape contents—except that, as we shall soon see, each newly created clone follows a different transition as its first step. Cloned machines continue operating in parallel with each other. And clones can create new clones of themselves, and so on.

Before seeing our first example of a nondeterministic Turing machine, we need to understand the decision problem GTHENONET, which is described in figure 8.8. The input is a genetic string (surrounded for convenience by x's), and the input is accepted if it contains some "G" character with exactly one "T" between the "G" and either end of the string. See figure 8.8 for some detailed examples.

It's not too hard to solve GTHENONET with a standard, deterministic Turing machine. But we will instead present a simple, elegant approach that employs nondeterminism. The basic idea will be to scan the input, searching for a G. If a G is found, the machine will transform itself into three clones labeled 1, 2, 3 as follows:

- Clone 1 will move to the left. It will determine whether there is exactly one T between the currently scanned G and the start of the input.
- Clone 2 will move to the right. It will determine whether there is exactly one T between the currently scanned G and the end of the input.
- Clone 3 will continue moving to the right, searching for another occurrence of a G. If and when it finds a G, three new clones are created. We might label these new clones 3.1, 3.2, 3.3. The behavior of clones 3.1, 3.2, and 3.3 is the same as clones 1, 2, and 3 respectively, except that they start at the currently scanned G.

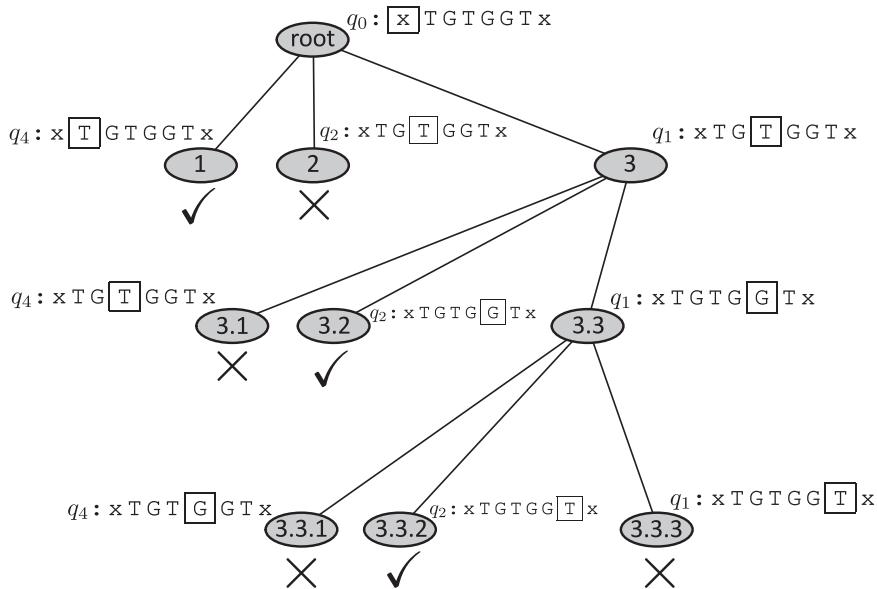


**Figure 8.9: The nondeterministic Turing machine  $G\text{thenOneT}$ .** State  $q_1$  is the only state that employs nondeterminism. When the machine is in state  $q_1$ , and the scanned symbol is a  $G$ , three clones are created—in states  $q_1$ ,  $q_2$ , and  $q_4$  respectively.

Obviously, the description above covers only the start of the process, which continues recursively: if clone 3.3 encounters a  $G$ , it spawns clones 3.3.1, 3.3.2, 3.3.3—and so on.

Figure 8.9 shows the nondeterministic Turing machine  $G\text{thenOneT}$ , which implements the nondeterministic computation just described. At first glance, this nondeterministic machine looks essentially the same as a deterministic one. The key difference is that states can have ambiguous transitions. That is, there can be states in which scanning some particular symbol results in two or more valid transitions. Figure 8.9 has exactly one of these nondeterministic states:  $q_1$  has three valid transitions if a  $G$  is scanned, leading to  $q_2$ ,  $q_4$ , or back to  $q_1$ . Another way of saying this is that three new clones are created whenever the machine enters  $q_1$  and scans a  $G$ .

Figure 8.10 shows the machine in action, processing the input “ $xTGTGGTx$ ”. The diagram in this figure is a computation tree, very similar to our earlier computation trees for Python programs. Each node in the tree represents a Turing machine clone. At the root of the tree is the initial Turing machine, which we also refer to as the root machine. This machine begins in state  $q_0$ , scanning the leftmost symbol. This configuration is summarized on the figure as  $q_0 : \boxed{x} TGTGGTx$ , using the same notation first introduced in chapter 5 (see page 76). After two steps, which are not explicitly shown on the computation tree, the machine finds itself in state  $q_1$ , scanning a  $G$ . This is precisely the ambiguous configuration discussed earlier, which therefore launches three new clones, labeled 1, 2, and 3. The initial state of each clone is shown on the computation tree. The nondeterministic computation continues in this way, eventually launching a total of 10 clones (including the root). Similarly to our earlier examples, the computation tree possesses leaf clones and internal clones. In this example, the seven leaves are clones 1, 2, 3.1, 3.2, 3.3.1, 3.3.2, and 3.3.3. By definition, they spawn no additional clones. The internal clones are the root,



**Figure 8.10: A computation tree for `GthenOneT` processing the input “`xTGTGGTx`”.**  
 Each shaded oval represents a Turing machine clone, with edges indicating the spawning of new clones. (For example, the root machine spawns clones 1, 2, and 3.) The initial configuration of each clone is shown next to it. Clones that accept their input are marked with ✓; clones that reject are marked with ✗.

3, and 3.3. By definition, the internal clones spawn new clones and immediately disappear. So the result of the nondeterministic computation is determined only by the leaves.

In this simple example, every leaf either accepts or rejects, which are denoted in the figure by ✓ and ✗ respectively. But just as with nondeterministic Python programs, nondeterministic Turing machines can produce infinite trees or trees with nonterminating leaves. The definition of the possible outputs is extremely similar to the definition for Python programs (on page 152): if the tree is finite and all leaves terminate and reject, the result is *reject*; if any leaf accepts, the result is *accept*; otherwise the result is *undefined*. In the particular example of figure 8.10, three leaves accept, so the overall result is *accept*.

## 8.6 FORMAL DEFINITION OF NONDETERMINISTIC TURING MACHINES

Recall that in a deterministic Turing machine, the transition function maps the current state  $q$  and scanned symbol  $x$  to a 3-tuple  $(q', x', d')$ , specifying the new state  $q'$ , the new symbol  $x'$  to be written on the tape, and the direction  $d'$  to move the head (see page 74 for a refresher on this). In symbols, the transition function of a deterministic machine maps

$$(q, x) \mapsto (q', x', d').$$

In a nondeterministic Turing machine, the transition function maps  $(q, x)$  to a *finite set* of possible 3-tuples, rather than a *single* 3-tuple. In symbols, the nondeterministic transition function maps

$$(q, x) \mapsto \{(q'_1, x'_1, d'_1), (q'_2, x'_2, d'_2), \dots\}.$$

In practice, most of these sets are singletons (which means they have only one element). In `GthenOneT`, for example, the transition function maps  $(q_2, T)$  to a singleton:

$$(q_2, T) \mapsto \{(q_3, T, R)\}.$$

In cases like this, the nondeterministic transition function behaves exactly like a deterministic transition function, moving the machine into the only possible new configuration. But for some values of  $(q, x)$ , the transition function maps to multiple possibilities. In `GthenOneT`, as we've already seen, the only nondeterministic configuration is  $(q_1, G)$ . Written in symbols, the transition function maps  $(q_1, G)$  to a set of three 3-tuples:

$$(q_1, G) \mapsto \{(q_1, G, R), (q_2, G, R), (q_4, G, L)\}.$$

These observations lead us to the formal definition of a nondeterministic Turing machine:

**Definition of a nondeterministic Turing machine.** A *nondeterministic Turing machine* is defined in the same way as a Turing machine (page 74), except that the transition function maps to *sets* of 3-tuples, rather than single 3-tuples.

In chapter 5, we saw an argument that deterministic Turing machines and deterministic Python programs are equivalent. We won't revisit those arguments in detail here, but it's easy to generalize the discussion to include nondeterminism. So we state the following claim without further proof.

**Claim 8.2.** Given a nondeterministic Python program, there exists an equivalent nondeterministic Turing machine, and vice versa.

Applying this type of equivalence to the fact that deterministic and nondeterministic Python programs are also equivalent (see claim 8.1 on page 153), we immediately conclude that deterministic and nondeterministic Turing machines have the same computational power. We state this formally:

**Claim 8.3.** Given a nondeterministic Turing machine  $M$  that solves a computational problem  $P$ , there exists a deterministic Turing machine  $M'$  that also solves  $P$ .

As with Python programs, it's worth noting that this claim refers only to computability and not to efficiency. The machines  $M$  and  $M'$  solve the same problem  $P$ , but their running times could be vastly different.

### *Nondeterministic transducers*

Recall from chapter 5 that Turing machines can be accepters, transducers, or both (see page 77). Accepters give an accept/reject decision, whereas transducers produce an output string. Our discussion so far has focused on nondeterministic Turing machines that are also accepters. It's not hard to extend the formalism to transducers, but we won't pursue that here. The details are very similar to nondeterministic Python programs, which can always be regarded as transducers.

## 8.7 MODELS OF NONDETERMINISM

Throughout this chapter, we have adopted the view that nondeterminism means executing multiple threads simultaneously. It's important to realize that there are other models of nondeterminism that appear to be quite different superficially, but in fact lead to exactly the same theoretical results. Here we summarize the three main models of nondeterminism:

- **Simultaneous threads.** This is the standard model used in this book. We imagine that a nondeterministic program executes all of its threads simultaneously.
- **Random choice.** In this model, a nondeterministic program executes only one thread. Every time it reaches a nondeterministic transition, the program chooses one of the transitions at random.
- **External choice.** In this model, a nondeterministic program again executes only one thread. But the method of choosing transitions differs from the “random choice” model. We imagine that the program has an external operator. This operator could be a human, or it could be some other entity such as the operating system of a modern computer. Every time the program reaches a nondeterministic transition, it asks the operator to choose one of the transitions.

In all three models, the computation tree of a nondeterministic computation looks the same. The definition of a program's output is based on the computation tree, so the definition remains the same in all three models. The only difference is how we imagine the computation tree being translated into physical reality.

## 8.8 UNRECOGNIZABLE PROBLEMS

Back in section 6.5, we used our new knowledge of universal computation to extend our understanding of recognizable and undecidable problems. Now we can use our new knowledge of nondeterminism to further extend our understanding. In section 6.5, we saw examples of decision problems (or, equivalently, languages) that are recognizable but undecidable. But are there any *unrecognizable* languages? In fact, we can easily find examples of such problems, by taking the complement of recognizable, undecidable problems:

**Claim 8.4.** The complement of a recognizable, undecidable decision problem is unrecognizable.

*Proof of the claim.* Let  $D$  be a recognizable, undecidable decision problem. Assume  $\overline{D}$  is recognizable and argue for a contradiction. Because  $D$  is recognizable, we have some program  $P_1$  that decides positive instances of  $D$ . Similarly, because  $\overline{D}$  is recognizable, we have some program  $P_2$  that decides positive instances of  $\overline{D}$ —or equivalently,  $P_2$  decides negative instances of  $D$ . Create a new nondeterministic program  $P$  that executes both  $P_1$  and  $P_2$  nondeterministically (i.e., in separate threads), returning the first result produced by either  $P_1$  or  $P_2$ . Note that  $P$  decides  $D$ : specifically,  $P$  terminates on all inputs because  $P_1$  terminates on positive  $D$ -instances and  $P_2$  terminates on negative  $D$ -instances. By claim 8.1 (page 153), there is an equivalent deterministic program  $P'$  that also decides  $D$ . But this contradicts the fact that  $D$  is undecidable, and the claim is proved.  $\square$

For example, let **NOTYESONSTRING** be the complement of **YESONSTRING** (i.e., on input  $P, I$ , the solution is “yes” if and only if  $P(I)$  is *not* “yes”). Then **NOTYESONSTRING** is unrecognizable.

## 8.9 WHY STUDY NONDETERMINISM?

Why do we study nondeterminism as a theoretical concept? It might appear foolish to study a physically unrealistic model of computation. And our nondeterministic model is physically unrealistic, because it allows an arbitrarily large number of threads to be created. Despite this, there are in fact several good reasons for studying nondeterminism:

- It is sometimes more natural to analyze a problem from a nondeterministic point of view. And it is sometimes easier to prove results using nondeterministic machines or programs. Claim 8.4 above is one example of this, and we will see more examples in the next chapter when we study nondeterministic finite automata.
- To a certain extent, nondeterminism is a reasonable model of how modern computer systems work. For example, when you submit a query to a web search engine, your query may be simultaneously processed by hundreds of computers—this is a vivid and realistic example of nondeterministic computation in action.
- The theory of nondeterministic computation has led to numerous results in complexity theory that have practical implications—especially the theory of NP-completeness which we will study in chapter 14. Consider the following example, which will be discussed in more detail on page 215. It’s obvious that large integers can be factorized efficiently using a nondeterministic Turing machine, since we can quickly launch enough clones to test every possible factor simultaneously. But no one knows how to perform factorization efficiently on a deterministic machine. Many modern cryptography systems depend on the (unproven) assumption that factorization cannot be done efficiently in practice. Therefore, theoretical results that shed light on the distinction between deterministic and nondeterministic computation can have implications for practical applications such as whether or not our cryptography is secure.

## EXERCISES

**8.1** Consider the decision problem HASMULTIPLEOFK defined as follows:

### PROBLEM HASMULTIPLEOFK

- **Input:** A single ASCII string containing a positive integer  $K$ , followed by a semicolon, followed by a list of positive integers  $m_1, m_2, \dots$ . Each  $m_i$  and  $K$  is represented in decimal notation, and the  $m_i$  are separated by whitespace.  
Example: “823 ; 18910 5235 3422”.
- **Solution:** “yes” if any of the  $m_i$  is a multiple of  $K$ ; and “no” otherwise.  
Example: For the input given above, the solution is “no”, because none of 18910, 5235, 3422 is divisible by 823.

Write a Python program that solves HASMULTIPLEOFK nondeterministically, using a separate thread for each integer  $m_i$  in the input.

**8.2** The problem FINDMULTIPLEOFK, defined below, is very similar to HASMULTIPLEOFK. However, it is a general computational problem rather than a decision problem.

### PROBLEM FINDMULTIPLEOFK

- **Input:** A single ASCII string containing a positive integer  $K$ , followed by a semicolon, followed by a list of positive integers  $m_1, m_2, \dots$ . Each  $m_i$  and  $K$  is represented in decimal notation, and the  $m_i$  are separated by whitespace.  
Example: “823 ; 18910 5235 3422”.
- **Solution:** Any value  $m_i$  that is a multiple of  $K$  is a solution. If none of the  $m_i$  is a multiple of  $K$ , the solution is “no”. Examples: for the input given above, the solution is “no”, because none of 18910, 5235, 3422 is divisible by 823. For the input “10 ; 720 342 90”, the solution set is {“720”, “90”}.

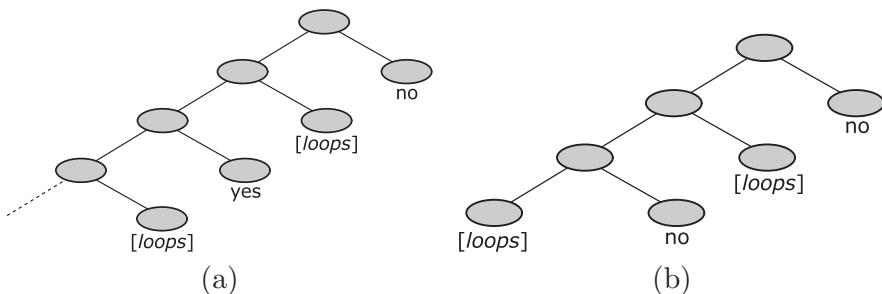
Write a Python program that solves FINDMULTIPLEOFK nondeterministically, using a separate thread for each integer  $m_i$  in the input.

**8.3** This exercise asks you to write a nondeterministic program for factorizing an integer  $M$ , and perform some experiments on it. The computational problem FACTOR is defined formally in figure 10.13, page 215. A simple deterministic program solving FACTOR is given in figure 10.14, page 216. This program works by testing all integers in the range  $[2, M-1]$  to see if they divide  $M$ . Note that we can significantly speed up the program by setting  $M' = \text{int}(\sqrt{M})$ , and testing the much smaller range  $[2, M']$ .

- (a) Write a nondeterministic version of this improved program by partitioning the range  $[2, M']$  into  $K$  smaller ranges. The program should launch  $K$  threads and test a different range in each thread.

- (b) Time your program for some moderate values of  $K$ , such as  $K = 2, 5, 10$ . When running on a multicore computer, do you observe any speedup compared to the single-threaded implementation?
- (c) For technical reasons, most Python implementations do not permit genuine parallelism when using multiple threads, so you probably didn't observe any speedup in part (b). However, there is a workaround for this. Instead of the `threading` module, use Python's `multiprocessing` package, which allows genuinely simultaneous computations by running separate threads in separate operating system processes. Rerun your experiments. What kind of speedups do you observe now?

**8.4** For each of the following computation trees, give a valid output of the computation, or alternatively state that the computation is undefined:



**8.5** Consider the program `ndFindNANA.py` (figure 8.5, page 149). Draw the computation tree for this program on the following input:

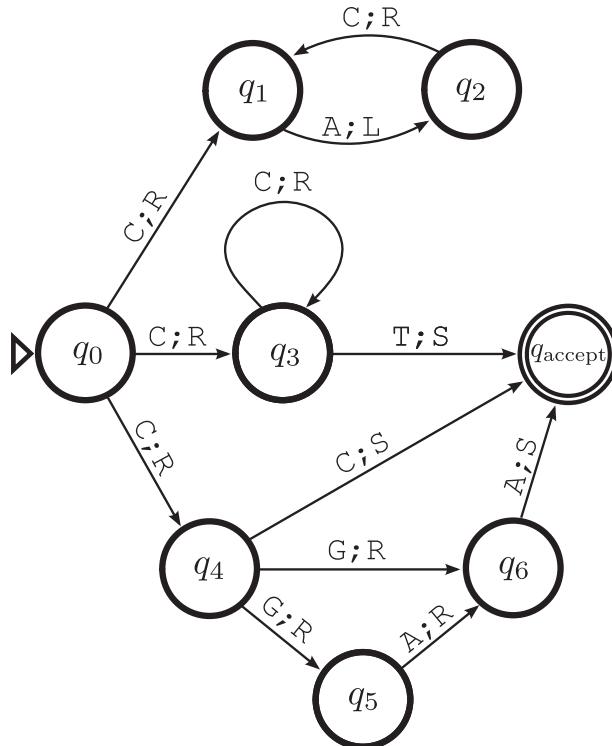
CCCCCCCCGGGGGGTATAGGGGGGGGGGGAGAAAA

**8.6** Let  $P_K$  be the nondeterministic program solving FACTOR with  $K$  threads, as described in exercise 8.3. (Note that you do not have to complete exercise 8.3 to attempt this question. In particular, you do not need to write the program  $P_K$ ; you only need to understand the definition of  $P_K$ .) Suppose we set  $K = 5$ , and run the program  $P_5$  with input  $M = 2030$ . Draw the computation tree for this computation.

**8.7** Create a nondeterministic Turing machine that searches genetic strings for one of the following three patterns: “CG...GC”, “CA...AC”, or “CT...TC”. Here, the “...” represents any string of zero or more characters. You may assume the input is flanked by “x” characters. So, your machine should accept inputs like “xGGTCGAAAGGCAAx” or “xCTTCx” and reject inputs like “xCCCAGCx” or “xCTCx”.

It is reasonably easy to construct a deterministic Turing machine solving the above problem, but the intention of this question is to practice using nondeterminism. Therefore, your machine should spawn a separate clone for each of the three search patterns.

**8.8** Create a nondeterministic Turing machine that accepts genetic strings whose length is a multiple of 3, 7, or 11, and rejects all other inputs.



**Figure 8.11:** The nondeterministic Turing machine  $M$  used in exercise 8.9.

**8.9** Let  $M$  be the nondeterministic Turing machine shown in figure 8.11.

- Which states of  $M$  exhibit nondeterminism?
- Draw the computation tree of  $M$  for input “CGAT”. What is the total number of clones involved in this computation? What is the output of the computation?
- Give examples of inputs for which the output of  $M$  is (i) *accept*; (ii) *reject*; (iii) *undefined*.
- Prove that  $M$  will launch only finitely many clones, regardless of the input.

**8.10** Would a multicore quantum computer be Turing equivalent to a standard Turing machine? Justify your answer.

**8.11** Prove that if  $L_1$  and  $L_2$  are recognizable languages, then  $L_1 \cup L_2$  is also a recognizable language.

**8.12** Define a decision problem  $D$  to be *nasty* if both  $D$  and  $\overline{D}$  are unrecognizable. This exercise leads you through a proof that a large class of problems are nasty. First we need some slightly weird notation. Given a decision problem  $D$ , define a new decision problem  $D\overline{D}$  as follows: The input to  $D\overline{D}$  consists of two strings  $I_1, I_2$ . The solution is “yes” if and only if  $D(I_1) = \text{“yes”}$  and  $D(I_2) = \text{“no”}$ . For a concrete example of this, consider  $D = \text{YESONEMPTY}$ . Then  $D\overline{D}$  is

a problem that we might call YESANDNOTYESONEMPTY, or YANYONEMPTY for short. YANYONEMPTY receives two input strings  $P_1, P_2$  describing Python programs. The solution is “yes” if and only if  $P_1(\epsilon) = \text{“yes”}$  and  $P_2(\epsilon) \neq \text{“yes”}$ . Your job for this exercise is to prove the following claim, which demonstrates that problems like YANYONEMPTY are indeed nasty:

**Claim 8.5.** Let  $D$  be a decision problem that is recognizable but undecidable. Then  $D\bar{D}$  is nasty.

# 9



## FINITE AUTOMATA: COMPUTING WITH LIMITED RESOURCES

The opinion of Descartes was much more extraordinary, for he made the souls of brutes to be mere automata.

—Joseph Priestley, *Disquisitions Relating to Matter and Spirit* (1777)

In computer science, the word *automaton* usually refers to a simple, abstract model of computation. Examples of different classes of automata include the standard Turing machines of chapter 5, the nondeterministic Turing machines of chapter 8, and the cellular automata described on page 109. In this chapter, we focus on yet another class of computational models, known as *finite automata*. Finite automata are, in some sense, the “simplest” nontrivial computers, and this is perhaps the main reason for studying them. From this simplest-possible model, we can understand some of the properties that other computational models may or may not possess. Finite automata come in two flavors: *deterministic* finite automata (dfas) and *nondeterministic* finite automata (nfas). We begin with an investigation of dfas.

### 9.1 DETERMINISTIC FINITE AUTOMATA

A dfa is just a particularly simple type of Turing machine: a dfa is a Turing machine whose head always moves to the right, and never changes the tape contents. The formal definition of a dfa looks very similar to the Turing machine definition on page 74. The main difference is that the transition function is much simpler—it doesn’t need to specify a new symbol to write on the tape, or the direction of the head movement. This can be formalized in a rigorous definition:

**Definition of a deterministic finite automaton (dfa).** A *dfa* consists of

- an alphabet  $\Sigma$  of symbols, including a blank symbol;
- a state set  $Q$ , including the start state  $q_0$ , and at least one of the halting states  $q_{\text{accept}}, q_{\text{reject}}$ ;

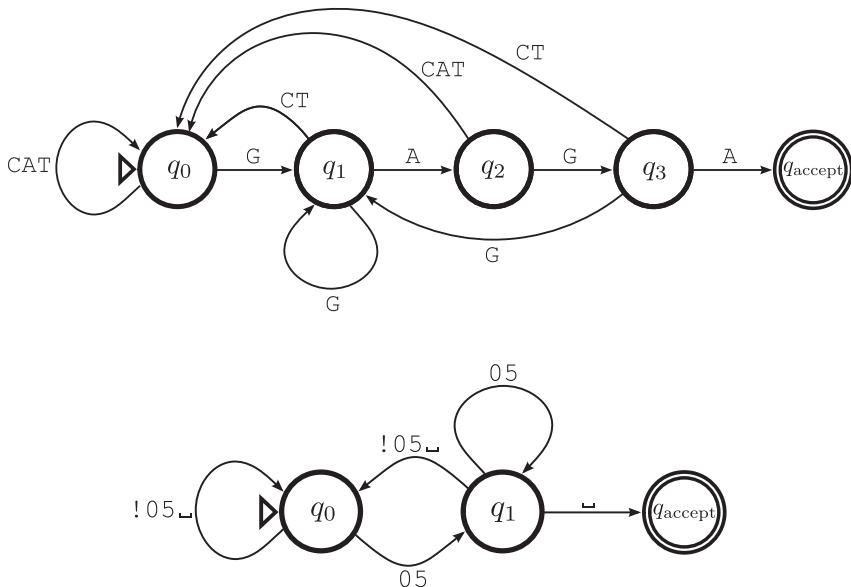


Figure 9.1: Examples of dfas. Top: containsGAGA. Bottom: multipleOf5.

- a transition function mapping  $(q, x) \mapsto q'$ , where  $q$  is a state,  $x$  is a symbol, and  $q'$  is the new state; the transition function is often denoted with the Greek letter  $\delta$  (delta), so we have

$$\delta(q, x) = q'.$$

In addition to the simplified transition function, there are two further minor differences between Turing machines and dfas. First, dfas are always accepters and never transducers, and therefore dfas never have a  $q_{\text{halt}}$  state. Second, the only purpose of the blank symbol is to terminate the input—so we may assume the dfa enters either  $q_{\text{accept}}$  or  $q_{\text{reject}}$  on reading a blank. Thus, dfas always halt after at most  $n$  steps, where  $n$  is the length of the input.

Dfas perform computations exactly as you would expect. We just think of them as Turing machines in which the head always moves to the right. By definition, the input to a dfa is a finite sequence of non-blank symbols like “abccba”, terminated by infinitely many blanks that we don’t bother to represent explicitly. The dfa begins in state  $q_0$ . Each step of the computation reads the current symbol of input, transitions to the appropriate state, and moves the head one cell to the right. And as already described above, the computation always halts with an accept/reject decision.

Dfas can be described using state diagrams, just as with Turing machines. Transitions are labeled only with the scanned symbol, since there’s no need to specify a new symbol or head direction. Figure 9.1 gives two examples. In the top panel, we see a dfa for our old favorite: containsGAGA. Note the similarity to the containsGAGA Turing machine in figure 5.5 (page 77). The bottom panel of figure 9.1 shows a dfa called multipleOf5. This dfa accepts strings

```

q0->q0: CAT
q0->q1: G
q1->q2: A
q1->q0: CT
q1->q1: G
q2->q3: G
q2->q0: CAT
q3->qA: A
q3->q0: CT
q3->q1: G

```

**Figure 9.2:** An ASCII description of the containsGAGA dfa, available in the book materials as `containsGAGA.dfa`.

like “652735” and “20”—nonnegative integers in decimal representation that are multiples of five. All other strings are rejected. In both of these examples, note the use of the same important convention we used for Turing machine state diagrams: if a transition is not explicitly represented, there is an implied transition into  $q_{\text{reject}}$ . Hence, if `containsGAGA` encounters a blank (i.e., the end of the input) before entering  $q_{\text{accept}}$ , then it halts and rejects. Similarly, if `multipleOf5` encounters a blank while in state  $q_0$ , it halts and rejects. We can see here the importance of assuming the input will be terminated with a blank symbol.

These two examples in figure 9.1 demonstrate that, despite their great simplicity, dfas can be used to solve computational problems such as `CONTAINSGAGA` and `MULTIPLEOF5`. We will be exploring the full computational power of dfas in section 9.4 onward.

Just as with Turing machine descriptions (see figure 5.19, page 96), we can agree on an ASCII format for the *description* of a dfa. The detailed specification of these descriptions is omitted, but figure 9.2 shows an example. As with Turing machines, dfa descriptions are not unique, but this presents no problems in practice. The book materials provide ASCII descriptions for all the finite automata in this chapter, using the “`.dfa`” file extension.

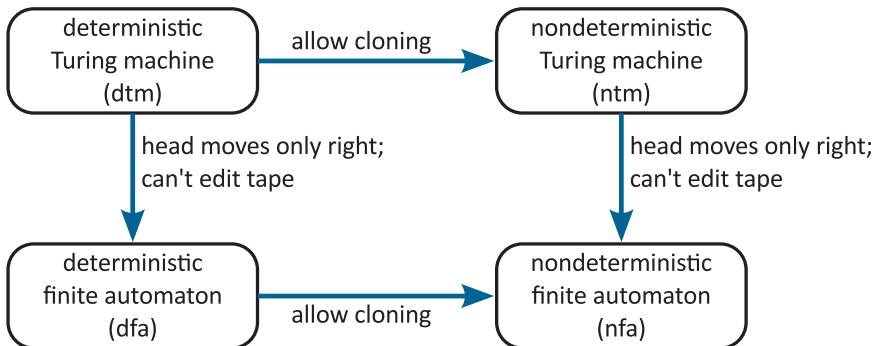
This would be a good time to do some explicit experiments and simulations of dfas. The Python program `simulateDfa.py` is provided for this purpose. For example, try the following command at an IDLE prompt:

```
>>> simulateDfa(rf('containsGAGA.dfa'), 'TTTGAGATTT')
```

You can also experiment with dfas using the “finite automaton” option in JFLAP. This is highly recommended, but you will need to take careful note of the next subsection, which discusses dfas without the blank symbol.

### Dfas without the blank symbol

This section covers a minor technicality, which you can skip unless you are comparing this material to another textbook or to JFLAP. Most other books and the JFLAP software package define dfas without using a blank symbol. In this alternative “no-blank” approach, multiple accepting states are sometimes needed. The output of the dfa is defined by processing all symbols in the input



**Figure 9.3: The relationship between Turing machines, finite automata, and nondeterminism.** Note, in particular, the two different routes to nfas in the bottom right. We can either (i) start with dfas in the bottom left, and add cloning (i.e., nondeterminism) or (ii) start with nondeterministic Turing machines in the top right, and restrict to machines that move only to the right and don't edit their tapes.

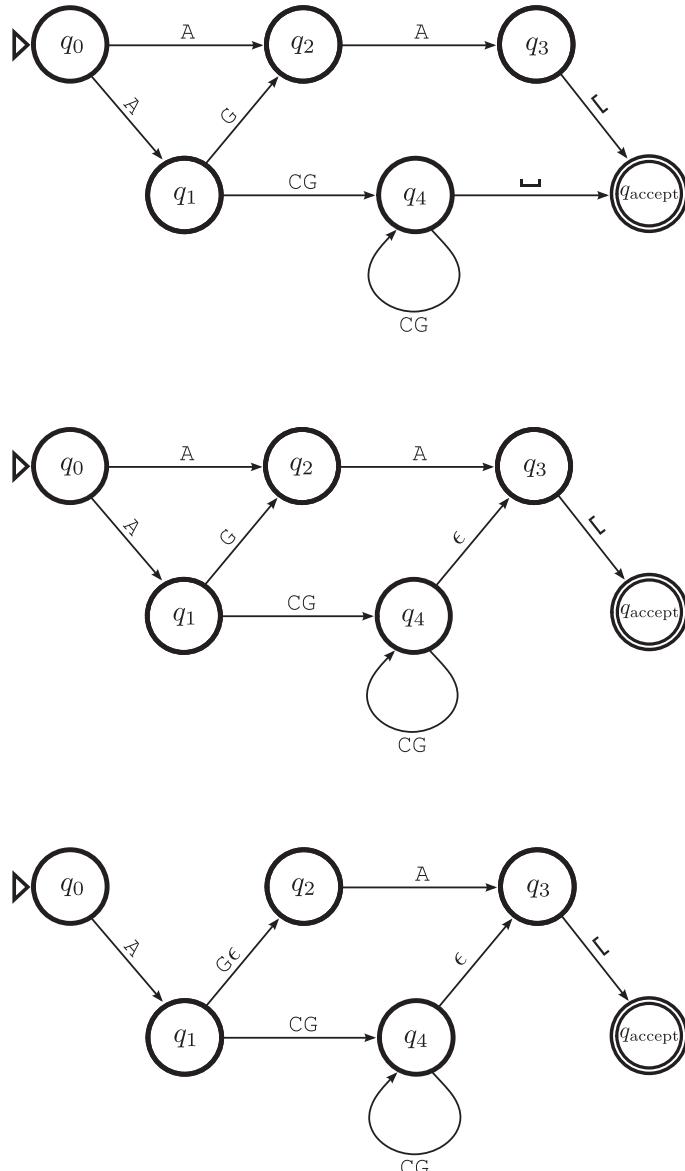
string, then accepting the string if the dfa halted in one of the accepting states. In particular, a “no-blank” dfa does *not* immediately halt on entering an accepting state—the entire input string must be processed before checking to see whether it was accepted. In this book, we use a blank symbol to terminate the input, but our dfas *do* halt immediately on entering an accepting state. The main advantage of this is that our dfas really are special cases of Turing machines, which also halt immediately on entering an accepting state.

#### *Equivalence of finite automata*

Recall from page 77 that two distinct Turing machines that are accepters are considered equivalent if they always produce the same decisions on the same input. Another way of stating this is that accepters are equivalent if they accept the same language. This definition of equivalence carries over to dfas (and nfas): two finite automata are *equivalent* if they accept the same language.

## 9.2 NONDETERMINISTIC FINITE AUTOMATA

There are two equally good ways of defining nondeterministic finite automata, as shown in figure 9.3. The figure also summarizes the relationships between some other automata we have studied. We start, in the top left, with Turing machines. These standard Turing machines can also be called deterministic Turing machines (dtms). Following the arrow to the right, we give dtms the ability to clone themselves, resulting in nondeterministic Turing machines (ntms). Alternatively, from dtms we can follow the down arrow, restricting to machines that move only right and don’t edit their tapes—this results in dfas. From dfas, we can follow the right arrow, adding the ability to clone and obtaining nondeterministic finite automata (nfas). Or, from ntms, we can follow the down arrow, restricting to nonediting machines that move only right, again obtaining nfads.



**Figure 9.4: Three equivalent NFAs.** All three NFAs accept the same language. The middle NFA includes one  $\epsilon$ -transition, and the bottom NFA includes two  $\epsilon$ -transitions. In the book materials, these NFAs are called `simple1.nfa`, `simple2.nfa`, and `simple3.nfa`.

### State diagrams for NFAs

Just as with the other automata we have studied, NFAs can be described by state diagrams. Figure 9.4 shows three examples that will all turn out to be equivalent. Let's initially focus on the first of the three graphs (available in the book materials as `simple1.nfa`). We see immediately that  $q_0$  uses nondeterminism: if the first

tape cell contains an “A”, the nfa can transition into either  $q_1$  or  $q_2$ . State  $q_1$  is also nondeterministic: if a “G” is scanned in  $q_1$ , the next state can be either  $q_2$  or  $q_4$ . All other states are deterministic. By manually checking all the possibilities, it’s easy to see that this nfa accepts the strings “AA”, “AGA”, and an “A” followed by any combination of C’s and G’s (provided there is at least one C or G).

It’s common to use another convenient notation on nfa state diagrams: a transition can be labeled with the empty string,  $\epsilon$ . This means that the nfa can nondeterministically follow the  $\epsilon$ -transition *without* moving the tape head. In other words, the machine clones itself, with one clone following the  $\epsilon$ -transition, and the other clone remaining behind. We see an example with the middle nfa of figure 9.4 (available as `simple2.nfa`). This nfa in fact accepts precisely the same language as the nfa above it. The only difference is an  $\epsilon$ -transition from  $q_4$  to  $q_3$ . The bottom nfa (`simple3.nfa`) of figure 9.4 shows yet another nfa that accepts the same language, this time using two  $\epsilon$ -transitions.

To reinforce your understanding of nfas, do some simulations and experiments now using the provided `simulateNfa.py` program. For example, you can try

```
>>> simulateNfa(rf('simple1.nfa'), 'ACCCCCG')
```

You can also experiment with nfas in JFLAP; simply create a finite automaton and include some nondeterministic transitions.

## Formal definition of an nfa

Although we could use either of the routes in figure 9.3, we will choose to follow the lower route and base our formal definition on dfas. (You will also notice a close resemblance to the definition of ntms on page 157.) We can write this formally:

**Definition of a nondeterministic finite automaton (nfa).** A *strict nfa* is defined in the same way as a dfa (page 164), except that the transition function maps to *sets* of states, rather than single states. An *nfa* is a generalization of a strict nfa, in which we also allow  $\epsilon$ -transitions.

For example, the top diagram of figure 9.4 represents a strict nfa, whereas the lower two diagrams represent nfas. The transition function for the top nfa of figure 9.4 maps  $(q_0, A)$  to the set of two possibilities  $\{q_1, q_2\}$ , since the machine will clone itself with one clone transitioning to  $q_1$ , and the other to  $q_2$ .

It’s easy to convert an nfa with  $\epsilon$ -transitions into a strict nfa. Exercise 9.7 asks you to investigate this in detail, but the key idea is that  $\epsilon$ -transitions create additional possibilities in the transition function. For example, in the bottom diagram of figure 9.4, the transition function maps  $(q_0, A)$  to the set of two possibilities  $\{q_1, q_2\}$ . The  $q_1$  comes from the explicit  $A$ -transition from  $q_0$  to  $q_1$ . Then, because the automaton can immediately follow the  $\epsilon$ -transition to  $q_2$ , the state  $q_2$  must also be included as a possibility. Compare this result with the strict nfa at the top of the figure.

It’s also worth noting another way of viewing  $\epsilon$ -transitions. Since an nfa is a special case of a nondeterministic Turing machine, we can view an  $\epsilon$ -transition as

the Turing machine transition “ $\sim ; s$ ”. Recalling the notation from page 78, this is a transition that is valid for any scanned symbol, changing from one state to another but keeping the head stationary.

### How does an nfa accept a string?

The definition of acceptance for an nfa is the same as for nondeterministic Turing machines on page 156. Specifically, the nfa

- *accepts* if *any* clone accepts;
- *rejects* if *all* clones reject;
- otherwise the output is undefined (this can happen only if there is a cycle of  $\epsilon$ -transitions, meaning that a clone can enter an infinite loop).

For a concrete example of this, reexamine figure 9.4. Convince yourself that all three nfas in figure 9.4 accept the same language, and are therefore equivalent.

### Sometimes nfas make things easier

One reason for working with nfas instead of dfas is that they can be simpler, easier, and more natural. Sometimes, a language is most easily described with a nondeterministic decision. As an example, consider the language

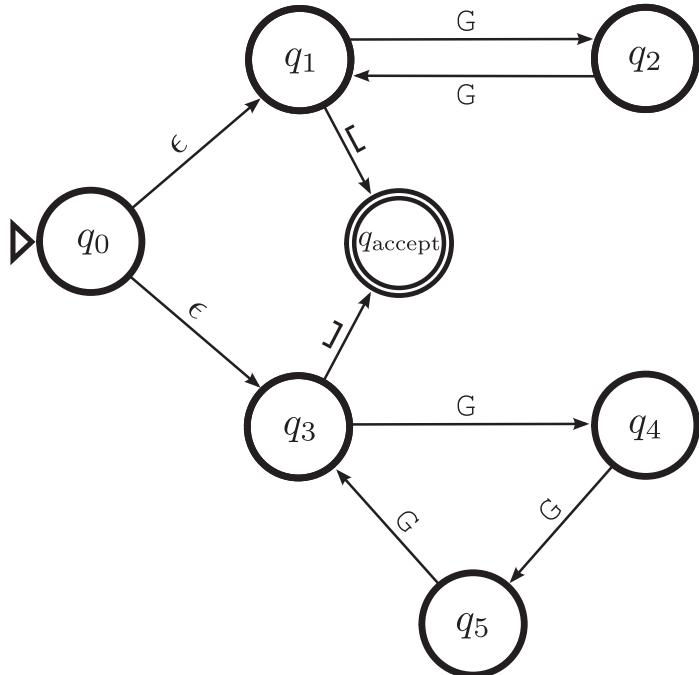
$$L = \{G^n : n \text{ is a multiple of 2 or 3}\}.$$

Figure 9.5 shows an nfa that accepts  $L$ . Notice how this `mult2or3Gs` nfa decomposes naturally into two possibilities: multiples of 2, and multiples of 3. The “multiples of 2” possibility is processed by states  $q_1$  and  $q_2$ ; the “multiples of 3” possibility is processed by states  $q_3$ ,  $q_4$ , and  $q_5$ . The nfa can choose nondeterministically between the two possibilities, processing both simultaneously. We will see below that we can also come up with a *deterministic* finite automaton that accepts  $L$ , but the deterministic version is more difficult to understand.

## 9.3 EQUIVALENCE OF NFAS AND DFAS

When we studied Turing machines, we discovered many features that could be added to the basic machine without increasing the computational power of the machine. Such features included multiple tapes, multiple heads, and the ability to clone machines via nondeterminism. In every case, the new type of Turing machine could solve exactly the same problems as the old one. So it’s natural to wonder whether adding nondeterminism to a dfa can add some computational power. As we will now see, the answer is again no.

Recall that two or more nfas and/or dfas are *equivalent* if they accept the same language. We have already seen, in figure 9.4, an example of three distinct, but equivalent, finite automata. We will now prove that every nfa has an equivalent dfa, demonstrating that the class of nfas has the same computational power as the class of dfas.



**Figure 9.5: The nfa `mult2or3Gs`.** It accepts strings of  $n$  G characters, where  $n$  is a multiple of 2 or 3.

**Claim 9.1.** Every nfa is equivalent to some dfa.

*Proof of the claim.* Given an nfa, we construct a dfa whose states consist of *subsets* of the nfa states. Transitions between these dfa states are computed by examining all possible transitions in the original nfa. The example below shows how this is done in practice. Figure 9.6 describes the algorithm in formal notation, but the notation tends to obscure the relatively simple ideas involved. It's probably easiest to work through the example below first, referring back to figure 9.6 only if necessary.

A formal proof of the algorithm's correctness needs to show that (i) the algorithm always terminates, and (ii) the resulting dfa  $D$  is in fact equivalent to the original nfa  $N$ :

- (i) **Termination.** The key point is that  $N$  has a finite number of states (say,  $K$ ), so the constructed dfa  $D$  also has a finite number of states (at most, the number of subsets of  $K$  elements, which is  $2^K$ ). The alphabet is also finite, so the loop at step 2 in figure 9.6 is guaranteed to terminate.
- (ii) **Equivalence of  $N$  and  $D$ .** It's not hard to check that  $D$  accepts a given string if and only if  $N$  does, and we omit the details. There is one minor technicality. If  $N$  contained a cycle of  $\epsilon$ -transitions, it is capable of entering an infinite loop and thus producing an undefined output. The construction above transforms this undefined behavior into a simple *reject*. If perfect “equivalence” were desired in this case, including the possibility

**Input:** Nfa  $N$  with states  $Q = \{q_0, q_1, q_2, \dots\}$ . Assume  $N$  uses implicit rejection, so  $q_{\text{reject}} \notin Q$ .

**Output:** Dfa  $D$  with states  $S = \{s_0, s_1, s_2, \dots\}$ .  $D$  will also use implicit rejection. Each  $s_i$  is a nonempty subset of  $Q$ .

Initially,  $S$  is empty.

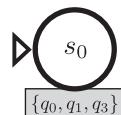
1. Define  $s_0$  to be the set of states  $N$  can reach without reading any symbols—in other words,  $s_0$  consists of  $q_0$  together with any  $q_i$  that can be reached via one or more  $\epsilon$ -transitions from  $q_0$ . Add  $s_0$  to  $S$ .  
Exception: If  $q_{\text{accept}} \in s_0$ , then  $N$  accepts all inputs. Treat this as a special case and return a dfa  $D$  that accepts all inputs.
2. While there remains any unprocessed transition  $(s_i, x)$ , where  $s_i \in S$  and  $x$  is an alphabet symbol, perform the following steps:
  - (a) Compute the set  $s$  of all  $N$ -states that can be reached in  $N$  from any element of  $s_i$  by reading an  $x$ -symbol. Make sure to include all possibilities of following  $\epsilon$ -transitions before and after the  $x$ -transition. Exceptions: (i) If  $q_{\text{accept}} \in s$ , redefine  $s$  as the singleton  $s_{\text{accept}} = \{q_{\text{accept}}\}$ . (ii) If  $s$  is empty, this is an implicit rejection, so return to step 2.
  - (b) If  $s \notin S$ , add  $s$  to  $S$  as a new state  $s_j$ . Otherwise we assume  $s$  corresponds to an existing state  $s_j$ .
  - (c) Add an  $x$ -transition in  $D$ , from  $s_i$  to  $s_j$ .

Figure 9.6: Algorithm for converting an nfa  $N$  to a dfa  $D$ .

of undefined output, then it's possible to instead produce a dfa with infinite loops in the appropriate places.  $\square$

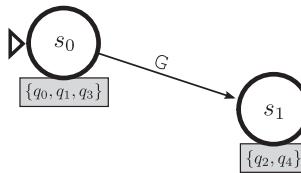
As an example, let's convert the `mult2or3Gs` nfa of figure 9.5 into a dfa.

**Step 1 of figure 9.6.** The initial state of the nfa is  $q_0$ , but because of the  $\epsilon$ -transitions out of  $q_0$ , the nfa can clone itself and start in any one of the three states  $\{q_0, q_1, q_3\}$  before reading any characters of input. Therefore, we create an initial state for our dfa, call it  $s_0$ , and label it with “ $\{q_0, q_1, q_3\}$ ” to remind us that it represents this subset of states in the original nfa:

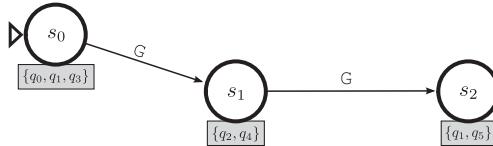


**Step 2 of figure 9.6, first iteration.** This iteration will be applied to the state  $s_0$  and the symbol `G`. So we ask ourselves, given that the nfa is in one of the states  $\{q_0, q_1, q_3\}$ , which states could it transition to after reading a `G`? From  $q_1$ , it can go to  $q_2$ ; from  $q_3$ , it can go to  $q_4$ ; and these are the only possibilities (other than  $q_{\text{reject}}$ , which is deliberately excluded). So we create a new state for our dfa, call it  $s_1$ , and label it with “ $\{q_2, q_4\}$ ” to remind us

that it represents this subset:



**Step 2 of figure 9.6, second iteration.** This iteration will be applied to the state  $s_1$  and the symbol  $G$ . Beginning in  $q_2$  or  $q_4$  and consuming another  $G$  of input moves us either to  $q_5$  or back to  $q_1$ , so we add  $s_2 = \{q_1, q_5\}$ , giving



Continuing iterations of step 2, and for the moment considering only  $G$ -transitions, we eventually get to the diagram in figure 9.7(a). A common mistake is to create a new state  $s_j$  that is the same as some existing state  $s_i$ . In this example, when computing the transitions from  $s_6$ , we get the set of possibilities  $\{q_2, q_4\}$ . But we don't define a new state  $s_7 = \{q_2, q_4\}$ , because this would be identical to the already-existing  $s_4$ .

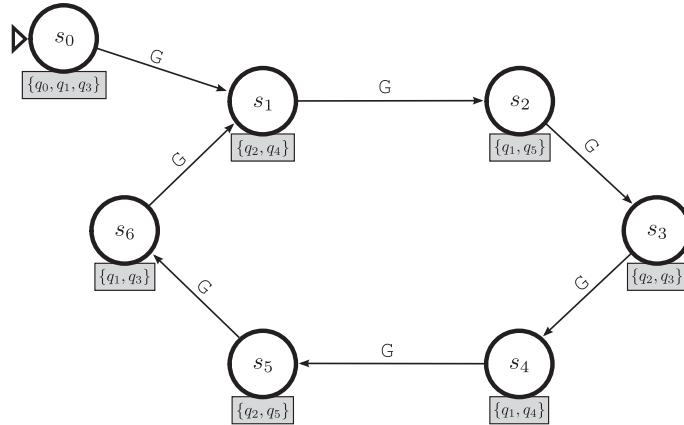
So far, we have considered only the possibility of reading  $G$ 's. Now we need to continue iterating step 2 of figure 9.6, considering transitions from all  $s$ -states via all other possible symbols in the alphabet. But looking back to figure 9.5, we see that for this particular nfa, the only other symbol that needs to be considered explicitly is the blank symbol. (Any other symbol results in an empty  $s$ -state, which gets discarded at the end of step 2a.) So, consider the effect of reading a blank symbol from  $s_0$ . This leads to the new state  $s_{\text{accept}} = \{q_{\text{accept}}\}$ , since  $q_1$  and  $q_3$  both lead to  $q_{\text{accept}}$  in the nfa. Applying the same reasoning to the remaining  $s_i$  produces the final output of the algorithm, shown in figure 9.7(b).

Figure 9.8 provides another example of converting an nfa to a dfa using this algorithm. Try it out for yourself: starting with the nfa `example2.nfa` in the top panel of this figure, run the algorithm to completion and check that you obtain the dfa `example2.dfa` in the bottom panel. You can also run simulation experiments using the provided files.

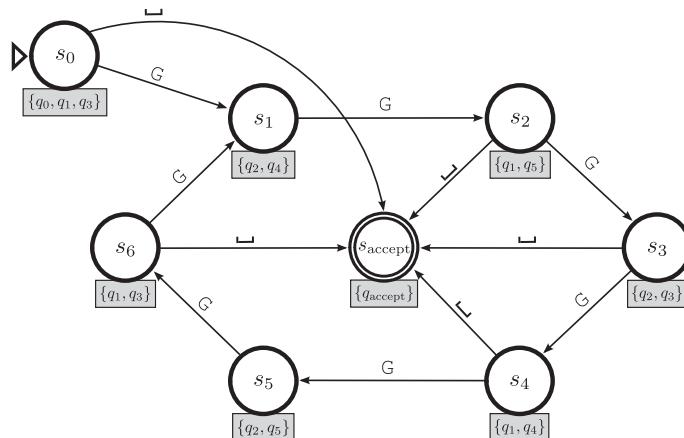
## Nondeterminism can affect computability: The example of pdas

The only computational models studied in detail in the main body of this book are Turing machines and finite automata. In both cases, we now know that adding nondeterminism to the model does not change the set of problems that the model can solve. Based on this, you might guess that nondeterminism never affects computability for a given model of computation. But this guess would be wrong.

Another important and frequently studied model is the *pushdown automaton* (pda). A description and analysis of pdas is available as an online supplement to



(a) Result after several iterations of step 2



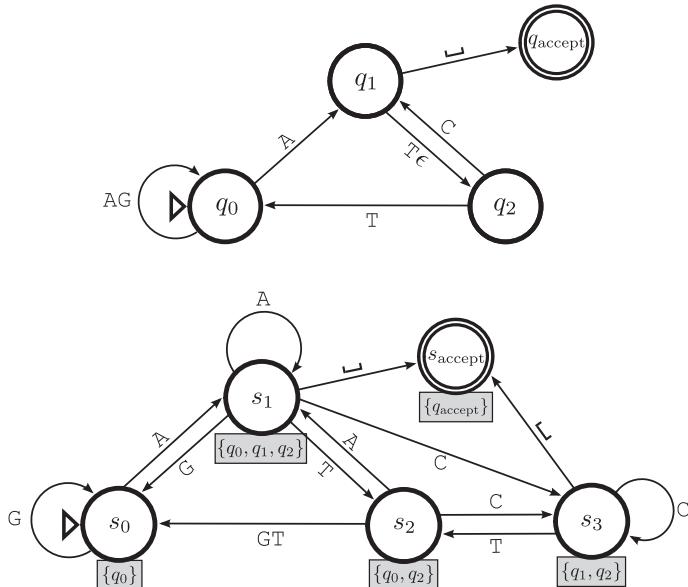
(b) Final result

**Figure 9.7:** Steps in an example converting an nfa to a dfa.

this book. But pdas are worth mentioning here because they provide an extremely interesting example of the effect of nondeterminism. Specifically, it turns out that nondeterministic pdas can solve a larger class of computational problems than deterministic pdas. Therefore, nondeterminism *does* affect computability for some computational models.

### Practicality of converted nfas

When we convert an nfa into an equivalent dfa, the result is often a compact, practical dfa that could be used in real computer programs. In principle, however, an nfa with  $K$  states can result in an equivalent dfa of up to  $2^K$  states. For large values of  $K$ , this kind of exponentially large dfa is not practical. Therefore, just as with Turing machines, it is clear that adding nondeterminism to dfas does not change computability, but it does have an effect on practicality.



**Figure 9.8: A second example of converting an nfa to a dfa.** *Top:* The nfa example2.nfa. *Bottom:* An equivalent dfa example2.dfa, generated using the algorithm of figure 9.6.

It's worth noting that almost everywhere else in this book, we are equating “practicality” with “reasonable running time.” In the case of dfas and strict nfas, however, the running time is at most  $n$  steps for input of length  $n$ —the automaton reads each character of the input at most once, then halts. So the running time of a dfa is always “reasonable.” In the previous paragraph, therefore, we instead equated “practicality” with “reasonable size.”

### Minimizing the size of dfas

Dfas are not unique: given any dfa  $M$ , there are many other dfas that decide the same language as  $M$ . And as we just discussed, all of these dfas have the same running time for the same input. But some of them have many more states than others. So it's natural to ask, given a dfa  $M$ , can we find an equivalent dfa  $M'$  that is as small as possible? Specifically, can we easily find an  $M'$  with a minimal number of states? The answer is yes, there is an efficient algorithm for minimizing dfas. We won't be studying the minimization algorithm here, but it is worth noting that the algorithm is important and useful. One of the key applications is in creating efficient implementations of regular expressions, which are discussed next.

## 9.4 REGULAR EXPRESSIONS

Professional software developers and computer scientists often use a tool known as *regular expressions* for various frequently encountered tasks. The phrase “regular

expression” is often abbreviated to “regex” or “regexp.” Examples of regexes in action include the following:

- You could use regular expressions to find all files on your computer that contain digits in their filenames.
- More generally, the UNIX command line utility `grep` and the equivalent `findstr` on Windows use regular expressions to efficiently search text for certain kinds of patterns.
- Many development environments for programming allow you to search using regular expressions in their advanced search options.
- Many programming languages (including Java and Python) provide libraries that let you employ regular expressions in your own programs.

So, it’s clear that regexes are a valuable tool. But what do they have to do with the theory of computation? It turns out that finite automata are closely related to regular expressions. In fact, any finite automaton is equivalent to some regular expression and vice versa. To build our understanding of this, we will first define regular expressions, and then examine their equivalence to finite automata.

## Pure regular expressions

Initially, we will define a particular class of regular expressions that we call *pure* regular expressions. (“Pure” is not a standard term employed by other books in this context, but it will be useful for us.) Let  $\Sigma$  be an alphabet. For convenience, we will assume that  $\Sigma$  does not contain any of the following four symbols:  $| ()^*$ . For a concrete example, let’s take  $\Sigma$  to be the set of alphanumeric ASCII characters a–z, A–Z, and 0–9.

Every regular expression  $r$  will represent a language on  $\Sigma$ . We start off with certain *primitive* regular expressions:

- $\emptyset$  represents the empty language,  $\{\}$ .
- $\epsilon$  represents the language containing the empty string,  $\{\epsilon\}$ .
- Any symbol  $s \in \Sigma$  represents the language  $\{s\}$ , containing exactly one string of length 1.

We can build up more interesting pure regular expressions by applying the operations of concatenation, alternatives, and repetition. This would be a good time to review the definitions of the language operations on page 52. In particular, recall that the repetition operation is technically known as the *Kleene star*, and also carefully note the definition of concatenation.

To explain these operations in the context of regular expressions, suppose  $r_1, r_2$  are regular expressions representing languages  $L_1, L_2$  respectively. Then we can create new regular expressions via the following three operations:

- **concatenation:**  $r_1r_2$  represents the language  $L_1L_2$ ; for example, the regex CG represents the language {CG}.
- **alternatives:**  $r_1|r_2$  represents the language  $L_1 \cup L_2$ ; for example, the regex CG | GGGTA represents the language {CG, GGGTA}.
- **Kleene star:**  $r_1^*$  represents  $L_1^*$ —strings from  $L_1$  repeated zero or more times in arbitrary combinations; for example, the regex (A | C) $^*$  represents the language { $\epsilon$ , A, C, AA, AC, CA, CC, AAA, ...}.

By themselves, these rules are ambiguous. For example,  $A|CG$  could represent  $\{A, CG\}$  or  $\{AG, CG\}$ , depending on whether we read the regular expression as  $A|(CG)$  or  $(A|C)G$ . This ambiguity is removed by declaring that the Kleene star (“ $*$ ”) has the highest precedence, followed by concatenation, and finally alternatives (“ $|$ ”). Parentheses can be used to enforce a different precedence when desired. For example,  $AC^*$  represents  $\{A, AC, ACC, \dots\}$  because the  $*$  has higher precedence than the concatenation between  $A$  and  $C$ . But  $(AC)^*$  overrides this precedence, resulting in  $\{\epsilon, AC, ACAC, ACACAC, \dots\}$ .

The easiest way to understand pure regular expressions is with a few examples:

Pure regex $r$	Language represented by $r$
$C AG GGT$	$\{C, AG, GGT\}$
$(C A)(G TT)$	$\{CG, CTT, AG, ATT\}$
$C^*$	$\{\epsilon, C, CC, \dots\}$
$CG^*T$	$\{CT, CGT, CGGT, \dots\}$
$(CG)^*T$	$\{T, CGT, CGCGT, \dots\}$
$(C G)^*$	$\{\epsilon, C, G, CC, GG, CG, GC, \dots\}$
$C^* G^*$	$\{\epsilon, C, G, CC, GG, CCC, GGG, \dots\}$
$(AC^*T)^*$	$\{\epsilon, AT, ACT, ACCT, ATAT, ACTACT, \dots\}$

## Standard regular expressions

As already mentioned, we refer to the regular expressions defined above as *pure* regular expressions. This is because they use a minimal set of operators—only  $*$ ,  $|$ , and parentheses. Real implementations typically allow a much richer set of features in regular expressions. We list a few of the most useful features:

- A “.” (period) represents any single character in the alphabet. For example,  $G.^*G$  is the language of all strings beginning with a  $G$  and ending with another  $G$ .
- A “+” is the same as  $*$ , except that there must be at least one repetition. For example,  $(CA)^+$  is the same as  $(CA)^*$ , except that the empty string is not included.
- Certain sets of characters can be described using a flexible square bracket notation that allows lists of characters and ranges of characters. For example,  $[CGT]$  is the same as  $C|G|T$ ; the expression  $[A-Z]$  represents any single uppercase letter; and  $[A-Za-z0-9]^*$  represents any alphanumeric string.

Because these extra features are so common, let’s refer to regexes that use these features as *standard* regular expressions. Any standard regex can be converted into a pure regex. This should be clear from the following examples, which use the alphabet  $\Sigma = \{C, A, G, T\}$ .

Standard	Pure
.	$C A G T$
$(CA)^+$	$CA(CA)^*$
$[CGT]$	$C G T$

So, standard regexes tend to be shorter—they are easier to read, write, and use. Any practical regex tool will permit the standard features described here

and many more besides. But for theoretical purposes, pure regexes are easier to analyze since there are fewer operators to consider. Therefore, we consider only pure regexes from now on.

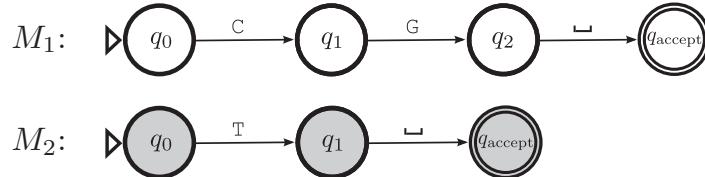
## Converting between regexes and finite automata

As mentioned earlier, every regex is equivalent to some dfa, and vice versa. There do exist formal algorithms for converting regexes to dfas, and dfas to regexes. But we will not study these conversion algorithms here. Instead, we examine the main ideas behind these conversions via simple examples.

### Converting regexes to nfas

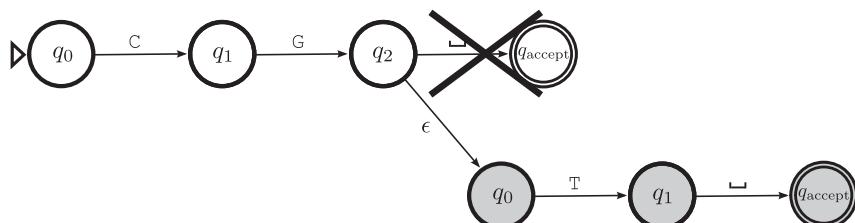
Note that we already know how to convert nfas to dfas (see claim 9.1, page 171). So even if our ultimate goal is to convert regexes to dfas, it is sufficient to initially produce an nfa and then apply the nfa-to-dfa conversion. In a real implementation, we would then apply the dfa-minimization algorithm mentioned above (page 175) to produce efficient code for recognizing strings that match a given regex.

The main elements of regexes are concatenation, alternatives, and the Kleene star. Let's see how to emulate each of these using an nfa. Specifically, suppose we already have regexes  $r_1$  and  $r_2$ , with equivalent dfas  $M_1$  and  $M_2$  respectively. We want to come up with new nfas that are equivalent to  $r_1r_2$ ,  $r_1|r_2$ , and  $r_1^*$ . Our descriptions will be valid for any  $r_1$ ,  $r_2$ , but let's use a running example to assist understanding. We'll take  $r_1 = CG$  and  $r_2 = T$ , with the corresponding dfas  $M_1$  and  $M_2$ :



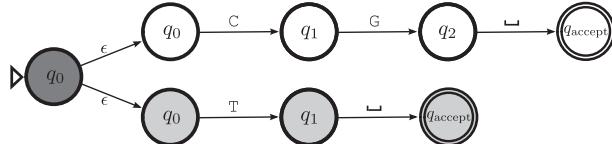
Here's how we implement each of the regex operations:

- **concatenation:** To produce an nfa equivalent to  $r_1r_2$ , any transitions into  $M_1$ 's accept state get converted into  $\epsilon$ -transitions that enter  $M_2$ 's start state:



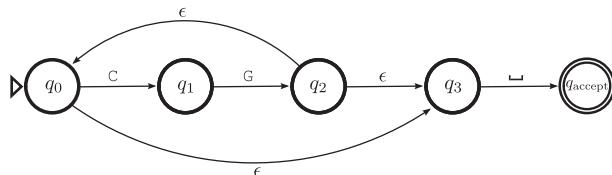
We would also need to relabel the states with unique names, but that step isn't shown here.

- **alternatives:** To produce an nfa equivalent to  $r_1|r_2$ , create a new start state with  $\epsilon$ -transitions to the start states for  $M_1$  and  $M_2$ :



To meet our formal definition of an nfa we would need to relabel the states uniquely and merge the separate accept states into a single state. Again, these superficial changes are not shown.

- **Kleene star:** To produce an nfa equivalent to  $r_1^*$ , create a new state and three new  $\epsilon$ -transitions as in the following example:



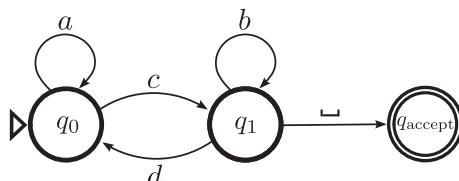
By applying these three techniques repeatedly, it turns out that any regular expression can be converted into an nfa.

### Converting dfas to regexes

The three techniques above can essentially be applied in reverse to convert a dfa to a regex. The key idea is to gradually simplify the dfa, labeling edges with *regular expressions* rather than single symbols. But the details can be fiddly, and we don't study them systematically. Simple dfas can usually be converted using common sense, especially after you have studied the example in figure 9.9. The top panel is a dfa available in the book resources as `GACetc.dfa`. By following the suggested steps, we arrive at the bottom panel, where we can read off the equivalent regex as

$$(GAC^*T)^*G(\epsilon \mid AC^*)$$

In the general case, this simplification process always boils down to a diagram of the following form, perhaps with some edges removed:



Here,  $a, b, c, d$  are arbitrary regexes. The equivalent regex can always be read off this diagram as

$$(a^*cb^*d)^*a^*cb^*$$

This expression can be adapted in obvious ways if one or more edges are missing. For example, if the  $d$ -edge is missing, the equivalent regex is  $a^*cb^*$ .

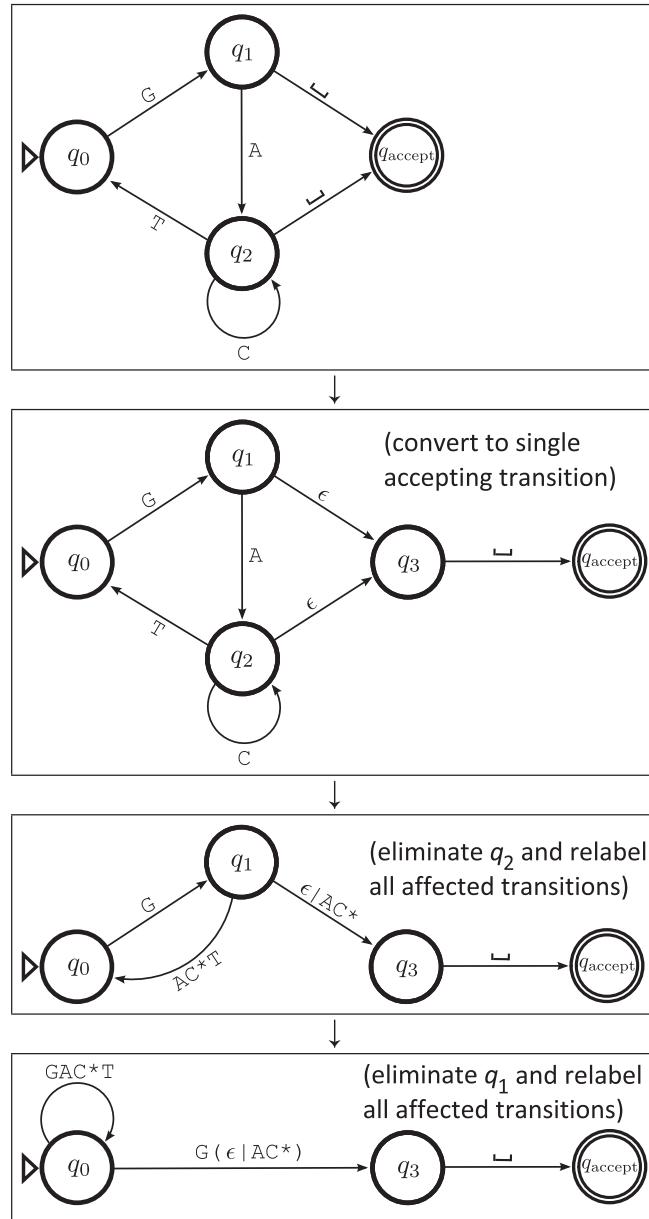


Figure 9.9: Example of converting a dfa to a regex.

The equivalence of finite automata and regular expressions has been mentioned several times already, and now we formalize it as a claim. Formally, a dfa or nfa  $M$  is *equivalent* to a regular expression  $r$  if the language accepted by  $M$  is the same as the language represented by  $r$ .

**Claim 9.2.** Given any dfa or nfa, there is an equivalent regular expression. Given any regular expression, there is an equivalent dfa.

*Sketch proof of the claim.* We don't attempt a rigorous proof. Instead, note that the previous few pages sketched the techniques for converting any dfa into a regular expression and vice versa. These techniques can in fact be fleshed out into detailed algorithms and proved correct, but we don't investigate those details here.  $\square$

## 9.5 SOME LANGUAGES AREN'T REGULAR

Recall that a language is *decidable* if it is decided by some Turing machine (or Python program, or other Turing-equivalent computer). The analogous concept for dfas is a *regular* language:

**Definition of a regular language.** A language is *regular* if and only if it is decided by some dfa. Equivalently, a language is regular if it's decided by some nfa, or if it's represented by some regular expression.

There is no need to give more examples of regular languages—we have seen so many already. Every nfa, dfa, and regex in this chapter corresponds to some regular language.

A fundamental question now arises: Is there any difference between decidable languages and regular languages? In other words, are there any languages that can be decided by a Turing machine, but not by a finite automaton? This is the main question addressed in this section. And we will soon see that the answer is yes: Turing machines can decide more languages than finite automata. This means that finite automata are a strictly weaker computational model than Turing machines. In contrast to all of the Turing-equivalent models studied in chapter 5, we have finally found a model that is not equivalent to the Turing machine!

But before we dive into a proof of this fact, let's address a much easier question: Could a language be decidable by a finite automaton, but *not* by a Turing machine? The answer to this is clearly no, because dfas are a special class of Turing machines. The following claim summarizes this argument.

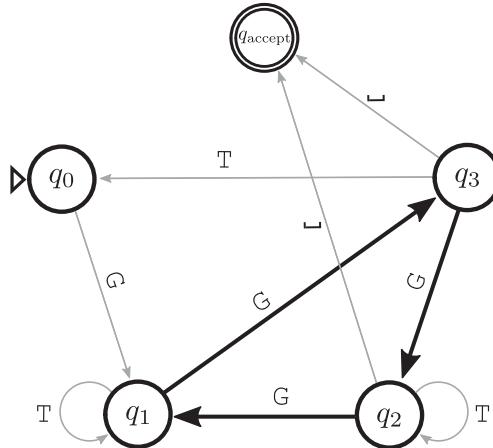
**Claim 9.3.** If a language is regular, it is also decidable.

*Proof of the claim.* Let  $L$  be a regular language. Then it is decided by some dfa  $M$ . But  $M$  is also a Turing machine. (Recall that a dfa is just a Turing machine in which the head always moves to the right and the tape is never altered.) So  $L$  is decided by a Turing machine, and the claim is proved.  $\square$

### The nonregular language GnTn

Now we can move on to the more interesting fact that there exist nonregular, decidable languages. We will initially focus on just one of these nonregular languages, called GnTn. It's defined as follows:

$$\begin{aligned} \text{GnTn} &= \{G^n T^n : n \geq 0\} \\ &= \{\epsilon, GT, GGTT, GGGTTT, \dots\}. \end{aligned}$$



**Figure 9.10: A visualization of the proof that  $\text{GnTn}$  is not regular.** A hypothetical 5-state dfa  $M$  is shown. The bold arrows show a cycle of 3 G's encountered while processing the string “GGGGGGTTTTTT”, which is eventually accepted. Because we can traverse the cycle any number of times while still eventually reaching  $q_{\text{accept}}$ , we know that  $M$  must also accept the strings “GGGTTTTT”, “GGGGGGGGTTTTTT”, and so on.

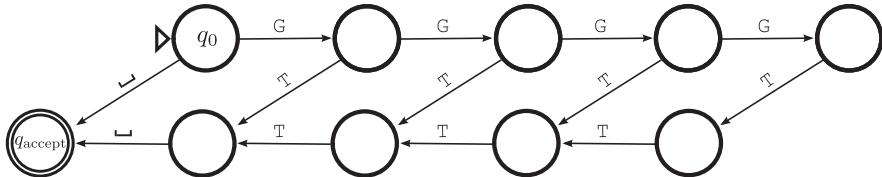
We can easily write a Python program that decides  $\text{GnTn}$ , so this is certainly a decidable language. (See `GnTn.py` in the book resources, or write your own version as an easy Python exercise.) We now prove that  $\text{GnTn}$  isn't regular:

**Claim 9.4.**  $\text{GnTn}$  is not a regular language.

*Proof of the claim.* We assume that  $\text{GnTn}$  is a regular language, and argue for a contradiction. By our assumption,  $\text{GnTn}$  is regular, so it is decided by some dfa  $M$ . Of course  $M$  has some fixed and finite number of states—say, a total of  $K$  states including the accept state. Now consider the particular string  $S = G^{K+1}T^{K+1}$ , consisting of  $K+1$  “G” characters followed by  $K+1$  “T” characters.  $\text{GnTn}$  contains  $S$ , so we know that  $M$  accepts  $S$ . Figure 9.10 shows one hypothetical dfa  $M$  with  $K=5$  states. The corresponding string under consideration is  $S = \text{GGGGGGTTTTTT}$ . Of course, figure 9.10 is just to help us visualize the situation: the given  $M$  does not in fact decide  $\text{GnTn}$ , and in any case the proof must work for any  $M$ .

Let's now focus on what happens when any possible  $M$  processes the *first half* of  $S$ —the part with all the G's in it. In particular, are there any states that get visited twice? The answer is yes: there are only  $K$  states in  $M$ , but there are  $(K+1)$  G's. So by the time  $M$  has processed the G's, at least one of the states must have been visited twice. And this has an important consequence: the sequence of states that have been visited so far must contain a cycle. Why? Say  $q$  is a state that was visited twice, and think about what happens immediately after  $M$  visits  $q$  for the first time:  $M$  eventually comes back to  $q$ , completing a cycle.

Let  $L$  be the length of this cycle, and recall that while  $M$  was visiting the states in the cycle, it was processing *only* G characters. In other words, whenever  $M$  arrives at state  $q$ , it can proceed to read  $L$  further G's and return to exactly the same state,  $q$ . This means we can insert the substring  $G^L$  into the first half of  $S$  without

Figure 9.11: A dfa that decides  $G^n T^n$  when  $n \leq 4$ .

affecting the computation— $M$  will continue on from  $q$  in exactly the same way, regardless of how many times it goes around the cycle. In particular,  $M$  should reach the same result when it processes  $S = G^{K+1} T^{K+1}$  and  $S' = G^{K+1+L} T^{K+1}$ . (Study the particular example of figure 9.10 to help your understanding of this. Note we could also have skipped the cycle altogether, resulting in another accepted string  $S'' = G^{K+1-L} T^{K+1}$ .)

We have now arrived at a contradiction. Recall that  $M$  decides  $G^n T^n$ . So it should accept  $S$  (which is in  $G^n T^n$ ) and reject  $S'$  (which is not in  $G^n T^n$ ). But this contradicts the previous conclusion, that  $M$  produces the same result for  $S$  and  $S'$ .  $\square$

### The key difference between Turing machines and finite automata

The previous claim encapsulates the most important idea of this chapter: because there exist languages like  $G^n T^n$  that are decidable but not regular, we know that Turing machines are strictly more powerful than finite automata. However, it's worth investigating the distinction between Turing machines and finite automata in more detail. What is the essential characteristic of dfas that prevents them from deciding  $G^n T^n$ ?

We can answer this question by trying to build a dfa that decides  $G^n T^n$ . Admittedly, we already know it's an impossible task. But if we ignore that and make the attempt anyway, we end up with something like the dfa in figure 9.11. This dfa works correctly when  $n \leq 4$ . And it's obvious we could repeat the pattern, extending the dfa to the right as much as desired so that it works for all elements of  $G^n T^n$  up to any fixed value of  $n$ . But the dfa must have some fixed, finite number of states. In some sense, the size of its “memory” is limited to its number of states. So it can never “remember” seeing more than some fixed number of  $G$ 's. When this limit is exceeded, the dfa fails. In summary, it is the *finite memory* of a dfa that makes it more limited than a Turing machine.

But Turing machines have only a finite number of states also. Why don't they suffer from the same problem? It's because they can use the tape to remember as much information as desired. Indeed, a Turing machine can record infinitely many distinct strings on its tape. So although it has a finite set of *states*, it has infinitely many possible *configurations*.

## 9.6 MANY MORE NONREGULAR LANGUAGES

$G^n T^n$  is not alone—it turns out that there are many nonregular languages. In this section, we will study a tool called the pumping lemma, which can be used to

prove that a language isn't regular. To get started on this, look back to the proof of the claim that  $G^nT^n$  isn't regular (claim 9.4, page 182). There we saw that because of a cycle of  $G$ -transitions, the dfa would need to accept both  $S = G^{K+1}T^{K+1}$  and  $S' = G^{K+1+cL}T^{K+1}$ . But if we examine this proof more carefully, we see that there is no reason to go around the cycle exactly once or twice. We can choose to go around the cycle zero times, or three times, or four times, and so on. In general, traversing the cycle  $c$  times would show that the dfa must accept the string  $G^{K+1+cL}T^{K+1}$ —for any integer  $c \geq 0$ . Focusing on the particular example of figure 9.10, this would require the dfa to accept all of the following strings:

GGGTTTTTT  
GGGGGGGTTTTTTT  
GGGGGGGGGGTTTTTT  
GGGGGGGGGGGGTTTTTT  
  
⋮

This process is called *pumping* the string  $S$ . It involves choosing a substring of  $S$  (in this case “GGG”), and replacing it with zero, one, or more copies of itself.

To make our proofs work, we are normally interested in pumping with a substring before some fixed point in the original string. For example, in the proof that  $G^nT^n$  isn't regular, we knew that the substring must occur in the first half of  $S$ . This motivates the following definition:

**Definition of “pumping before  $N$  in  $L$ .”** Let  $L$  be a language and let  $S$  be a string in  $L$ . Fix an integer  $N > 0$ . We say that  $S$  can be *pumped before  $N$  in  $L$*  if  $S$  has a nonempty substring  $R$  with the following two properties:

- $R$  finishes before or at the  $N$ th symbol of  $S$ .
- Replacing  $R$  with any number of copies of itself (including zero copies) results in a string that is still a member of  $L$ .

To specify the substring  $R$ , we say that  $S$  *can be pumped with  $R$* , or  $R$  *can be used for pumping  $S$* .

The value of  $N$  in this definition can be called the *pumping cutoff*, or just the *cutoff*. When the language  $L$  is clear from the context, we can omit the “in  $L$ .” Similarly, when the cutoff  $N$  is clear from the context, we can omit “before  $N$ .”

To see some examples of pumping, take  $L$  to be the language generated by the regex  $ATC^*|TG(AC)^*GT$ . Then strings such as “A”, “AT”, and “TGGT” cannot be pumped at all, regardless of the chosen cutoff. The strings “ATC” and “ATCC” can be pumped before the cutoff 4, by pumping with the one-character substring “C”. And the strings “TGACGT” and “TGACACGT” can also be pumped before the cutoff 4, by pumping with the substring “AC”. See figure 9.12 for more detailed examples. Note that the “number of pumps” refers to the number of copies of the substring  $R$  that are left after the pumping has finished. So, pumping zero times is equivalent to deleting  $R$ , while pumping twice is equivalent to adding one additional copy of  $R$ .

String $S$	Number of pumps			Valid cutoffs
	0	1	2	
AT <u>CC</u>	ATC	ATCC	ATCCC	$N \geq 3$
<u>AT</u> CC	AT	ATCC	ATCCCC	$N \geq 4$
TG <u>ACACGT</u>	TGACGT	TGACACGT	TGACACACGT	$N \geq 4$
<u>TGACACGT</u>	TGGT	TGACACGT	TGACACACACGT	$N \geq 6$
TGGT	no substring can be used for pumping			None

**Figure 9.12:** Examples of pumping strings in the language represented by the regex  $\text{ATC}^*|\text{TG(AC)}^*\text{GT}$ . In the left column, the pumped substring of  $S$  is underlined. The right column shows the cutoff values  $N$  for which the underlined substring can be used for pumping.

## The pumping lemma

One of the most interesting properties of the regular languages is that, loosely speaking, they can always be pumped. There are two caveats here: first, finite languages can't be pumped; and second, short strings can't always be pumped. But for an infinite regular language, all sufficiently long strings in the language can be pumped before some fixed cutoff. This result is normally called the *pumping lemma*:

**Claim 9.5 (Pumping lemma).** Let  $L$  be an infinite regular language. Then there is some cutoff  $N$  such that all strings in  $L$  of length at least  $N$  can be pumped before  $N$  in  $L$ .

*Proof of the claim.* The proof closely mirrors the proof of the claim that GnTn isn't regular (claim 9.4, page 182). Because  $L$  is regular, it's decided by some DFA  $M$ . And  $M$  has a fixed number of states, say  $K$  states. We will take as our pumping cutoff  $N = K + 1$ . We have to show that all strings in  $L$  with length at least  $K + 1$  can be pumped before  $K + 1$ . So, let  $S$  be a string in  $L$  of length at least  $K + 1$ . What happens as  $M$  processes the first  $K + 1$  symbols of  $S$ ? There are only  $K$  states, which means that at least one of the states must be visited twice, which means there must be a cycle of length  $\leq K$ . The symbols on this cycle are a substring that can be used for pumping  $S$  before  $K + 1$ , so the proof is complete.  $\square$

There are two technical remarks to make here:

1. In the proof of claim 9.4 on page 182, the cycle consisted entirely of G-transitions. In the above proof, the cycle could consist of any sequence of symbols. But otherwise the reasoning is identical—see figure 9.10.
2. Where did we use the fact that  $L$  is infinite? Actually, we didn't really need this fact. The claim is vacuously true for finite languages, since we can choose a cutoff greater than the longest string in  $L$ . But to guarantee the existence of the string  $S$  in the proof above, we need to know that there are strings in  $L$  of length at least  $K + 1$ , and this is certainly true if  $L$  is infinite.

The main application of the pumping lemma is in proving that certain languages are not regular. The high-level strategy is to use a variant of the

pumping lemma that logicians call its *contrapositive*: instead of using the fact that “regular languages can be pumped,” we use the equivalent statement “if it can’t be pumped, then it’s not regular.”

In practice, this contrapositive approach always boils down to a proof by contradiction. To show that an infinite language  $L$  isn’t regular, first assume that it is regular, then show that its long strings can’t always be pumped, which contradicts the pumping lemma. Doing this correctly requires great care in distinguishing between what you can assume and what you must prove. We will first look at two examples and then make some further remarks about this. Our first example repeats the proof that  $\text{GnTn}$  isn’t regular, but this time by applying the pumping lemma instead of the previous direct proof.

**Claim 9.6.**  $\text{GnTn}$  is not a regular language.

*Proof of the claim (using the pumping lemma).* Assume  $\text{GnTn}$  is regular and argue for a contradiction.  $\text{GnTn}$  is infinite, so the pumping lemma applies. Thus there exists some cutoff  $N$  such that all  $S \in \text{GnTn}$  with  $|S| \geq N$  can be pumped before  $N$ . Consider the particular choice of  $S = \text{G}^N \text{T}^N$ . Note that  $S$  is a member of  $\text{GnTn}$ . So by our assumption,  $S$  can be pumped before  $N$ . This means there is a nonempty substring  $R$  in the first  $N$  characters of  $S$ , such that  $S$  can be pumped with  $R$ . We don’t know exactly what this substring  $R$  is, but we do know that it must consist of one or more G characters—let’s suppose it consists of  $k$  G’s, where  $k \geq 1$ . If we pump  $S$  with one extra copy of  $R$ , we get the new string  $S' = \text{G}^{N+k} \text{T}^N$ . But  $S'$  is not a member of  $\text{GnTn}$ , contradicting the fact that  $S$  can be pumped.  $\square$

For a second example, let’s investigate the language  $\text{MoreGthanT}$ , which is the set of ASCII strings containing more G characters than T characters.

**Claim 9.7.** The language  $\text{MoreGthanT}$  is not regular.

*Proof of the claim.* Assume  $\text{MoreGthanT}$  is regular and argue for a contradiction.  $\text{MoreGthanT}$  is infinite, so the pumping lemma applies. Thus there exists some cutoff  $N$  such that all  $S \in \text{MoreGthanT}$  with  $|S| \geq N$  can be pumped before  $N$ . Consider the particular choice of  $S = \text{T}^N \text{G}^{N+1}$ . Note that  $S$  is a member of  $\text{MoreGthanT}$ . So by our assumption,  $S$  can be pumped before  $N$ . This means there is a nonempty substring  $R$  in the first  $N$  characters of  $S$ , such that  $R$  can be used for pumping. We don’t know exactly what this substring  $R$  actually is, but we do know that it must consist of one or more T characters—let’s suppose it consists of  $k$  T’s, where  $k \geq 1$ . If we pump  $S$  with one extra copy of  $R$ , we get the new string  $S' = \text{T}^{N+k} \text{G}^{N+1}$ . But  $S'$  is not a member of  $\text{MoreGthanT}$ , contradicting the fact that  $S$  can be pumped.  $\square$

Notice the formulaic, mechanical feel to these proofs. Producing proofs of this kind is an important skill for theoretical computer scientists. Fortunately, the process can be learned by practicing. The main steps in each proof are as follows:

1. Apply the pumping lemma to the language  $L$ , which produces a fixed (but unknown) cutoff  $N$ . Your proof must work for any sufficiently large  $N$ .
2. Choose a particular string  $S$  that is a member of  $L$  and has length at least  $N$ . This is the only step in the proof that requires genuine creativity. You have

complete freedom to choose  $S$ , and you need to come up with something that will produce a contradiction when pumped before  $N$ . That is, pumping with any substring before  $N$  must produce a new string that is *outside*  $L$ .

3. Consider all valid substrings  $R$  of  $S$ . The pumping lemma says  $S$  can be pumped, but tells us nothing about exactly which substring can be used for pumping. Therefore, your proof must work for every nonempty substring  $R$  drawn from the first  $N$  characters of  $S$ . By coming up with a clever choice of  $S$  in the previous step, you can ensure  $R$  has some kind of very simple structure. In the two examples above, for instance,  $R$  was guaranteed to consist of all G's (first example) or all T's (second example).
4. Observe that pumping with  $R$  some particular number of times results in a string  $S'$  that is outside  $L$ . Your proof only needs to work for one particular number of pumps. Zero pumps or two pumps usually work. In both proofs above, we used two pumps (i.e., one additional copy of  $R$ ). In the first proof, zero pumps would also have worked. In the second proof, two or more pumps would work.

## 9.7 COMBINING REGULAR LANGUAGES

We end this chapter by proving that regular languages are closed under five common language operations. This includes one new operation: we define the *reverse* of a language  $L$ , denoted  $L^R$ , to be formed by reversing each string in  $L$ .

**Claim 9.8.** Let  $L, L_1, L_2$  be regular languages. Then all the following are also regular:  $L^*$ ,  $\overline{L}$ ,  $L^R$ ,  $L_1 \cup L_2$ , and  $L_1 \cap L_2$ .

*Proof of the claim.* To show that  $L^*$  is regular, let  $r$  be a regular expression representing  $L$ . Then the regex  $r^*$  represents  $L^*$ , as required.

To show  $\overline{L}$  is regular, consider a dfa  $M$  that decides  $L$ . Insert explicit transitions to the  $q_{\text{reject}}$  state if necessary. Now swap the  $q_{\text{reject}}$  and  $q_{\text{accept}}$  states of  $M$ . This gives a new dfa deciding  $\overline{L}$ , as required.

To show  $L^R$  is regular, consider a dfa  $M$  that decides  $L$ . Roughly speaking, we want to swap the  $q_0$  and  $q_{\text{accept}}$  states of  $M$ , reverse all the arrows in the state diagram, and thus obtain a new finite automaton that decides  $L^R$ . This construction requires some extra details to make sure the start and end of the string (especially blanks) are handled correctly, but we leave those details as an exercise.

To show  $L_1 \cup L_2$  is regular, let  $r_1$  and  $r_2$  be regular expressions representing  $L_1, L_2$  respectively. Then the regex  $r_1|r_2$  represents  $L_1 \cup L_2$ , as required.

To show  $L_1 \cap L_2$  is regular, note that intersection can be expressed in terms of union and complementation. Specifically, for any sets  $A$  and  $B$ , we have  $A \cap B = \overline{(\overline{A} \cup \overline{B})}$ . So the regularity of  $L_1 \cap L_2$  follows immediately from the previous results on union and complementation.  $\square$

We can combine these results with the pumping lemma to prove that many more languages are not regular. For example, consider the language SameGT, defined as the set of all genetic strings containing the same number of G's and T's. Examples of strings in SameGT include  $\epsilon$ , CCC, TACG, and GTGTGTAA.

**Claim 9.9.** SameGT is not regular.

*Proof of the claim.* Assume SameGT is regular and aim for a contradiction. Consider the language GsTs represented by the regex  $G^*T^*$ . GsTs is clearly regular, since it is represented by a regex. SameGT is regular by our initial assumption. Therefore,  $GsTs \cap \text{SameGT}$  is also regular (using the previous claim). But note that  $GsTs \cap \text{SameGT}$  is precisely the language GnTn, which we have previously proved is not regular. This is a contradiction and completes the proof.  $\square$

As a more sophisticated, interesting, and practical example, consider the question of whether a computer programming language such as Java or Python might be regular. The claim below shows that Java isn't regular. In reading the proof, don't worry if you're not familiar with Java. The key ideas can be adapted to most other modern programming languages, including Python. However, we won't study an explicit proof for Python, because of a rather silly technical reason: Python doesn't use brace characters for code blocks, and the proof below relies directly on the use of such braces.

**Claim 9.10.** The language of Java programs is not regular.

*Proof of the claim.* Let  $J$  be the language of Java programs as defined on page 51. Assume  $J$  is regular and argue for a contradiction. Let  $K$  be the language represented by the regular expression “class C  $\{*\}^*$ ”. That is,  $K$  is the language consisting of the string “class C” followed by any number of open-brace characters, followed by any number of close-brace characters. Set  $L = J \cap K$ . We know  $K$  is regular because it's represented by a regular expression. And  $J$  is regular by our initial assumption. Therefore, because the intersection of regular languages is regular,  $L$  is also regular.

Java requires opening and closing braces to occur in matching pairs, with one exception that we will address later. So in fact  $L$  consists of exactly the strings “class C  $\{"\}^n$ ”, for  $n \geq 1$ . (It would be useful to verify that, for example, “class C  $\{\{\}\}$ ” is a legal Java program. Try compiling it now!)

By the claim above, reversing a regular language preserves regularity, so  $L^R$  is also regular. And  $L^R$  consists of the strings “ $\}^n\{^n$  C ssalc” for  $n \geq 1$ . But we can apply the pumping lemma to  $L^R$ , using exactly the same argument as for the language GnTn (see claim 9.6, page 186), and thus conclude that  $L^R$  is not regular. This contradicts the earlier result that  $L^R$  is regular, so the proof is complete.

Finally, we need to address the fact that there is a sneaky way to have unmatched braces in Java: a brace could occur as part of a string literal, for example in the program “class C {String a=“{”; }”}. The proof can be patched up to address this technicality, but we leave the details as an exercise.  $\square$

## EXERCISES

**9.1** Construct a dfa that accepts ASCII strings containing exactly two or five z characters.

**9.2** Construct an nfa that accepts the language  $\{C^n : n \text{ is a multiple of 3, 5, or 7}\}$ .

### 9.3 Construct a dfa that recognizes the language

$$L = \{abc, abcde\} \cup \{(xy)^n : n \text{ is divisible by } 3\}.$$

**9.4** Let's agree to describe nfas and dfas using ASCII strings in a format similar to that of figure 9.2, page 166. Give a plain-English description of the language accepted by the following dfa:

```

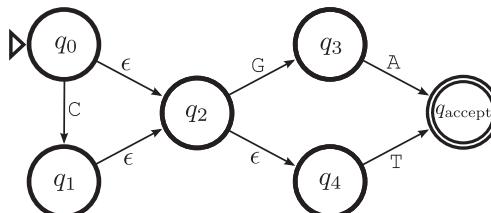
q0->q0: 0369
q0->q1: 147
q0->q2: 258
q0->qA: _
q1->q0: 258
q1->q1: 369
q1->q2: 147
q2->q0: 147
q2->q1: 258
q2->q2: 369

```

**9.5** Implement a dfa simulator in Python. Either write your program from scratch or use the provided Turing machine simulator `simulateTMv2.py` as a starting point. Your program should take as input two parameters  $P, I$ , where  $P$  is the ASCII description of a dfa (formatted as in the previous question), and  $I$  is an ASCII input string. The output is “yes” if the dfa accepts  $I$ , and “no” if it rejects  $I$ . (Note: The book materials already provide dfa and nfa simulators, called `simulateDfa.py` and `simulateNfa.py` respectively. But these simulators are rather trivial because they rely on the provided `Dfa` and `Nfa` classes. In contrast, this question asks you to write a stand-alone dfa simulator in the same spirit as the stand-alone Turing machine simulator `simulateTMv2.py`.)

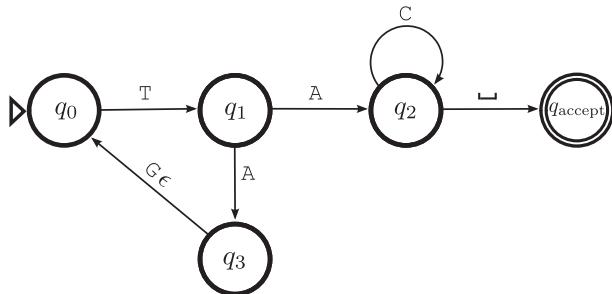
**9.6** Construct two distinct, but equivalent, nfas that accept the language of genetic strings containing GAGA or GTGT.

**9.7** Let  $M$  be the following nfa:

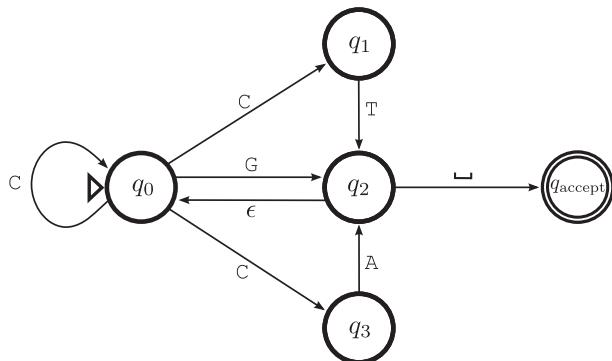


- (a) Convert  $M$  into a strict nfa.
- (b) Using your experience from part (a) as a guide, describe an algorithm for converting any nfa into a strict nfa.

**9.8** Use the algorithm described in section 9.3 to convert the following nfa into a dfa. You must label your dfa states using the convention of figure 9.8 (page 175).



**9.9** Repeat the previous exercise with the following nfa:

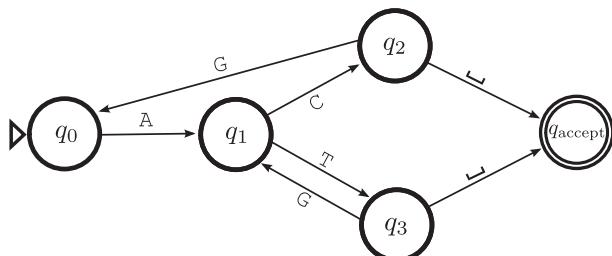


**9.10** Take  $\Sigma = \{C, A, G, T\}$ . Prove that the language of all strings on  $\Sigma$  containing between 3 and 5 G's (inclusive) is regular.

**9.11** Again take  $\Sigma = \{C, A, G, T\}$ . Find a regular expression for the language of all strings on  $\Sigma$  containing an even number of occurrences of the substring “TCA”, each separated by zero or more G characters. For example, the following strings are in this language:  $\epsilon$ , “TCATCA”, “GGGTCAGGTCAGGGG”, “TCAGGGTCAGGTCATCAGGGG”.

**9.12** Find an nfa equivalent to the regex  $(T(GGA)^*|C)^*$ .

**9.13** Convert the following dfa into an equivalent regex:



**9.14** Read the documentation for the standard regular expression library in Python. (The library is called `re`.) Use this library to implement one or more useful Python programs. For example, produce programs that decide whether the ASCII input (i) is alphanumeric, (ii) is a genetic string, (iii) contains the substring GAGA and the substring CTTCC, in that order, (iv) matches the

regular expression  $(AA(CG)^*(T^*))^*|GGT$ , (v) contains the definition of a Python function.

**9.15** Use the pumping lemma to prove that the following language is not regular:  $\{C^nTTC^m : m > n + 5\}$ .

**9.16** Use the pumping lemma to prove that the following language is not regular: the set of all ASCII strings that mention US states more often than countries in the EU. For example, “Alabama Georgia Alabama Spain” is in the language (3 US states, 1 EU country), but “SpainAlabama5GeorgiaAlabama23452cc Spain Portugal ItalyPennsylvania” is not in the language (4 US states, 4 EU countries). (Hint: You’ll probably need to use the fact that the intersection of two regular languages is also regular.)

**9.17** Take  $\Sigma = \{C, A, G, T\}$ . For each of the following languages on  $\Sigma$ , state whether the language is regular and give a rigorous proof of your claim:

- (a)  $\{C^m A^n C^p : m, n, p \geq 2\}$
- (b)  $\{C^m A^n C^m : m, n \geq 2\}$
- (c)  $\{C^m A^n TA^n : n \geq 0 : m > n\}$
- (d)  $\{CCA^n T^{2m} : n > m\}$
- (e)  $\{C^p A^q T^r : 1 \leq p \leq 20 \text{ and } q < r\}$
- (f)  $\{(A|C)^* : \text{number of A's and C's is equal}\}$
- (g) strings that do not contain “GAGA”
- (h) strings that do not contain “NANA” (as usual, N represents any one of C, A, G, T)

**9.18** This exercise asks you to complete some of the details omitted in the proof that the reverse of a regular language is also regular (page 187). Let  $M$  be a dfa deciding a language  $L$ . Give a detailed description of how to construct  $N$ , an nfa that decides  $L^R$ .

**9.19** Complete our proof that the set of Java programs is not a regular language (claim 9.10, page 188), giving a rigorous description of how to account for unmatched braces in string literals.

**9.20** Let a *read-only Turing machine* be the same as a Turing machine except that it cannot change the tape symbols. Prove that, given a read-only Turing machine  $R$ , there exists a dfa  $M$  equivalent to  $R$ .



Part II

COMPUTATIONAL COMPLEXITY THEORY





# 10



## COMPLEXITY THEORY: WHEN EFFICIENCY DOES MATTER

We should explain, before proceeding, that it is not our object to consider this problem with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the *nature* and *number* of the *operations* required to be performed during its complete solution.

—Ada Lovelace, from Note F in her 1843 translation  
of *Sketch of the Analytical Engine*

So far in this book we have concentrated on the theory of *computability*. By definition, computability theory ignores the amount of time or storage required to run a program and instead concentrates on whether the answer to a given problem can be computed at all. In contrast, the field of *computational complexity theory* asks whether solutions can be computed *efficiently*. So, we must of course restrict our attention to problems that are computable, and then ask, how much time and storage will be required to produce the answer?

By the way, we'll use the term “complexity theory” as a shorthand for “computational complexity theory.” But watch out: there are several other unrelated branches of science that are also called “complexity theory,” so web searches for this phrase may turn up unexpected results.

As in the previous chapters, we try to take a ruthlessly practical approach, so we'll mostly think in terms of Python programs. But it's important to remember that the definitions and results will hold for other programming languages and for Turing machines too. As with computability theory, if your goal is complete mathematical rigor, then it's best to interpret everything in terms of Turing machines.

### 10.1 COMPLEXITY THEORY USES ASYMPTOTIC RUNNING TIMES

With those caveats, let's plunge into some complexity theory. Suppose a friend shows you a Python program  $P$ , and then asks how long it will take to run  $P$ .

You run the program on some typical input and it takes about 5 seconds, so you tell your friend “about 5 seconds.” Is this a useful answer? Is it what your friend wanted to know? Maybe, or maybe not. If your friend will typically be running the program on similar hardware with similar inputs, it’s quite possible that “5 seconds” is a useful and essentially complete answer. On the other hand, there are at least three reasons this answer could be incomplete:

1. **Dependence on hardware.** Different computers run at different speeds. And specific hardware features—such as the number of cores, specialized floating-point units, and high-end graphics cards—can dramatically alter the running time of a program on different machines, even if the basic specs of the CPUs on those machines are similar. Therefore, we may need to augment our “5 seconds” answer with a description of the machine used for the experiment.
2. **Dependence on easy or hard input.** You probably already noticed an issue that was hidden under the rug earlier: the “5 seconds” answer was produced by running  $P$  on “typical” input. But for some problems, there may not be any way of defining typical inputs. Even if we restrict consideration to inputs of a fixed size (say, up to 1 kilobyte), there may be some inputs that are easy and some that are hard. For example, consider the task of finding the shortest route between two points on a city map. The optimal route can be found with less effort on a grid system (such as New York) than on an irregular system (such as Paris) of similar size. A simple way around this easy-versus-hard dependence is to instead consider *worst-case* performance, in which we state the longest possible running time for a given input size. For example, after enough experimentation and/or theoretical analysis of  $P$ , it might be possible to say “ $P$  takes at most 7 seconds, for any input file up to 1 kilobyte in length.” Another possibility is to examine *average-case* performance. Complexity theorists have studied both the worst-case and average-case measures intensively. But worst-case analysis turns out to be a more useful starting point, so we won’t pursue average-case analysis any further in this book.
3. **Dependence on length of the input.** Another obvious problem is that many programs take longer to run when you increase the length of the input. For example, a face detection algorithm might take longer on a 10-megapixel image than on a 1-megapixel image. We could describe some of this dependence by giving multiple data points (e.g., “up to 7 seconds for 1 kB inputs, and 2.5 hours for 1 MB inputs”). In many situations, this is a practical and useful approach. But any finite list of running times leaves the lingering question of what happens if we increase the input size even more. So a more complete answer would also describe the *shape* of the running-time curve (when plotted against input size) for *large* inputs. A good way to describe the running time for large inputs is *big-O notation*, which is described in more detail in the next section. For now, two brief examples of big-O notation will be enough for us to continue our discussion:  $O(n)$  means the running-time curve is at most linear (i.e., a straight line);  $O(n^2)$  means the curve is at most quadratic (i.e., a parabola). So, when answering the question of how long it will take to run  $P$ , we should include

a statement such as “ $P$ ’s running time is in  $O(n^2)$ , where  $n$  is the size of the input file.”

Hence, a comprehensive answer to the question “How long will it take to run  $P$ ?” would be something like, “On a desktop PC with an Intel Core i5-750 CPU,  $P$  takes up to 7 seconds for 1 kB inputs, and 2.5 hours for 1 MB inputs; and theoretical analysis shows that  $P$ ’s running time is in  $O(n^2)$ , where  $n$  is the size of the input file.” This may seem rather verbose, but it is in fact quite realistic. When computer scientists write papers about their algorithms—whether for publication in an academic journal, or as an internal document at a high-tech company—the performance of the algorithms is usually described in terminology very similar to this example.

This discussion demonstrates that at least two metrics can be used to measure the running time of an algorithm. The first metric is *absolute running time*, which depends on details such as the hardware, implementation, and input. The second metric is *asymptotic running time* (usually described in big-O notation), which is independent of the hardware and implementation details, instead describing how the running time scales for large inputs. In any practical application, both metrics are important: before choosing a method for solving a problem, both the absolute and asymptotic running times of that method should be investigated and understood.

However, in this book, we are interested in making *general* statements about program running times—statements that don’t depend on the hardware used, implementation details, or nature of the input. Therefore, in the remainder of the book, we assess problems and their methods of solution using only the asymptotic running-time metric. For this, we need big-O notation, which is described in the next section. We also need a formal definition of “running time”; this is addressed in section 10.3.

## 10.2 BIG-O NOTATION

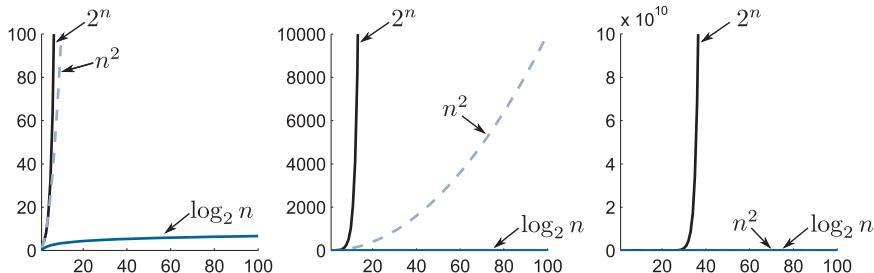
Running times are measured in terms of the length,  $n$ , of the input string. *For the rest of the book,  $n$  will always denote the length of an input string, unless stated otherwise.* Most running times can be measured using some combination of three kinds of *basic functions*:

- logarithmic:  $\log_b n$ , for some base  $b > 1$
- polynomial:  $n^k$ , for some exponent  $k \geq 1$
- exponential:  $b^n$ , for some base  $b > 1$

Possibly, we should also include the factorial function  $n!$  in our list of basic functions. Recall that  $n!$  is defined as

$$n! = 1 \times 2 \times 3 \times \cdots \times n.$$

For our purposes, factorial functions behave very much like exponential functions, so it’s easiest to think of  $n!$  as being in the “exponential” class of basic functions above.



**Figure 10.1: Logarithmic, polynomial, and exponential functions have vastly different growth rates.** The three graphs above each show the same set of three functions:  $\log_2 n$ ,  $n^2$ , and  $2^n$ . In each case, the horizontal axis has the same range, 0–100. But the vertical axis is compressed by a factor of 100 in the middle graph and a factor of 1 billion in the right-hand graph.

There is some potential for confusion with the “polynomial” class of basic functions too. A single term like  $n^5$  is a *basic polynomial*. But by itself, the word “polynomial” usually refers to a linear combination of basic polynomials, such as  $3n^5 + 2n^3 + 9$ .

The three basic function types above can be combined using addition, multiplication, and composition. So we end up with running times like  $7n^3 + n \log_2 n$ ,  $3^{n^2+5n}$ , and  $6n! + n^2(\log_2 n)^4$ . In expressions like these, the components joined by “+” signs are called *terms*. (So in these examples,  $7n^3$ ,  $n \log_2 n$ , and  $n^2(\log_2 n)^4$  are all terms.)

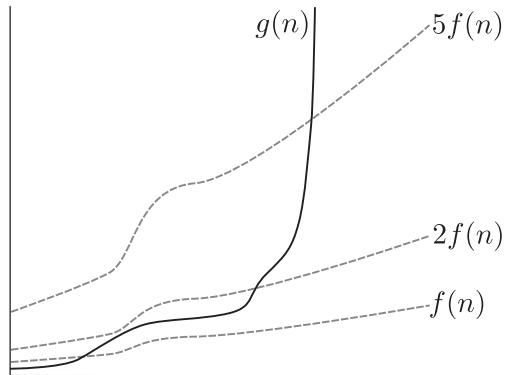
We would like to be able to compare running times to determine which is the best. But recall from the previous section that the “best” running time should be measured *asymptotically*. Specifically, this means we (i) ignore multiplicative constants, and (ii) compare only for sufficiently large values of  $n$ . This motivates the following definition.

**Formal definition of big-O notation.** Let  $f$  and  $g$  be functions mapping nonnegative integers to nonnegative integers. We write  $f \in O(g)$  if there is a positive constant  $C$  such that

$$f(n) \leq C g(n)$$

for all sufficiently large  $n$ . “Sufficiently large” means that there exists  $N$  such that  $f(n) \leq C g(n)$  for all  $n \geq N$ . The notation “ $f \in O(g)$ ” is read as “ $f$  is in big-O of  $g$ .”

This formal definition is a little hard to work with, unless you are familiar with college-level calculus. The formal definition was presented for completeness but in practice we won’t need to use it. Instead, we can use an informal prescription for understanding big-O notation. The intuition behind our informal approach relies on the well-known ideas that log functions grow slowly, polynomials grow quickly, and exponentials grow ridiculously quickly—see figure 10.1 for a graphical reminder of this. So logarithmic running times are better than polynomial running times, which are better than exponential running times. But



**Figure 10.2:** Example of a function  $g(n)$  dominating  $f(n)$ . A function  $g$  dominates function  $f$  if  $g$  is eventually larger than any multiple of  $f$ , such as the  $2f$  and  $5f$  examples shown here.

it's not always so simple. The basic functions can be combined in various ways, so our informal approach is based on selecting the *dominant term* of a function.

### Dominant terms of functions

We will build up our understanding of dominant terms in stages. First, we say that  $g(n)$  *dominates*  $f(n)$  if  $g(n) > cf(n)$  for all sufficiently large  $n$ , for all positive constants  $c$ . That is, if we plot both functions, then eventually the graph of  $g$  goes above the graph of  $f$  and stays there forever, even when we scale  $f$  up by a large constant. See figure 10.2 for a visual representation of this. Note that domination is transitive: if  $h$  dominates  $g$ , and  $g$  dominates  $f$ , then  $h$  dominates  $f$ .

Next, we define *basic terms* as the terms in the following ordered list:

$$\begin{aligned}
 & 1, \log n, (\log n)^2, (\log n)^3, \dots, \\
 & n, n^2, n^3, \dots, \\
 & 2^n, 3^n, 4^n, \dots, \\
 & n!, \\
 & 2^{n^2}, 3^{n^2}, 4^{n^2}, \dots, \\
 & 2^{2^n}, \\
 & \dots
 \end{aligned} \tag{*}$$

Don't be disturbed by the fact that the base  $b$  of the logarithms has disappeared in the first line above. The base of a logarithm changes it only by a constant factor, and constant factors are ignored by dominant terms. That's why we don't even bother to specify the base of the logarithm in many of the expressions in this book.

Notice the structure of the list (\*) of basic terms. The first three lines represent the three classes of basic functions: logarithmic, polynomial, and exponential.

After that, we have three more lines representing factorials ( $n!$ ), exponentials of polynomials (e.g.,  $2^{n^2}$ ), and finally a *double exponential* ( $2^{2^n}$ ) in which the exponent is itself an exponential.

The key idea, which we state here without proof, is that each basic term in  $(\star)$  dominates any term earlier in the list. In fact, each term grows so much faster than its predecessors that sums and products of earlier terms are dominated too. Multiplicative constants can always be ignored. These ideas are best illustrated by some examples:

Terms	Reason
$(\log n)^3$ dominates $\log n$	Predecessor in $(\star)$
$n$ dominates $544(\log n)^{43}$	Predecessor (ignore constant)
$n^5$ dominates $96n^3$	Predecessor (ignore constant)
$n!$ dominates $5^n + n^3$	Sum of predecessors
$n^2$ dominates $n \log n$	Product of predecessors
$3^n$ dominates $2^n n^8$	Product of predecessors
$3^n$ dominates $2^n n^8 + n \log n$	Sum and product of predecessors
$n^2 3^n$ dominates $n^3 2^n$	The most dominant factor wins
$n^5 3^n$ dominates $n^4 3^n$	Break ties via next-largest factor

The final line above demonstrates that, if two expressions have the same basic term as their most dominant factor, we break the tie using the next-largest factor. Also, there is some fine print to the “product of predecessors” rule: when multiplying entries from the same row, products should be simplified before applying the rule. For example,  $n^5$  does not dominate  $n^2 n^4 = n^6$ .

Applying these ideas allows us to extract the dominant term from most running-time functions that arise in practice. The following definition summarizes this technique.

**Definition of a dominant term.** Let  $f$  be a function that is a sum of terms, where each term is a product of one or more of the basic terms in  $(\star)$ . The *dominant term* of  $f$  is the term that dominates all others, with any multiplicative constant ignored.

Let’s write  $\text{DT}(f)$  for the dominant term of  $f$ . Here are some examples to illustrate the definition:

$f(n)$	$\text{DT}(f)$
$8n!$	$n!$
$3n^2 + 5n \log n + 8$	$n^2$
$7n^3 + 5n^2 3^n + 6n^5 2^n + 8n 3^n$	$n^2 3^n$

Multiplication and logarithms of functions affect dominant terms according to the following three rules, where  $f, g$  are functions and  $c, k$  are constants:

$$\begin{aligned} \text{DT}(fg) &\rightarrow \text{DT}(f)\text{DT}(g), \\ \text{DT}(\log f) &\rightarrow \log(\text{DT}(f)), \\ \text{DT}(\log(cf^k)) &\rightarrow \text{DT}(\log f). \end{aligned}$$

If you know some calculus, it's a worthwhile exercise to prove the first two rules above. But in this book, we will use them without proof. The third line above follows directly from the fact that  $\log(cx^k) = \log(c) + k\log(x)$ . Also, remember that the base of a logarithm changes it only by a constant factor, so we can leave the base unspecified. Here are some examples of dominant terms involving multiplication and logarithms of functions:

$f(n)$	$\text{DT}(f)$	Reason
$(2^n + n^4)(\log n + n^3)$	$2^n n^3$	multiply dominant terms
$\log(n^3 + 6n^2)$	$\log n$	move DT inside, then $\log n^3 = 3 \log n$
$\log(2^n + 3n^4)$	$n$	move DT inside, then $\log 2^n = \text{const} \times n$

The final line here uses the fact that logs and exponentials are inverses of each other, and we ignore the constant produced by the unspecified base of the logarithm.

## A practical definition of big-O notation

With a good understanding of dominant terms established, we are ready for a practical definition of big-O notation. (Contrast this with the formal definition on page 198.)

**Practical definition of big-O notation.** Let  $f$  and  $g$  be functions that are sums of the terms defined above. We write  $f \in O(g)$  if either (i)  $f$  and  $g$  have the same dominant term or (ii)  $\text{DT}(g)$  dominates  $\text{DT}(f)$ .

The key point is that  $f$  and  $g$  need not have the same dominant term—either the dominant terms are the same, or  $g$ 's term dominates. In words, we could say “ $f$  is no larger than  $g$ , for sufficiently large  $n$ , and ignoring multiplicative constants.” Some examples will illustrate this:

$f(n)$	$\text{DT}(f)$	$g(n)$	$\text{DT}(g)$	$f \in O(g)?$
$3n + 5n \log n$	$n \log n$	$3n(\log n)^2$	$n(\log n)^2$	yes
$n^2 + 5n^4$	$n^4$	$2^n + \log n$	$2^n$	yes
$5n^3 + 2n$	$n^3$	$2n^3 + \log n$	$n^3$	yes
$5n^3 \log n + 2n$	$n^3 \log n$	$2n^3 + \log n$	$n^3$	no

The “ $O$ ” in big-O stands for order, which is another word for the magnitude of a function. So the intended meaning of  $f \in O(g)$  is something like “ $f$  is in the class of functions of order  $g$ .” But this terminology can be misleading, since it's too easy to think that  $f$  and  $g$  must have the same order, whereas in fact  $f$ 's order can be smaller than  $g$ 's. Therefore, it seems preferable to read “ $O$ ” as “big-O,” as in “ $f$  is in big-O of  $g$ .” This also helps in eliminating confusion with little- $o$ , which means something different (see 202).

Note that many authors write  $f = O(g)$  instead of  $f \in O(g)$ . The use of “=” instead of “ $\in$ ” is very common, but it is also confusing since it contributes to the mistaken impression that the order of  $f$  “equals” the order of  $g$ . If it helps, think of  $O(g)$  as a *set* of functions, so that “ $f \in O(g)$ ” literally means that  $f$  is an

element of the set  $O(g)$ . Big-O can be defined rigorously this way, although our own definition did not take that approach.

Next we examine two common mistakes involving big-O notation. The first one has been mentioned twice already, but it's worth emphasizing again:

**Common big-O mistake #1: Forgetting the “at most.”** Always remember that  $f \in O(g)$  means that  $f$  is *at most*  $g$  (for large  $n$ , ignoring constants). For example, the fact that  $f \in O(2^n)$  does not mean that  $f$  is an exponential function;  $f$  could in fact be a logarithmic or polynomial function, since these are dominated by  $2^n$ . To make this example even more explicit, note that  $n^3 \in O(2^n)$  and  $\log n \in O(n)$ .

**Common big-O mistake #2: Constants or extra terms inside the  $O(\cdot)$ .** Technically, it isn't wrong to write something like  $O(7n^2)$  or  $O(n^3 + \log n)$ . But it's always misleading to use multiplicative constants or extra terms inside the parentheses of a big-O expression. The reason is that these things are completely superfluous, so the reader will be confused by their presence and quite possibly assume that the writer doesn't understand big-O notation. For example, suppose you write “the running time of the algorithm is in  $O(7n^2)$ .” This may be technically correct, but the 7 is irrelevant, because the running time is also in  $O(n^2)$ , and in  $O(43255n^2)$ . Therefore, you should always use just a single term without a multiplicative constant inside a big-O expression.

## Superpolynomial and subexponential

A function is *superpolynomial* if it is not in  $O(n^k)$  for any  $k \geq 1$ . That is, it grows faster than any polynomial. Exponential functions like  $2^n$  and  $3^{n^2}$  are superpolynomial, but we will soon see more subtle examples.

A function is *subexponential* if it is in  $O(2^{n^\epsilon})$  for all  $\epsilon > 0$ . That is, it grows more slowly than any exponential function. Obvious examples include logarithmic and polynomial functions like  $\log n^3$  and  $6n^2$ . (Warning: Some authors use a different definition of subexponential.)

There are functions that sneak in between polynomials and exponentials—functions that are both superpolynomial and subexponential. This includes the class of *quasipolynomial* functions, which have the form  $2^{(\log n)^k}$ , or more generally  $2^{p(\log n)}$  for some polynomial  $p$ .

## Other asymptotic notation

A property is *asymptotic* if it depends only on sufficiently large values of  $n$ . For this reason, big-O is described as *asymptotic notation*. Four other types of asymptotic notation are used in computer science: little-o ( $o$ ), big-Omega ( $\Omega$ ), big-Theta ( $\Theta$ ), and soft-O ( $\tilde{O}$ ). None of this notation will be used in this book (except that soft-O is briefly mentioned in some exercises). But knowledge of all four types will help you understand material from other sources, so we'll now define them using dominant terms. The following table gives these definitions, assuming that “ $<$ ”

means “is dominated by”; “ $\leq$ ” means “is equal to, or is dominated by”; and “ $\geq$ ” means “is equal to, or dominates.”

- $f \in O(g)$  means  $DT(f) \leq DT(g)$
- $f \in o(g)$  means  $DT(f) < DT(g)$
- $f \in \Theta(g)$  means  $DT(f) = DT(g)$
- $f \in \Omega(g)$  means  $DT(f) \geq DT(g)$
- $f \in \tilde{O}(g)$  means  $DT(f) \leq DT(g)p(\log n)$  for some polynomial  $p$

The idea behind soft-O is to ignore *polylogarithmic* factors (i.e., factors of the form  $p(\log n)$ , for some polynomial  $p$ ) since they are often irrelevant in practice. For example, if the running time of an algorithm is in  $O(n^3(\log n)^5)$ , we can say it is in  $\tilde{O}(n^3)$ .

## Composition of polynomials is polynomial

We already know how to compute dominant terms (and hence big-O) resulting from addition and multiplication of functions (see page 200). But what about the *composition* of functions? That is, what is  $O(f(g(n)))$ ? In general, this can be messy. But there is an important special case when  $f$  and  $g$  are polynomials (or at least, bounded by polynomials): the composition of polynomial functions is always polynomial.

**Claim 10.1.** Suppose  $f(n) \in O(n^j)$  and  $g(n) \in O(n^k)$ . Then  $f(g(n)) \in O(n^{jk})$ .

The proof of this claim requires calculus and is omitted. But the claim is intuitively reasonable, since the elementary properties of exponents tell us that  $(n^k)^j = n^{jk}$ .

## Counting things with big-O

The main use of big-O in this book is to measure the running time of programs. But to do that, often we need to count the number of objects that the program will process. This is an area that mathematicians call *combinatorics*. Of course, we don’t need exact answers to our combinatoric problems: instead, we can use big-O.

As an initial example, consider the following problem: How many edges could occur in a graph with  $k$  nodes? Well, each edge goes from one of the  $k$  nodes to another of the  $k$  nodes, which gives a maximum of  $k \times k = k^2$  possibilities. So, there are  $O(k^2)$  edges in a graph. Note that we did *not* need a sophisticated analysis to obtain this answer. In fact, we probably overcounted: if the graph is undirected and has no self-edges (i.e., from a node to itself), it turns out there are only  $k(k - 1)/2$  possible edges. But this quantity is still in  $O(k^2)$ , so our asymptotic estimate is perfectly reasonable. There is an important lesson here: Usually, you don’t need fancy combinatorics to estimate program running times. Instead, use simple upper bounds to obtain an asymptotic estimate using big-O notation.

Quantity	Value
number of subsets of a set with $k$ elements	$2^k$
number of subsets of size exactly $s$ , of a set with $k$ elements	$O(k^s)$
number of subsets of size at most $s$ , of a set with $k$ elements	$O(k^s)$
number of edges in a graph with $k$ nodes	$O(k^2)$
number of Hamilton cycles in a graph with $k$ nodes	$O(k!)$
number of Hamilton paths with a given source and destination in a graph with $k$ nodes	$O(k!)$

**Figure 10.3:** Asymptotic estimates for combinatorial problems that will be used later in the book.

Figure 10.3 gives results for some of the common combinatorial problems that we will need later. Exercise 10.4 (page 224) asks you to prove and explain some of these results. Note especially the first result in the table: the fact that a set with  $k$  elements has  $2^k$  subsets is an essential result that you should memorize, because it is needed so often. Also note that the concepts of *Hamilton cycle* and *Hamilton path* are defined on page 47.

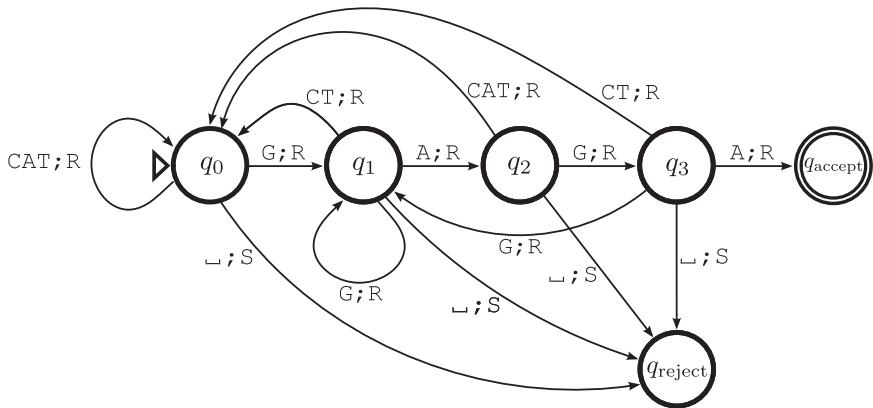
## 10.3 THE RUNNING TIME OF A PROGRAM

Recall that we use two different models of computer programs in this book: Python programs and Turing machines. Python programs are useful in practice, whereas Turing machines are useful for theoretical analysis. In chapter 5, we saw that the two models are equivalent in terms of what they can compute. But the models can be very different in terms of running times. Therefore, we will initially give separate definitions of “running time” for Python programs and Turing machines. Later (see page 219), we will see how to unify the separate definitions, for the particular case of random-access Turing machines.

### Running time of a Turing machine

Let  $M$  be a standard Turing machine, or a multi-tape Turing machine, or a random-access Turing machine. Recall that  $M$  has a transition function, mapping the scanned symbol and current state to a newly written symbol, a new state, and a new head position. While the machine is performing a computation, each application of the transition function is called a *step* or a *transition*.

The *running time* of  $M$  on input  $I$  is the number of steps from the start of  $M$ 's computation with input  $I$ , until  $M$  enters a halting state. Figure 10.4 provides a simple example: a Turing machine for the familiar decision problem `CONTAINSGAGA`. (This particular version of the `containsGAGA` machine is not quite the same as the ones presented earlier—it's been altered to make its running time very easy to analyze.) When we think of a Turing machine as a state diagram, as in figure 10.4, the concept of a “step” corresponds precisely with following one



**Figure 10.4:** State diagram for a `containsGAGA` Turing machine.

of the arrows. By tracing out the number of arrows followed, you should be able to check the following running times for this `containsGAGA` machine:

Input $I$	Running time of <code>containsGAGA</code> on $I$
GAGA	4
GAGATTTTTT	4
GAGCTTTTTT	12
CCGAGA	6
C	2
$\epsilon$	1

Obviously, the running time of `containsGAGA` varies, and depends not only on the length of  $I$  but also the detailed content of  $I$ . For example, the 1000-character string “GAGAAAAA...” requires only 4 steps, but the 1000-character string “TTTTT...” requires 1001 steps. However, it’s easy to see that an input of length  $n$  requires at most  $n+1$  steps on this machine—this is the *worst-case* running time, as already discussed on page 196. This result motivates the following general definition:

**Definition of the running time of a Turing machine.** Let  $M$  be any type of Turing machine. The *running time* of  $M$  is a function  $R_M(n)$ , defined on nonnegative integers. The value of  $R_M(n)$  is the maximum number of computational steps used by  $M$ , taken over all inputs of length at most  $n$ .

The running time of  $M$  is also called the *time complexity* of  $M$ , or simply the *complexity* of  $M$ .

Two more examples will flesh out this idea. First consider the `binaryIncrementer` machine (top panel of figure 5.10, page 83). It’s easy to see that this machine always sweeps over the input once from left to right (requiring  $n-1$  steps), then sweeps back once from right to left (requiring another  $n-1$  steps), for total running time of  $2n-2$ .

Next, let's analyze `moreCsThanGs` (figures 5.7–5.9, pages 80–82). The analysis here is much more complicated. After a while, you can convince yourself that the worst-case input involves an equal number of C's and G's, as in the example of figure 5.7. You might even be able to calculate the exact number of steps in such a computation. But that would be tedious and pointless. Instead, this is a case where *asymptotic* (i.e., big-O) analysis would clearly be preferable, since we will ultimately be interested in the shape of the running-time curve and not an exact formula for it. Based on the pattern in figure 5.7, we see that the head of the machine will sweep up and/or down the input  $O(n)$  times. Each of these sweeps involves at least  $n/2$  steps, and at most  $n$  steps. Thus, each sweep requires  $O(n)$  steps, and there are  $O(n)$  sweeps. Multiplying these together (see page 200), we get a running time in  $O(n^2)$  for this machine.

Here's a summary of our running-time results:

Machine $M$	Running time $R_M(n)$	Asymptotic running time
<code>containsGAGA</code>	$n + 1$	$O(n)$
<code>binaryIncrementer</code>	$2n - 2$	$O(n)$
<code>moreCsThanGs</code>	???	$O(n^2)$

Take careful note of the fact that we computed the asymptotic running time of `moreCsThanGs` without ever computing its exact running time. This is the approach that we will take for all running-time examples in the rest of the book. Therefore, the phrase “running time” will be used interchangeably with “asymptotic running time.”

## Running time of a Python program

Recall from page 23 that our definition of “Python program” was given with respect to some fixed reference computer system  $C$ . Computer system  $C$  is a real computer (e.g., you can take  $C$  to be your own laptop or desktop), except that it is idealized in certain ways to permit programs to use as much storage as necessary.

We know from the discussion of simulating real computers with Turing machines (page 92) that  $C$  uses a fixed instruction set, and that the execution of any Python program will result in the execution of some sequence of CPU instructions. This motivates the following definition: the *running time of a Python program  $P$  on input  $I$*  is the number of CPU instructions executed by the Python process when  $P$  is executed with input  $I$ .

Thus, the fundamental indivisible unit of computation for Python programs is the CPU instruction. We will refer to each executed instruction as a *step* in the Python program. (Recall that for Turing machines, a step is a single application of the transition function.) Once we agree on an appropriate definition of “step” for Python programs and Turing machines, the general definition of “running time” is identical. We restate the definition here for completeness, emphasizing that it applies to Python programs and Turing machines:

**Definition of the running time of a Python program or a Turing machine.** Let  $P$  be a Python program or a Turing machine. The *running time* of  $P$  is a function  $R_P(n)$ , defined on nonnegative integers. The value

```

1 import utils; from utils import rf
2 def containsGAGA(inString):
3     if 'GAGA' in inString:
4         return 'yes'
5     else:
6         return 'no'

```

**Figure 10.5:** Python program `containsGAGA.py`.

of  $R_P(n)$  is the maximum number of computational steps used by  $P$ , taken over all inputs of length at most  $n$ .

The running time of  $P$  is also called the *time complexity* of  $P$ , or simply the *complexity* of  $P$ .

Some examples will make this clear. We again use the `CONTAINS GAGA` problem for illustration, this time analyzing the Python implementation `containsGAGA.py`, reprinted in figure 10.5. It's immediately obvious that, even for a given input such as  $I = \text{"GAGA"}$ , measuring the exact number of CPU instructions executed will be difficult. Instead, we use asymptotic analysis. But even with the extra freedom that asymptotic analysis provides, it seems hard to make progress. What asymptotic cost should be assigned to, say, the `in` operator at line 3? It might seem hopeless to give a mathematically precise answer to this question. We will address this issue in detail shortly (see page 209), but it's preferable to see some concrete examples first. Therefore, let's proceed using a combination of common sense, the Python documentation, and our experience as programmers. Figure 10.6 gives the asymptotic cost for a few Python idioms that are frequently used in this book. Let's use this common-sense approach to analyze `containsGAGA.py` in figure 10.5. Line 3 is in  $O(n)$ , and every other line is in  $O(1)$ . So the aggregate running time is in  $O(n)$ .

We now move on to a more sophisticated example. Consider the computational problem `MATCHINGCHARINDICES`, described in figure 10.7. `MATCHINGCHARINDICES` takes two words as input, and a solution lists the indices of all pairs of identical characters that appear in the two input words. A Python program solving the problem is given in figure 10.8. Line 5 creates an empty list, and line 11 appends the relevant pairs of indices to this list. At line 13, the entries in the list are concatenated, separated by space characters. The resulting string is returned as the solution.

Now let's apply our commonsense approach to find the running time of `matchingCharIndices.py`:

**Claim 10.2.** The program `matchingCharIndices.py` in figure 10.8 has running time in  $O(n^2 \log n)$ .

*Proof of the claim.* The `split()` method at line 3 is in  $O(n)$ , according to figure 10.6.

Moving down to line 9, it's clear this will be executed  $O(n^2)$  times, since that line is inside a doubly nested loop, with each loop having  $O(n)$  executions. Line 10 presents the trickiest part of the analysis. We are constructing a string

- The `in` operator requires  $O(m)$  time, where  $m$  is the length of the string being searched (or more generally, the size in bytes of the data structure being searched). That is, `s in t` requires  $O(\text{len}(t))$ . Note that this estimate applies to searching strings, lists, and tuples. Some other Python data structures can be searched more efficiently.
- Appending to a list requires constant time.
- Constructing a string `s` is usually in  $O(\text{len}(s))$ , including the following specific examples:
  - `s = str(i)`, for an integer  $i$
  - `s = s1 + s2`, for strings  $s_1, s_2$
  - `s = t.join(L)` for string  $t$  and list of strings  $L$
- `s.split()` is in  $O(\text{len}(s))$ .

**Figure 10.6:** Specific examples of running-time estimates for some Python code fragments.

### PROBLEM MATCHINGCHARINDICES

- **Input:** A single string consisting of two alphanumeric ASCII words  $w_1, w_2$  separated by a space. For example, the input could be “hello world”.
- **Solution:** A list of all pairs of indices in the input words that match. More precisely, writing  $w_m[i]$  for the  $i$ th character of word  $m$ , the solution is a list of pairs  $i, j$  such that  $w_1[i] = w_2[j]$ . Pairs are separated by space characters, and the indices in a pair are written in decimal notation separated by a comma. For example, a solution for the input “hello world” is “2, 3 3, 3 4, 1”.

**Figure 10.7:** Description of the computational problem MATCHINGCHARINDICES.

`thisPair`; by figure 10.6, this will cost  $O(\text{len}(\text{thisPair}))$ . So, how long could `thisPair` be? Well, the number of digits in  $i$  and  $j$  is at most  $\log_{10} n$ , so the length of `thisPair`—and hence the total cost of line 10—is in  $O(\log n)$ , for each of its  $O(n^2)$  executions. Line 11 takes only constant time (because appending to a list is a constant time operation) for each of its  $O(n^2)$  executions. In aggregate then, the doubly nested loop costs  $O(n^2 \log n)$ .

For line 13, recall from figure 10.6 that `join()` is in  $O(m)$ , where  $m$  is the final number of characters in the output string. But this output length  $m$  is in  $O(n^2 \log n)$ —this is the worst case in which every character in  $w_1$  is the same as every character in  $w_2$ , so we get  $O(n^2)$  pairs of indices each of length  $O(\log n)$ .

Summarizing this, we have that lines 3–5 are in  $O(n)$ , lines 7–11 are in  $O(n^2 \log n)$ , and line 13 is in  $O(n^2 \log n)$ . So the program as a whole is in  $O(n + n^2 \log n + n^2 \log n)$ , which simplifies to  $O(n^2 \log n)$ .  $\square$

```

1  def matchingCharIndices(inString):
2      # split the input into a list of 2 words
3      (word1, word2) = inString.split(' ')
4      # create an empty list that will store pairs of indices
5      pairs = [ ]
6      # append every relevant pair of indices to the pairs list
7      for i in range(len(word1)):
8          for j in range(len(word2)):
9              if word1[i] == word2[j]:
10                  thisPair = str(i) + ',' + str(j)
11                  pairs.append(thisPair)
12      # concatenate all the pairs together, separated by space characters
13      return ' '.join(pairs)

```

**Figure 10.8:** The Python program `matchingCharIndices.py` that solves the problem `MATCHINGCHARINDICES` from figure 10.7.

Determining the complexity of programs, as in the example immediately above, is one of the most important skills for a computer scientist. Anyone who writes code or uses algorithms, from the theoretician to the commercial software developer, needs to understand how the performance of their programs scales with input size. However, as we will see in the next chapter, computational complexity theory often ignores fine distinctions in running times, focusing instead on the distinction between polynomial running times and exponential running times. Therefore, an essential skill needed for this book is to look at a program and determine whether or not it runs in polynomial time.

### The lack of rigor in Python running times

Feel free to skip ahead to section 10.4; this subsection is mostly targeted at readers who are used to analyzing everything in terms of Turing machines. Such readers may be bothered by the lack of rigor in our definition of running time for Python programs, which relies on counting CPU instructions. For example, we didn't specify the exact point at which to start and stop counting CPU instructions. And the number of instructions used probably depends on which version of Python is employed and on which Python interpreter is used. Even worse, when we compute the running time of a Python program in practice (as in the examples above), mathematical rigor is abandoned completely, since we rely on a combination of common sense and the Python documentation.

We have reached one of the places in this book where it is tempting to admit defeat for the supposedly “practical” approach we are taking to theoretical computer science. But let's examine the issues and see why it makes sense to continue using Python programs as our primary computational model. Suppose that we instead adopted Turing machines as our main vehicle for analyzing computational complexity. We have a perfectly rigorous definition of running time for Turing machines. However, we have seen that describing Turing machines completely and precisely is time consuming, tedious, and error prone. For an example of this, look back at the `countCs` machine, which is described

in three separate figures (figures 5.10–5.12, pages 83–85). And this tremendously complicated machine performs the rather trivial task of counting the number of “C” characters in the input. We need to analyze algorithms of far greater sophistication.

For this reason, descriptions of Turing machines that implement sophisticated algorithms are never given in complete detail. That is, Turing machines are not described at the level of individual states and transitions. Instead, textbooks and published papers use some kind of higher-level description of a Turing machine’s operation. The higher-level description could use ordinary English prose, or pseudo-code. But another option—the option taken in this book—is to describe algorithms using a real programming language, such as Python. Note that all of these approaches to describing Turing machines (English prose, pseudo-code, or Python code) require common sense and good judgment when estimating running times. Therefore, the apparent lack of rigor in our Python running times is no better or worse than the other approaches generally used to describe and analyze Turing machines. But the use of Python code may help us to gain a deeper, more practical understanding of the theoretical results presented.

So, we will continue to use Python programs as a computational model. If mathematical rigor is desired, think of the Python program as a high-level description of a Turing machine. If practical understanding is desired, think of the Python program running on a real computer system, with the running time determined by counting CPU instructions.

## 10.4 FUNDAMENTALS OF DETERMINING TIME COMPLEXITY

We’ve now seen two examples of determining the complexity of a Python program: `containsGAGA.py` (which is in  $O(n)$ ) and `matchingCharIndices.py` (which is in  $O(n^2 \log n)$ ). This section covers several new ideas that allow us to determine the complexity of more subtle Python programs.

### A crucial distinction: The length of the input versus the numerical value of the input

The program `MCopiesOfC.py`, shown in figure 10.9, presents a crucial new idea: distinguishing between the length of the input and the numerical value of the input. In `MCopiesOfC.py`, the input is a string representing an integer  $M$  in decimal notation, such as “37”. The returned value is a single string consisting of  $M$  copies of the character “C”. For example,

```
>>> MCopiesOfC('37')
'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC'
```

This example also introduces some important new notation. Whenever the input to a program or problem consists of a single integer, we will denote the value of that integer by  $M$ . (If the input consists of more than one integer, they will usually be denoted by symbols like  $M$ ,  $M'$ ,  $M_1$ ,  $M_2$ .) So, our conventions are that  $I$  is the input string,  $n$  is the *length* of  $I$ , and  $M$  is the *numerical value* of  $I$  (assuming  $I$  represents an integer in decimal notation). In the example above,

```

1 def MCopiesOfC(inString):
2     M = int(inString)
3     cList = []
4     # iterate M times, appending a single "C" character at each iteration
5     for i in range(M):
6         cList.append('C')
7         # join all the C's together into a single string, with no separator
8     return ''.join(cList)

```

**Figure 10.9: The Python program `MCopiesOfC.py`.** On input  $M$ , the output is the string “ $\text{C}\text{C}\dots\text{C}$ ”, consisting of  $M$   $\text{C}$ ’s.

$I = "37"$ ,  $n = 2$ , and  $M = 37$ . Note that  $n$  is just the number of digits in  $M$ , which is  $\log_{10} M$  rounded up to the next integer.<sup>1</sup> This simple relationship will be used repeatedly, so here are some examples to demonstrate it:

$M$	$\log_{10} M$	$n$
7	0.86	1
256	2.41	3
987,654,321	8.99	9

In particular, we always have  $n = O(\log M)$  and  $M = O(10^n)$ . Note that if the input were given in binary notation, then we would have  $M = O(2^n)$ . But in this book, our inputs will usually be given in decimal notation, resulting in  $M = O(10^n)$ . Let’s summarize these relationships for easy reference later:

$$\begin{aligned} \text{length of input } n &= O(\log M), \\ \text{value of input } M &= \begin{cases} O(10^n) & \text{if input uses decimal notation,} \\ O(2^n) & \text{if input uses binary notation.} \end{cases} \end{aligned} \quad (\star)$$

Now let’s take a look at the program `MCopiesOfC.py` in detail. The program starts with an empty list (line 3), then appends a “ $\text{C}$ ” character to the list  $M$  times. Each append (line 6) requires constant time, so the `for` loop is in  $O(M)$  in aggregate. The `join` at line 8 results in a string of length  $M$ , and is therefore in  $O(M)$  also. Hence, the program as a whole requires  $O(M)$  time.

So, what is the time complexity (i.e., running time) of `MCopiesOfC.py`? Beware: This is a trick question. Before reading on, try to spot the trick and give a correct answer for the running time of `MCopiesOfC.py`.

As you probably guessed,  $O(M)$  is the wrong answer to this trick question. The problem is that we don’t measure running time in terms of  $M$ , which is the numerical value of the input. By definition, running time is always measured in terms of  $n$ , the *length* of the input. So, although it is true to say that the number of steps required to run the program is in  $O(M)$ , we must restate this result in terms of  $n$  before describing it as the running time or complexity of the program. Referring back to the relationship  $(\star)$  above, we can substitute  $M = O(10^n)$  into  $O(M)$ , obtaining  $O(10^n)$  as the correct running time for the program. Note the

<sup>1</sup> In Python,  $n=\mathbf{math.floor}(\mathbf{math.log}(M, 10)) + 1$ .

```

1 def multiply(inString):
2     (M1, M2) = [int(x) for x in inString.split()]
3     product = M1 * M2
4     return str(product)

```

**Figure 10.10: The Python program `multiply.py`.** A correct (but overly generous) bound on the complexity of this program is  $O(n^2)$ , because of the multiplication operation at line 3.

dramatic difference between the two expressions  $O(M)$  and  $O(10^n)$ : the number of steps required to run the program is linear in the value of the input, but exponential in the length of the input.

Complexity theorists do also study the running time of algorithms as a function of  $M$ . For example, an algorithm is defined to run in *pseudo-polynomial time* if the running time is bounded by a polynomial in  $M$ . And there are many applications in which  $M$  (rather than  $n$ ) is the most sensible measure of the “size” of the input. But in this book, we use only the standard and most fundamental version of time complexity. So we will always adhere to the following slogan:

**Slogan: Time complexity is measured as a function of the length of the input, not the numerical value(s) of the input.**

Note that many problems have inputs that don’t represent numbers, and the slogan is irrelevant to those problems. But the slogan does apply to the many important problems that have one or more numerical inputs, including factoring an integer, determining whether an integer is prime, computing the greatest common divisor of two integers, and multiplying two integers.

## The complexity of arithmetic operations

Recall the problem `MULTIPLY` from page 58: the input consists of two numbers  $M, M'$  (example: “34 100”), and the solution is the product  $MM'$  (using same example: “3400”). Figure 10.10 shows `multiply.py`, which solves this problem. By the way, line 2 of `multiply.py` uses a Python idiom that we haven’t seen before, known as a *list comprehension*. List comprehensions are a convenient way of writing `for` loops very compactly, and we’ll be using them occasionally from this point onwards. Consult some Python documentation if you’re interested in finding out more about them. The complexity of a list comprehension is the same as the equivalent `for` loop. For example, line 2 of `multiply.py`, which splits the input and applies the `int` function to each component, runs in  $O(n)$ .

Now back to our main topic: What is the complexity of `multiply.py`? Lines 2 and 4 are both in  $O(n)$ , but line 3 presents a mystery, since it uses the built-in Python multiplication operation. Is this also in  $O(n)$ ? The answer is no: instead, it turns out that  $O(n^2)$  is a correct asymptotic bound for multiplication.

One way to see this is to analyze the *grade-school multiplication algorithm*—that is, the algorithm for multiplying large numbers that you probably first learned as a child in school. Figure 10.11 shows an example, multiplying two numbers with

$$\begin{array}{r}
 & 1 & 2 & 3 & 4 \\
 & \times & 5 & 6 & 7 & 8 \\
 \hline
 & 9 & 8 & 7 & 2 \\
 & 8 & 6 & 3 & 8 \\
 & 7 & 4 & 0 & 4 \\
 & 6 & 1 & 7 & 0 \\
 \hline
 & 7 & 0 & 0 & 6 & 6 & 5 & 2
 \end{array}$$

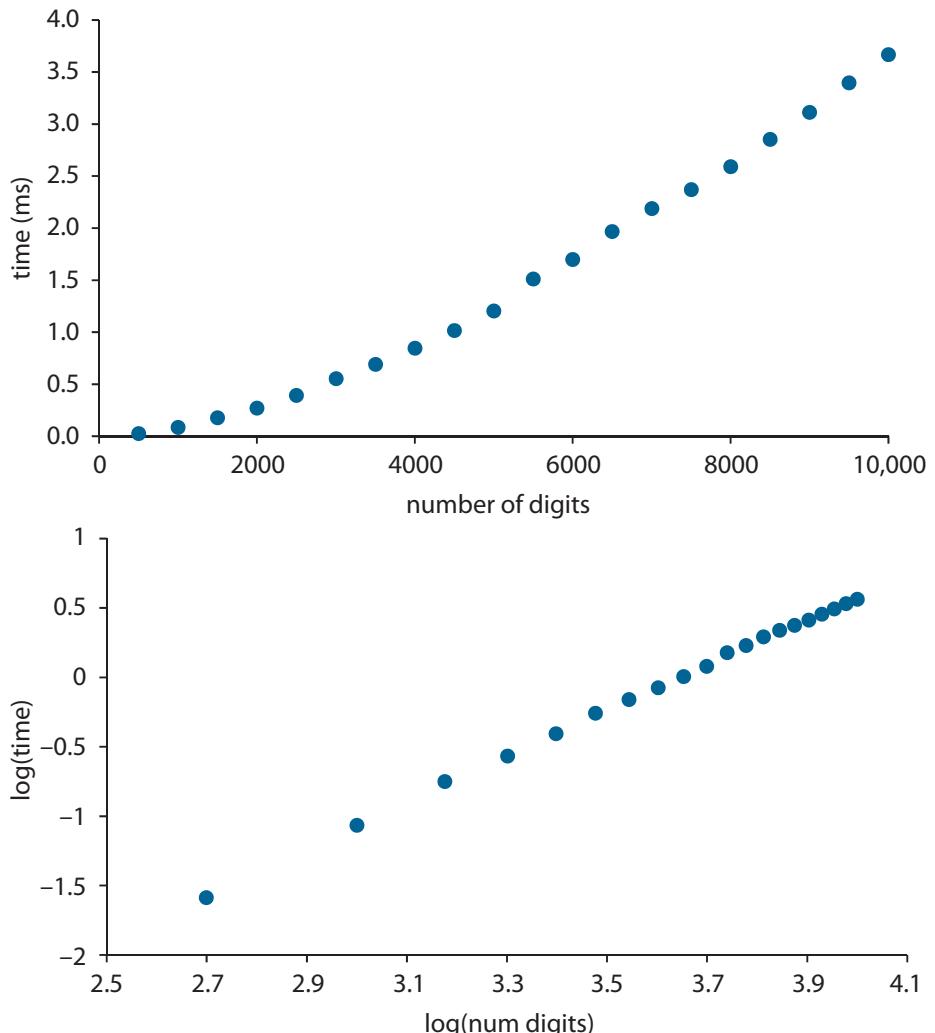
**Figure 10.11: An example showing the complexity of grade-school multiplication.**  
There are  $d = 4$  digits in each of the two numbers to be multiplied, but there are  $d^2 = 16$  pairs of digits to be multiplied (shown by the arrows).

$d = 4$  digits each. There are  $d^2 = 16$  pairs of digits to be multiplied (shown by the arrows), resulting in  $d^2 = 16$  new digits of intermediate results (between the two horizontal lines), which must be added for the final answer. Thus, grade-school multiplication is in  $O(d^2)$ . And since the number of digits  $d$  is less than the length of the input  $n$ , the algorithm is also  $O(n^2)$ .

In fact, when multiplying very large numbers, Python uses a more sophisticated multiplication algorithm that lies somewhere between  $O(n^2)$  and  $O(n)$ . Some timing results for my own computer are shown in figure 10.12. (You can run this experiment on your computer using `multiplyTimings.py`, provided with the book materials.) The top graph shows us two things. First, the amount of time it takes to multiply two numbers can be quite significant in absolute terms: we see here that it takes nearly 4 ms to multiply two 10,000-digit numbers. Second, the curved shape of the graph provides evidence that the running time is not linear. By taking the log of both quantities and replotted the same data points, we obtain the bottom graph, which is a much better fit to a straight line. The slope of the line is about 1.6, suggesting that Python's multiplication algorithm is in  $O(n^{1.6})$ . For simplicity, however, we will use the grade-school bound of  $O(n^2)$  in the remainder of the book. This bound is unnecessarily generous, but certainly correct.

We won't go into details for the other basic arithmetic operations. Simple analysis of the grade-school algorithms for these operations shows that division is also  $O(n^2)$ , while addition and subtraction are  $O(n)$ . These results are summarized as follows:

Operation	Asymptotic bound used in this book
multiplication	$O(n^2)$
division	$O(n^2)$
addition	$O(n)$
subtraction	$O(n)$



**Figure 10.12:** *Top:* The average time (in milliseconds) required for Python to use its built-in multiplication operation to multiply two  $d$ -digit random numbers. These results are for Python 3.4.1 running on an Intel Core i5 CPU. *Bottom:* The same results as the top graph, after taking the logarithm of both quantities.

### Beware of constant-time arithmetic operations

You may have noticed an apparent contradiction in this discussion. All modern CPUs have MULTIPLY and ADD instructions that are able to multiply or add two numbers in a single computational step. Therefore, surely these operations should require only  $O(1)$  (i.e., constant time) rather than  $O(n^2)$  or  $O(n)$ . The flaw in this argument is that the MULTIPLY and ADD instructions operate only on integers up to a fixed maximum size (say, 64 bits). Beyond this fixed maximum, more computational steps are required. In complexity theory, we are

### PROBLEM FACTOR

We need some terminology to describe FACTOR. Given an integer  $M > 1$ , a *nontrivial factor* of  $M$  is a number that divides  $M$ , other than 1 and  $M$ . For example, the nontrivial factors of 12 are 2, 3, 4, 6. If  $M$  has one or more nontrivial factors,  $M$  is *composite*. If  $M$  has no nontrivial factors,  $M$  is *prime*. For example, 12 is composite, but 13 is prime.

- **Input:** A positive integer  $M > 1$ , as an ASCII string in decimal notation.
- **Solution:** If  $M$  is composite, any nontrivial factor of  $M$  is a solution. If  $M$  is prime, the solution is “no”.

**Figure 10.13:** The computational problem FACTOR.

always concerned with the behavior as  $n$  becomes very large, so the  $O(n^2)$  and  $O(n)$  bounds are indeed the correct ones for us to use.

Beware: there are many computer science applications for which it is reasonable or useful to assume that all integer inputs to an algorithm are no larger than, say, 64 bits. (As one particular example, consider an image analysis algorithm that assumes the color of each pixel in the input image is represented by a 64-bit integer.) In such cases, it is correct to model all arithmetic operations, including multiplication and division, as  $O(1)$ . If you have taken a college-level algorithms course, you may have seen some complexity analysis that uses this  $O(1)$  assumption. As mentioned already, this is correct if the nature of the inputs justifies it. But in the remainder of this book, all our algorithms will deal with integers of arbitrary size. Therefore, we will always use  $O(n^2)$  for multiplication/division and  $O(n)$  for addition/subtraction. A subtle exception should be mentioned: multiplication/division of an arbitrary *single* integer by another *fixed* integer is only  $O(n)$ . See exercises 10.11 and 10.12 for details.

## The complexity of factoring

We now turn our attention to an important problem that will be used as an example in several later chapters: factoring an integer. A formal statement of the computational problem FACTOR is given in figure 10.13. Informally, the problem is to find a factor of the input integer  $M$ , or output “no” if  $M$  is prime. For example, on input “20”, the possible solutions are “2”, “4”, “5”, and “10”. On input “23”, the only solution is “no”, because 23 has no factors other than 1 and itself.

Figure 10.14 shows the Python program `factor.py`, which solves the problem FACTOR. The approach is very simple. We test every number between 2 and  $M-1$  inclusive. If any of these divides  $M$ , we return it immediately since we’ve found a factor. If none of them divides  $M$ , we return “no” since  $M$  must be prime. This program always returns the smallest factor if it exists; other approaches would return different solutions. Note that the program could be made more efficient if desired. One obvious improvement would be to search

```

1 def factor(inString):
2     M = int(inString)
3     for i in range(2,M):
4         if M % i == 0:
5             return str(i)
6     return 'no'

```

Figure 10.14: The Python program `factor.py`.

only up to  $\sqrt{M}$  (because if  $M$  has any nontrivial factors, at least one of them must be no larger than  $\sqrt{M}$ ). But we will stick with the naïve approach to keep our analysis simple.

What's the complexity of `factor.py`? The loop at line 3 is executed  $O(M)$  times, which is the same thing as  $O(10^n)$ . (If this isn't clear, reread the discussion of equation  $(\star)$  on page 211.) Inside the loop, we see the remainder operator “%” used at line 4. This is essentially the same as a division operation, and as we discussed on page 213, division is an  $O(n^2)$  operation. Thus, line 4 costs  $O(n^2)$  each time it is executed, giving an overall complexity for the program of  $O(n^2 10^n)$ . Finally, note that this is a crucial example of the distinction between the length of the input and the value of the input: integer factorization is polynomial in  $M$ , but exponential in  $n$ .

As usual, our big-O notation expresses an upper bound on the asymptotic running time. But the same analysis given above also results in an exponential lower bound for the worst case: we must allow about  $10^n$  operations for the execution of the `for` loop. Hence, `factor.py` requires exponential time in the worst case.

### The importance of the hardness of factoring

As already mentioned, FACTOR will be used as an example several times later in the book. So it will be useful to understand some background to this problem. FACTOR is of immense practical importance, since some commonly used cryptographic protocols rely on the assumption that factoring is hard. In particular, the popular public key cryptosystem known as RSA could easily be cracked if someone invented an efficient factoring algorithm.

RSA (which is an abbreviation based on its inventors—Rivest, Shamir, and Adleman) is frequently used to encrypt internet connections, enabling secure online transactions such as the communication of credit card numbers and banking information. When a computer  $C$  uses RSA for security,  $C$  generates two keys: a public key  $K$  and a private key  $K'$ . Both keys are large integers, perhaps thousands of digits in length. The value of the public key  $K$  is publicly available, and it is used to encrypt messages. So any other computer can use  $K$  to encrypt a message and send it to  $C$ . But the value of  $K'$  is kept secret and is known only to  $C$ . This private value is used to decrypt messages received by  $C$ . If any other computer knew the value of  $K'$ , it could decrypt  $C$ 's messages and thus defeat RSA's security.

But here's the catch:  $K$  and  $K'$  are related to each other by a certain mathematical transformation. It's widely believed there is no efficient way of computing  $K'$  from  $K$  directly. However, if the factors of  $K$  are known, then there is an efficient way to derive  $K'$  (and thus overcome RSA). In other words: *if there exists an efficient factoring algorithm, then the RSA cryptosystem is broken.*

As discussed already, the naive algorithm used by `factor.py` requires exponential time (about  $10^n$  steps), which is far too time consuming for cracking RSA. There do exist significantly better approaches, but even the best known algorithms for factoring still require superpolynomial time. That's why RSA is currently considered secure. But computer scientists have not been able to prove that factoring requires superpolynomial time, so it's theoretically possible that an efficient factoring algorithm could be invented in the future.

Factoring is important for other reasons too, but this fundamental connection to cryptography is enough to demonstrate the importance of FACTOR as a computational problem.

## The complexity of sorting

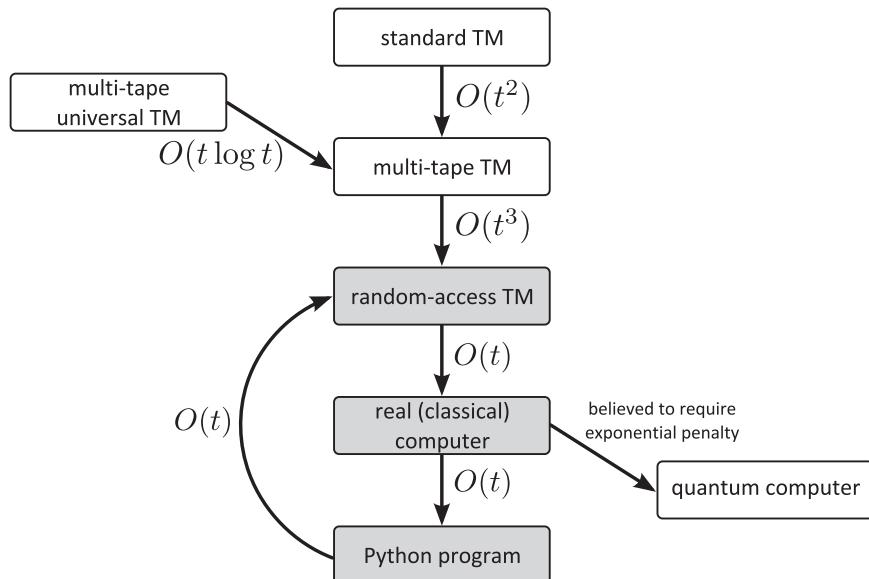
Many computational algorithms involve sorting lists of strings into lexicographical order. This amounts to solving the problem SORTWORDS on page 46. As you probably know, algorithms such as *heap sort* and *merge sort* solve this problem efficiently. The details of these algorithms aren't important for this book. All we need to know is that their running time is in  $O(n \log n)$ . Real-world experiments using Python's `sorted` function show that this bound also holds in practice. See exercise 10.14 for a little more discussion.

## 10.5 FOR COMPLEXITY, THE COMPUTATIONAL MODEL DOES MATTER

We are by now very familiar with the fact that, in terms of computability, standard Turing machines have exactly the same power as the most sophisticated computers ever built. In contrast, it turns out that through the lens of computational complexity, standard Turing machines are actually *less powerful* than typical computers. (As usual, our “typical computer” here is idealized: it can access as much storage as necessary.) The fundamental problem is that standard Turing machines have only one head and one tape, so maintaining a nontrivial data structure on the tape requires a lot of movement by the head. (For example, if the head is currently at the left end of the tape,  $n$  steps are required just to read the  $n$ th cell of the tape.) As a result, standard Turing machines do require more steps to achieve the same outputs as multi-tape Turing machines or real computers.

### Simulation costs for common computational models

As discussed above, the choice of computational model doesn't affect computability, but it *does* affect time complexity (i.e., running times). The next few subsections describe the relationships between running times in some of the common



**Figure 10.15: The cost of simulating one type of computational model with another.**

The meaning of an arrow like  $A \xrightarrow{O(f(t))} B$  is that a  $t$ -step computation on  $B$  can be simulated on  $A$  in  $O(f(t))$  steps. The three shaded models differ only by constant factors, and any one of these can be taken as the standard computational model for the remainder of the book.

computational models, as summarized in figure 10.15. These relationships are stated without proofs, but some informal justifications are provided.

As usual, all our results will use big-O notation, so we are ignoring multiplicative constants. Extensive justification for doing this has already been given in section 10.1. But it's worth mentioning a further reason here. There is a result known as the “linear speedup theorem” for Turing machines. Roughly speaking, this theorem states that given a single-tape Turing machine  $M$ , we can construct a new single-tape machine  $M'$  that runs  $C$  times faster than  $M$ , for any value of  $C$ . The trade-off is that machine  $M'$  has a larger alphabet than  $M$ , and a much more complex transition diagram. Nevertheless, it's clear that this construction makes multiplicative constants less relevant, further justifying the use of big-O notation.

### Multi-tape simulation has quadratic cost

Simulating a multi-tape Turing machine (with multiple independent heads) on a standard Turing machine imposes, at most, a quadratic penalty. That is, an algorithm that terminates in  $O(f(n))$  steps on a multi-tape machine becomes  $O(f(n)^2)$  when implemented on a single-tape machine (where  $n$  is the length of the input). The extra factor of  $f(n)$  arises because after  $t$  steps, the multi-tape machine has written in at most  $t$  cells. The standard machine can simulate a single step of the multi-tape machine by sweeping once over all  $t$  cells, finding the “x” symbols of figure 5.16 and taking appropriate actions. So the simulation

We aim to simulate a random-access machine  $R$  on a multi-tape machine  $M$ . To do this, we simulate  $R$ 's RAM with a dedicated tape on  $M$  that records both locations and content. Call this dedicated RAM-simulating tape *SimRam*. The operation of the SimRam tape can be understood by an example. If SimRam's non-blank symbols are “4:a;57:b;899:x”, this corresponds to machine  $R$  having  $\text{RAM}[4] = a$ ,  $\text{RAM}[57] = b$ ,  $\text{RAM}[899] = x$ .

Recall from page 92 that  $R$ 's RAM is initially blank and is not used for input or output. The key question turns out to be, how many non-blank symbols could be written on the SimRam tape after  $t$  steps? Some careful thought (and here we are admittedly skipping over some important details) shows that when  $R$  executes its  $t$ th step, the number of non-blank SimRam symbols can increase by at most  $O(t)$ . (Consider what happens when  $R$  writes successively to RAM locations 1, 10, 100, 1000, …, and see exercise 5.18 for additional details.) Thus, there are  $O(t^2)$  non-blank symbols on the SimRam tape after  $t$  steps of  $R$  have been simulated by  $M$ . Simulating a read or write from  $R$ 's RAM therefore costs  $O(t^2)$  on  $M$ , since we might need to scan the entire SimRam tape to find the relevant entry. This  $O(t^2)$  cost occurs for each of  $t$  steps, resulting in total cost  $O(t^3)$ —explaining the  $O(t^3)$  on the second vertical arrow of figure 10.15.

**Figure 10.16:** A brief and informal sketch of the  $O(t^3)$  simulation of a random-access machine by a multi-tape machine.

requires at most  $t^2$  steps, which explains the  $O(t^2)$  on the first vertical arrow of figure 10.15.

### Random-access simulation has cubic cost

Simulating a random-access Turing machine on a multi-tape Turing machine imposes, at most, a cubic penalty. That is, an algorithm that terminates in  $O(f(n))$  steps on a random-access machine  $R$  becomes  $O(f(n)^3)$  when implemented on a multi-tape machine  $M$ . (As usual,  $n$  is the length of the input.) If you're curious, some of the details of how to achieve this simulation are given in figure 10.16, but you should probably skip this on first reading. Papadimitriou gives a more detailed analysis of this style of simulation in his 1994 book.

### Universal simulation has logarithmic cost

Simulating an arbitrary multi-tape Turing machine on a universal multi-tape Turing machine imposes a logarithmic penalty. More specifically, there exists a surprisingly efficient, universal multi-tape machine  $U$  with the following property. Given *any* multi-tape machine  $M$  that terminates in  $t$  steps on input  $I$ , we can send a description of  $M$  and  $I$  into  $U$ , and  $U$  will terminate with the same answer as  $M$  within  $O(t \log t)$  steps. The constant hiding in the big- $O$  here depends on  $M$ 's alphabet size, number of tapes, and number of states, but it does not depend on  $I$ . This universal simulation is represented by the  $O(t \log t)$  on the upper diagonal arrow of figure 10.15. Justifying this simulation

requires some technical tricks that are beyond the scope of this book; Arora and Barak provide a detailed description in their 2009 book. But the result itself is very striking: universality requires only a logarithmic penalty. Combining this with the discussion in section 6.3 (page 107), we conclude that universality is both surprisingly widespread, and surprisingly efficient.

### Real computers cost only a constant factor

Recall from page 92 how we sketched a proof that a random-access Turing machine can simulate a given real computer system  $C$ : we need to check only that each of the instructions in  $C$ 's instruction set can be simulated by the random-access Turing machine. To determine the cost of this simulation, we need to find just the worst-case instruction. That is, what is the greatest number of steps that could be required to simulate a single instruction from  $C$ ?

It would be too tedious for us to examine these details here, but it turns out that the individual instructions in modern instruction sets can all be simulated using a *fixed* maximum number  $K$  of Turing machine steps. For example, it might turn out that for some particular instruction set, the 64-bit MULTIPLY instruction is the most difficult to simulate, requiring 2000 Turing machine steps. Then we would have  $K = 2000$ , and we would know that any program that runs in  $t$  instructions on  $C$  requires at most  $2000t$  steps to be simulated on a random-access Turing machine.

Thus, the penalty for simulating a real computer on a random-access Turing machine is at most a constant factor, which explains the  $O(t)$  on the third vertical arrow of figure 10.15.

### Python programs cost the same as real computers

We defined the running time of Python programs in terms of CPU instructions, which means the running time of a Python program is identical to the running time of an equivalent program written in assembly language. Thus, the fourth and final vertical arrow of figure 10.15 could be labeled  $t$ , but we have left it as  $O(t)$  for consistency with the other arrows.

### Python programs can simulate random-access Turing machines efficiently

It's not hard to write a Python program that (when run with sufficient memory) simulates a random-access Turing machine. One can check that each step of the Turing machine requires only a constant number of CPU instructions, meaning the simulation runs in  $O(t)$  time. This justifies the upwards curving arrow of figure 10.15.

### Quantum simulation may have exponential cost

We've already seen (section 5.6, page 98) that quantum and classical computers are Turing equivalent, so they can solve exactly the same set of problems.

But the story is different when we take efficiency into account. There exist computational problems for which the best known algorithm for classical computers is exponentially slower than the best known algorithm for quantum computers. Thus, although no one has yet proved it, most computer scientists believe that a classical computer requires an exponential penalty when simulating a quantum computer. This explains the arrow to “quantum computer” in figure 10.15.

### All classical computational models differ by only polynomial factors

Note that each of the classical computational models differs from the others in figure 10.15 by at most a polynomial. By composing all of the simulations from a standard Turing machine to a Python program (i.e., following all of the vertical arrows in sequence), we get a total simulation cost of  $O(O(t^2)^3) = O(t^6)$ , which is a polynomial. Thus, switching between any two classical models results in, at worst, a polynomial slowdown. It’s only when we go from the quantum model to a classical model that we might end up with an exponential slowdown.

### Our standard computational model: Python programs

Some of the results in complexity theory are true for a wide variety of computational models, whereas other results are rather sensitive to which computational model is chosen. Therefore, let’s agree to use one particular computational model for the remainder of this book.

Our standard computational model will be the Python program. Python programs will work well for our approach, because of their practical flavor. However, note from figure 10.15 (page 218) that the three shaded models differ by only constant factors. And because constant factors do not affect our asymptotic running times, these three models are equivalent for our purposes. Thus, random-access Turing machines or classical computers would serve equally well as the standard computational model in this book.

## 10.6 COMPLEXITY CLASSES

We already know how to define the complexity of a *program*, such as `factor.py`. Now, we’ll attempt to define the complexity of a *problem*, such as `FACTOR`. What’s the difference? The key point is that a computational problem can be solved by many different programs, and these programs might have different complexities. It could be difficult or impossible to find the single most efficient program that solves a given problem. Because of this, problem complexities are defined in a different way, via complexity classes.

A *complexity class* is a collection of computational problems that share some property related to complexity. For example, we define `Lin` to be the set of all problems that can be solved by an  $O(n)$  Python program. That is, `Lin` is the

problems that can be computed in linear time. Examples of problems in Lin include

- COUNTCS: Solution is the number of times “C” occurs in the input;
- SUM2: Input is two positive integers in decimal notation separated by a space; the solution is the sum of the two integers;
- CONTAINSGAGA: Solution is yes if the string “GAGA” occurs anywhere in the input; otherwise no.

Another example of a complexity class is Const, the set of problems solvable in constant time, which is usually written  $O(1)$ . Examples of Const problems are

- HUNDREDTHCHAR: Solution is the 100th character of the input (or “no” if the input is shorter than 100 characters);
- STARTSWITHGAGA: Solution is “yes” if the input string starts with GAGA; otherwise no.

We could also define various other meaningful classes if we choose to. For example, LogLin could be the class of problems with  $O(n \log n)$  methods of solution; Quad could be quadratic problems—that is,  $O(n^2)$ ; and so on. More generally, we can define a complexity class for any reasonable function  $f(n)$ :

**Definition of Time( $f(n)$ )**. Let  $f(n)$  be a function mapping positive integers to positive real numbers.<sup>2</sup> The class Time( $f(n)$ ) is defined as the set of computational problems that can be solved by some Python program whose running time is in  $O(f(n))$ .

Each of the complexity classes mentioned so far can be put into this framework:

$$\begin{aligned} \text{Const} &= \text{Time}(1), \\ \text{Lin} &= \text{Time}(n), \\ \text{Quad} &= \text{Time}(n^2), \\ \text{LogLin} &= \text{Time}(n \log n). \end{aligned}$$

These examples of complexity classes are both simple and important. But you won’t usually see the classes Const, Lin, Quad, and LogLin discussed in books on complexity theory. The reason is that these classes are too fine-grained: they capture details that turn out to be relatively unimportant for general statements about algorithms and computation.

It’s worth going into more detail about this. What is it about Lin, for example, that makes it too “fine-grained” for complexity theorists to study? There are at least two answers. The first answer is that Lin depends on the computational model being used. Recall from page 218 that there is generally a quadratic

<sup>2</sup> The function  $f$  must also be *time constructible*, which means, roughly speaking, that  $f(n)$  can be computed in time  $O(f(n))$ .

penalty to transfer an algorithm from a multi-tape Turing machine to a single-tape Turing machine. Therefore, it's possible that a computational problem  $F$  is in  $\text{Lin}$  when using the multi-tape model, but  $F$  is in  $\text{Quad}$  when using the single-tape model. The same issue affects other fine-grained complexity classes. (But see exercise 10.15 for an interesting exception:  $\text{Const}$  does not depend on the computational model.)

There is a second answer to the question of why classes like  $\text{Lin}$  are too fine-grained for complexity theory. This answer is a purely empirical observation: these fine-grained classes haven't produced many profound, beautiful, or important theoretical results. The deep and intriguing results of complexity theory seem instead to derive from more general properties of algorithms—such as whether they require exponential time, or whether they require randomness.

The distinctions between complexity classes like  $\text{Lin}$  and  $\text{Quad}$  can be very important, of course. For example, if the typical input to a given problem in genetics is a string of one billion bases, then a quadratic-time method of solution is probably useless, whereas a linear-time approach may well be practical. On the other hand, the objective of this book is to give general (and hopefully profound) answers to the question, what can be computed? And it turns out that the most useful answers to this question ignore the distinctions between complexity classes like  $\text{Lin}$  and  $\text{Quad}$ .

Instead, we will be defining much more general complexity classes. Formal definitions are given later, but it will be useful to have a sketch of the big picture now. So, here are informal descriptions of the three most important complexity classes:

- **Poly:** problems that can be solved in polynomial time
- **Expo:** problems that can be solved in exponential time
- **PolyCheck:** problems whose solutions can be verified in polynomial time

Other important complexity classes include **NPComplete** and **NPHard**, but they won't be defined until chapter 14.

This discussion should be sounding eerily familiar. Back in section 10.1 (page 195), we took a significant step away from implementation details: we agreed to use asymptotic running times (i.e., big-O) instead of absolute running times. Now we are taking another step away from implementation details. This time we will be clumping together all of the running times like  $O(n)$ ,  $O(n^3)$ , and  $O(n^{25})$  in the general class **Poly**. The classes **Expo** and **PolyCheck** arise from similarly large “clumps” of running times.

The high-level message is the same as in section 10.1. There, we emphasized that in any practical application, both the absolute and asymptotic running times should be understood. Let's now add general complexity classes (such as **Poly**, **Expo**, and **PolyCheck**) to this list of things that must be understood. In other words, whenever you try to solve a computational problem  $F$  in practice, you need to understand the following issues:

1. Which general complexity classes does  $F$  belong to? More precisely, is  $F$  in **Poly**, **PolyCheck**, **Expo**, **NPComplete**, and/or **NPHard**?
2. What is the asymptotic running time of your proposed method for solving  $F$ ?

3. What is the absolute running time of your proposed method for solving  $F$ ?

So, the next few chapters concentrate on the first question above, while acknowledging that the second and third questions are also important in any practical scenario.

## EXERCISES

- 10.1** Give the dominant term of each of the following functions:

- (a)  $f(n) = (\log_{10}(7n + 2))^3 + 9(\log_2(n^7))^2$
- (b)  $f(n) = 5n^7 + 2^{3n}$
- (c)  $g(n) = an^3 \log_2(n) + b(n \log_2(n))^3$  for some constants  $a, b > 0$
- (d)  $f(n) = g(h(n))^2$ , where  $h(n) = n^2 \log_2 n$  and  $g(n) = 3n^2 + 5n^4 + 6$

- 10.2** Given the following collection of functions  $f_1, f_2, f_3, f_4$ , list all pairs of the functions such that  $f_i \in O(f_j)$ ,  $i \neq j$ :

$$\begin{array}{ll} f_1(n) = n(\log n)^8 & f_2(n) = 5n^2 + 4(n+6)^3 2^n \\ f_3(n) = n^3 2^n & f_4(n) = f_1(n)f_3(n) \end{array}$$

- 10.3** Answer true or false to the following:

- (a)  $5n^3 \in O(n^3 \log n)$
- (b)  $n^2 n! \in O(2^n)$
- (c)  $\log(5n^3 + n \log n) \in O(\log n)$
- (d)  $\log n \in O(\log(5n^3 + n \log n))$
- (e)  $(6n^2 - 2n - 4)^3 \in O(n^7)$
- (f)  $n^8$  is subexponential
- (g)  $n^8$  is superpolynomial
- (h)  $n!$  is subexponential
- (i)  $n!$  is superpolynomial

- 10.4** This exercise asks you to prove some of the results in figure 10.3, page 204.

- (a) Prove that any set of  $k$  elements has exactly  $2^k$  subsets. (This result is usually proved using mathematical induction, a technique that we don't use in this book. In fact, the result can also be proved without explicitly using induction, as suggested by the following hint: Let  $N(k)$  denote the number of subsets of a set of  $k$  elements. Then  $N(k) = 2N(k-1) = 2^2N(k-2) = \dots = 2^kN(0) = 2^k$ . The only remaining task is to explain each of the links in this chain of equalities.)
- (b) Let  $K$  be a set of  $k$  elements, and let  $s \leq k$  be a positive integer. Prove that the number of subsets of  $K$  that have size exactly  $s$  is in  $O(k^s)$ .
- (c) Let  $K, k, s$  be as above. Prove that the number of subsets of  $K$  that have size at most  $s$  is in  $O(k^s)$ .
- (d) It might at first seem paradoxical that parts (b) and (c) both yield  $O(k^s)$ , since there must be many more sets of size at most  $s$  than there are of size exactly  $s$ . Explain in your own words why this result is in fact perfectly reasonable.

- (e) Recall the definition of a Hamilton cycle from page 47. Prove that the number of Hamilton cycles in a graph with  $k$  nodes is in  $O(k!)$ .
- (f) Recall the definition of a Hamilton path from page 47. Prove that the number of Hamilton paths between two given nodes in a graph with  $k$  nodes is in  $O(k!)$ .

**10.5** What is the exact running time of the `containsGAGA` Turing machine (figure 10.4, page 205) on input

- (a) GGGGGAGAGGGGG
- (b) GGGGGTGTGGGGG

**10.6** Make a reasonable estimate of the time complexity of the following Python programs:

- (a) `slow10thPower.py` (figure 10.17, page 226)
- (b) `countLines.py` (figure 3.1, page 32)
- (c) `pythonSort.py` (figure 4.2, page 47)
- (d) `containsNANA.py` (figure 8.2, page 145)
- (e) `listEvens.py` (figure 10.18, page 226)
- (f) `mysteryMultiply.py` (figure 10.19, page 226)

**10.7** Suppose `foo.py` is a Python program with time complexity  $O(n^4)$ . Give a reasonable estimate of the time complexity of simulating `foo.py` on each of the following computational models (or, if it is not possible to give a reasonable estimate, explain why):

- (a) A standard Turing machine
- (b) A multi-tape Turing machine (with multiple independent heads)
- (c) A random-access Turing machine
- (d) A multi-tape universal Turing machine (with multiple independent heads)
- (e) A quantum computer

**10.8** Suppose a particular Python program  $P$  has running time  $O(n \log n)$ . Give a reasonable estimate of the time complexity of simulating  $P$  on a multi-tape universal Turing machine with multiple independent heads.

**10.9** This problem concerns only the following complexity classes: `Const`, `Lin`, `LogLin`, `Quad`. For each of the computational problems given below, state which of the above complexity classes the problem belongs to. (Note: (i) Each problem receives a single ASCII string as input and (ii) a problem can belong to multiple complexity classes.)

- (a) `COUNTCs`: Solution is the number of C's in the input.
- (b) `CHAR10000`: Solution is the 10,000th character of the input string, or “no” if the input is shorter than 10,000 characters.
- (c) `PAIRSOFSTWORDS`: Input is split into words  $w_i$  separated by whitespace. Solution is a list of all ordered pairs  $(w_i, w_j)$  of these words such that  $w_i$  starts with S and  $w_j$  starts with T. Each ordered pair appears on a separate line with the two elements separated by a space character.
- (d) `IS_SORTED`: Input is split into words separated by whitespace. Solution is “yes” if the words are sorted in lexicographical order, and “no” otherwise.

```

# Input is a nonnegative integer M in decimal notation. Output is M10.
2 # We could compute this efficiently using the Python ** operator,
# but here we deliberately use a slow iterative method.
4 def slow10thPower(inString):
    M = int(inString)
6    product = 1
    for i in range(10):
8        product = product * M
10   return str(product)

```

Figure 10.17: The Python program `slow10thPower.py`.

```

# The input should be a positive integer M in decimal notation. This
2 # program returns a list of all positive even integers less than M,
# separated by commas.
4 def listEvens(inString):
    M = int(inString)
6    evens = []
    for i in range(2,M):
8        if i % 2 == 0:
            evens.append(str(i))
10   return ','.join(evens)

```

Figure 10.18: The Python program `listEvens.py`.

```

# This is a strange and useless program, but it is good for practicing
2 # the skill of estimating time complexity. The input is assumed to
# be a positive integer M in decimal notation.
4 def mysteryMultiply(inString):
    # make M concatenated copies of the input
6    copiedInString = int(inString) * inString
    # convert to integer and perform the mystery multiply
8    val = int(copiedInString)
    return str(val*val*val)

```

Figure 10.19: The Python program `mysteryMultiply.py`.

- (e) **NUMERICALSORT:** Input is split into words separated by whitespace. Words that do not represent positive integers in decimal notation are ignored. Solution is a list of the words that do represent positive integers, sorted in increasing numerical order and separated by space characters.
- (f) **ENDSINZ:** Solution is “yes” if the input ends in `Z`, and “no” otherwise.
- (g) **REPEATSACHARACTER:** Solution is `yes` if the input contains two instances of the same symbol; otherwise `no`.

**10.10** Let  $\text{SUMK}$  be the function problem of adding  $K$  integers. Formally, the input consists of the list  $K, M_1, M_2, \dots, M_K$  separated by whitespace, and the unique solution is  $\sum_{k=1}^K M_k$ . Prove that  $\text{SUMK} \in \text{Lin}$ .

**10.11** Let  $\text{MULTBY496}$  be the function problem whose input is a single integer  $M$  and whose unique solution is  $496M$ . Prove that  $\text{MULTBY496} \in \text{Lin}$ . (Note: This does not contradict section 10.4's discussion about  $O(n^2)$  multiplication. It is true that  $\text{MULTIPLY} \notin \text{Lin}$ , since multiplying any *two* arbitrary integers requires quadratic time. Nevertheless, the problem of multiplying *one* arbitrary integer by one *fixed* integer, such as 496, requires only linear time.)

**10.12** Let  $\text{MOD7}$  be the function problem whose input is a single integer  $M$  and whose unique solution is  $M$  modulo 7. For example, the solution for input “79” is “2”. Prove that  $\text{MOD7} \in \text{Lin}$ . (Note: This does not contradict section 10.4's discussion about  $O(n^2)$  division, for the same reasons discussed in the previous exercise.)

**10.13** Let  $\text{RESTRICTEDFACTOR}$  be a computational problem the same as  $\text{FACTOR}$  (page 215), except that we are interested only in finding factors in the interval  $[\sqrt{M}/2, \sqrt{M}]$ . What is the complexity of a program that solves  $\text{RESTRICTEDFACTOR}$  by testing all integers in this range? (Hint and discussion: It is exponential, just like `factor.py`, but with a smaller base in the exponential expression. This is one particular case of the computational problem  $\text{FACTOR-INRANGE}$ , introduced later on page 237, and it suggests that finding factors in a restricted range can still require superpolynomial time using the best known algorithms.)

**10.14** The well-known  $O(n \log n)$  bound for sorting algorithms, mentioned on page 217, is correct but deserves more careful analysis. Consider the special case in which there are  $k$  strings to be sorted, each exactly of length  $c$  characters. Then  $n = kc$  (here we neglect the need for delimiters between the strings). The algorithmic theory of sorting tells us we need  $O(k \log k)$  comparisons, each of which will require  $O(c)$  time in this case. So the total time is in  $O(ck \log k) = O(n \log k)$ , somewhat less than our original estimate (especially if we think of  $k$  typically being in  $O(\log n)$ ). But in this special case, we can do even better by using the sorting algorithm known as *radix sort*, which is in  $O(ck) = O(n)$ . Implement radix sort in Python and compare its efficiency with the built-in `sorted` function.

**10.15** Prove that the complexity class  $\text{Const}$  does not depend on the computational model.

# 11



## Poly AND Expo: THE TWO MOST FUNDAMENTAL COMPLEXITY CLASSES

One finds, however, that some computable sequences are very easy to compute whereas other computable sequences seem to have an inherent complexity that makes them difficult to compute.

—Hartmanis and Stearns, *On the Computational Complexity of Algorithms* (1965)

The most fundamental idea in computational complexity theory is the distinction between problems that can be solved in polynomial time and those that require superpolynomial time. This chapter introduces the two complexity classes that we need in order to study this idea in detail: Poly and Expo. It then goes on to investigate the boundary between these complexity classes. But first, we need formal definitions of the classes themselves.

### 11.1 DEFINITIONS OF Poly AND Expo

**Definition of Poly.** The complexity class Poly is the set of computational problems that can be solved by a Python program with running time in  $O(n^k)$ , for some  $k \geq 0$ .

Examples of problems in Poly include CONTAINSGAGA, MULTIPLY, and SORT-WORDS, which can be solved by programs that are  $O(n)$ ,  $O(n^2)$ , and  $O(n \log n)$  respectively.

Note that, in the definition of Poly above, the condition  $O(n^k)$  is equivalent to saying “polynomial running time,” since any polynomial is in  $O(n^k)$  for some  $k$ . More generally, any combination of polynomial and logarithmic terms is also in  $O(n^k)$ —this includes functions like  $27n^6 + 5n^3$  and  $n^2(\log n)^5 + (\log n)^{23}$ .

**Definition of Expo.** The complexity class Expo is the set of computational problems that can be solved by a Python program with running time in  $O(2^{p(n)})$ , for some polynomial  $p(n)$ .

### PROBLEM MCOPIESOFC

- **Input:** A positive integer  $M$  as a string in decimal notation. Example: “524”.
- **Solution:** A string consisting of  $M$  copies of the symbol “C”. Example: On input “5”, the solution is “CCCCC”.

**Figure 11.1:** Description of the computational problem MCOPIESOFC.

The polynomial  $p(n)$  in the definition of **Expo** might seem a little mysterious. Most of the time, you can get away with thinking of **Expo** as  $O(2^n)$ —or perhaps  $O(2^{cn})$  for some  $c \geq 1$ . But the formal definition also includes even faster-growing possibilities like  $O(2^{5n^3})$ , which is why we allow a general polynomial  $p(n)$  in the exponent.

To see how this definition works in practice, let’s prove that the problems MCOPIESOFC (figure 11.1) and FACTOR (figure 10.13, page 215) are both in **Expo**.

**Claim 11.1.** MCOPIESOFC is in **Expo**.

*Proof of the claim.* The program `MCopiesOFC.py` (figure 10.9, page 211) solves MCOPIESOFC, and the discussion on page 210 shows its running time is in  $O(10^n)$ . So it remains only to prove that  $O(10^n)$  is in  $O(2^{p(n)})$  for some polynomial  $p$ . But  $10 < 2^4$ , so  $10^n < (2^4)^n = 2^{4n}$ . Thus  $O(10^n)$  is in  $O(2^{4n})$ —that is, we can take our polynomial  $p$  to be  $4n$ .  $\square$

This kind of manipulation of exponents is an essential tool, and will be used without comment from now on.

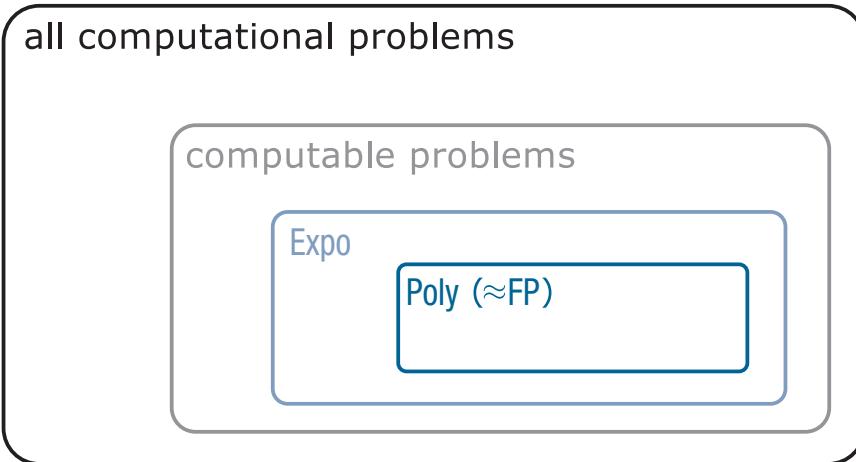
**Claim 11.2.** FACTOR is in **Expo**.

*Proof of the claim.* The program `factor.py` (figure 10.14, page 216) solves FACTOR, and the discussion on page 215 shows its running time is in  $O(n^2 10^n)$ . So it remains only to prove that  $O(n^2 10^n)$  is in  $O(2^{p(n)})$  for some polynomial  $p$ . But  $n^2 10^n$  is dominated by  $11^n$  (see list (★), page 199). And the same argument as in the previous proof shows that  $11^n$  is in  $O(2^{4n})$ , so we can again take  $p = 4n$ .  $\square$

The previous two proofs are a little more detailed than necessary. As a rule of thumb, if a function looks like it’s exponential, it is exponential. Exceptions arise only when the exponent itself grows faster than any polynomial. For example, a problem that requires  $O(2^{(2^n)})$  time is not in **Expo**. (This growth rate is called *double exponential*.)

## Poly and Expo compared to P, Exp, and FP

You can skip this subsection unless you are interested in comparing our main complexity classes (Poly and Expo) with certain complexity classes commonly defined in other textbooks (P, Exp, and FP). Recall that in this book we are aiming for a practical approach, and therefore we work with general computational problems rather than decision problems when possible. So Poly and Expo were



**Figure 11.2:** Poly is a subset of Expo, which is a subset of the computable problems. Poly is essentially identical to the class FP, but conflicting definitions of FP are in use (see page 229).

defined as classes of *general* problems. If we restrict these classes to *decision* problems only, we get the well-known classes P and Exp. Chapter 14 gives formal definitions and more detailed analysis of P and Exp.

Note that Poly is closely related to another complexity class known as FP, for “function polynomial-time.” Unfortunately, there are at least two conflicting definitions of FP in use: some books define FP as the class of *function* problems solvable in polynomial time, and others define FP as the class of *general* problems solvable in polynomial time. (Recall from page 58 that function problems have unique solutions, whereas general problems can have solution sets of any size.) Our definition of Poly is essentially identical to the second version of FP.

## 11.2 Poly IS A SUBSET OF Expo

Recall a basic fact about big-O notation: it incorporates the idea of “at most.” (See “Common big-O mistake #1” on page 202 for a reminder of this.) In particular, an  $O(2^n)$  algorithm requires *at most*  $c2^n$  steps (for some constant  $c$ , and for big enough  $n$ ). The algorithm may in fact be polynomial time, or linear time, or even constant time, but it is still technically correct to say that it is in  $O(2^n)$ . This basic property of big-O notation implies that Poly is a subset of Expo, as shown in the following claim.

**Claim 11.3.** Poly is a subset of Expo.

*Proof of the claim.* Given any basic polynomial  $n^k$ , the function  $2^n$  dominates  $n^k$  (see list (\*), page 199). Thus, any algorithm that is in  $O(n^k)$  is also in  $O(2^n)$ . Equivalently, any problem in Poly is also in Expo, and hence Poly is a subset of Expo.  $\square$

Figure 11.2 summarizes our state of knowledge about computational problems at this point: Among all computational problems, some are computable;

some of those can be solved in exponential time; and some of those can also be solved in polynomial time. It's known that each of these inclusions is *proper*—that is, each subset is strictly smaller than the one containing it. We have already proved that computable problems are a proper subset of all problems; this follows from the existence of uncomputable problems such as YESONSTRING and the halting problem. The fact that Poly is a proper subset of Expo will be proved in the next section.

## 11.3 A FIRST LOOK AT THE BOUNDARY BETWEEN Poly AND Expo

Take another look at figure 11.2. Each of the boundaries between one class and another has great importance in complexity theory, but the most fundamental is the boundary of Poly. Whenever we encounter a new computational problem  $F$ , we must first ask the question, is  $F$  a member of Poly? In this section, we will see that the answer to this question can be surprisingly subtle. We are going to analyze three pairs of related problems. In each pair, one of the two problems is inside Poly and the other is believed to be outside it. The objective is to understand that relatively small changes in a problem can cause big changes in complexity.

We'll tackle each pair of problems separately, beginning with ALL3SETS and ALLSUBSETS.

### ALL3SETS and ALLSUBSETS

The problems ALL3SETS and ALLSUBSETS are described formally in figure 11.3. Each takes as input a list of integers, and outputs a list of *subsets* of those integers. ALL3SETS outputs all the *3-sets* (i.e., subsets of size 3). ALLSUBSETS outputs all of the subsets, including the empty set.

You are strongly urged to write your own Python programs solving these problems, before reading on. Doing so will give you an absolute grasp of the difference in complexity between these two problems.

So, let's assume you have made your best attempt at solving ALL3SETS and ALLSUBSETS. There are of course many ways of computing solutions to these problems. One possible approach for each is given in figure 11.4. Hopefully these programs are reasonably self-explanatory. Subsets are modeled using Python lists (e.g., `[1, 2, 3]`); collections of subsets are modeled using lists of lists (e.g., `[ [1], [1, 2], [2, 3] ]`). Note that Python does have a built-in `set` data structure, but we avoid it here for technical reasons. The utility function `formatSetOfSets()`—used in both `return` statements—takes one of these lists of lists as input, and formats it with curly braces (e.g., “`{1} {1, 2} {2, 3}`”) as required in the problem statements.

Now let's think about running times. Let  $k$  be the number of distinct integers in the input, and note that  $k$  is certainly no larger than  $n$ , the length of the input. Program `all3Sets.py` uses a triply nested loop running over at most  $k$  elements each, so it uses  $O(k^3)$  computational steps. (And note that this agrees with figure 10.3 on page 204, which tells us that the number of subsets with size  $s$  is in  $O(k^s)$ .) But we need to measure the running time in terms of  $n$ , the length of

**PROBLEM ALL3SETS**

- **Input:** A list of distinct integers in decimal notation separated by whitespace. Example: “2 5 8 9”.
- **Solution:** A list of all 3-element subsets (i.e., 3-sets) that can be formed from the input. The elements of each set are separated by commas, and the sets are surrounded by braces. Whitespace may be included wherever convenient. Example: On input “2 5 8 9”, a solution is

```
{2,5,8} {2,5,9} {2,8,9} {5,8,9}
```

**PROBLEM ALLSUBSETS**

- **Input:** A list of distinct integers in decimal notation separated by whitespace. Example: “2 5 8 9”.
- **Solution:** A list of all subsets that can be formed from the input. The elements of each set are separated by commas, and the sets are surrounded by braces. Whitespace may be included wherever convenient. Example: On input “2 5 8 9”, a solution is

```
{} {2} {5} {8} {9} {2,5} {2,8} {2,9} {5,8} {5,9}
{8,9} {2,5,8} {2,5,9} {2,8,9} {5,8,9} {2,5,8,9}
```

**Figure 11.3:** Description of the computational problems ALL3SETS and ALLSUBSETS.

the input. Clearly,  $k \leq n$ , so the complexity of `all3Sets.py` is in  $O(n^3)$ . Hence, ALL3SETS is in Poly.

To analyze `allSubsets.py`, we first need to recall from figure 10.3 (page 204) that a set of  $k$  elements has  $2^k$  subsets. Hence, this program produces a list of at most  $2^n$  subsets, each of size at most  $n$ . Based on this insight, it's not hard to convince yourself that `allSubsets.py` is in  $O(n2^n)$ , proving that ALLSUBSETS is in Expo. That puts an asymptotic upper bound on the running time. Can we also obtain a lower bound to show that ALLSUBSETS is not in Poly? With a little more work, we can. The length of the output is at least  $2^k$ , since that is the number of subsets in the output. We won't give details here, but it can be shown that it's always possible to choose  $k \geq n/(3 \log_{10} n)$ . So the length of the output is at least  $2^{n/(3 \log_{10} n)}$ , which grows faster than any polynomial. This proves that ALLSUBSETS is not in Poly. Note that this also shows Poly is a proper subset of Expo. Thus, we can finally conclude that all of the inclusions in figure 11.2 are proper.

## Traveling salespeople and shortest paths

We now turn to one of the most celebrated problems in computer science: the *traveling salesperson problem*, or TSP for short. The motivation for the problem is that a salesperson is given a list of cities together with distances between some

```

1  def all3Sets(inString):
2      # the elements from which we can construct subsets
3      elems = inString.split()
4      # start with an empty list of 3-sets
5      threeSets = [ ]
6      # append each 3-set to the list threeSets
7      for i in range(0, len(elems)):
8          for j in range(i+1, len(elems)):
9              for k in range(j+1, len(elems)):
10                  this3Set = [ elems[i], elems[j], elems[k] ]
11                  threeSets.append(this3Set)
12
13  return utils.formatSetOfSets(threeSets)

```

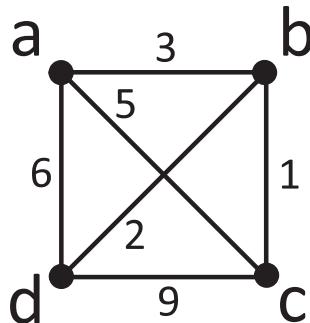
```

1  def allSubsets(inString):
2      # the elements from which we can construct subsets
3      elems = inString.split()
4      # start with an empty list of subsets
5      theSubsets = [ ]
6      # add the empty set “[ ]” to the list of subsets
7      theSubsets.append([ ] )
8      # For each element of the input, append copies of all
9      # previously computed subsets, but with the additional new element
10     # included.
11     for element in elems:
12         newSets = [ ]
13         for thisSet in theSubsets:
14             # create a new subset that includes the current element,
15             # and append it to the list of subsets
16             newSets.append(thisSet + [element])
17             # Update the master list of subsets with all the newly created
18             # subsets. This doubles the number of subsets in theSubsets.
19             theSubsets.extend(newSets)
20
21  return utils.formatSetOfSets(theSubsets)

```

**Figure 11.4:** The Python programs `all3Sets.py` (top) and `allSubsets.py` (bottom).

or all of them. The salesperson must visit each city exactly once, starting and ending in the same city, and using the shortest possible route. Translating this into more standard terminology, the input is an undirected, weighted graph like the one in figure 11.5. The nodes of the graph represent cities, and edge weights represent distances between cities. Recall from page 47 that a *Hamilton cycle* of a graph is a cycle that includes each node exactly once, except for the start node, which is visited again at the end. (The name comes from the 19th-century Irish mathematician William Hamilton.) So, the TSP asks us to find a Hamilton cycle that is as short as possible. For example, the graph in figure 11.5 has several different Hamilton cycles. These include  $a \rightarrow b \rightarrow c \rightarrow d$  ( $\text{length} = 3 + 1 + 9 + 6 = 19$ ) and  $a \rightarrow c \rightarrow b \rightarrow d$  ( $\text{length} = 5 + 1 + 2 + 6 = 14$ ).



**Figure 11.5:** Example of an undirected, weighted graph used as input to the traveling salesperson problem (TSP).

It's not hard to convince yourself that of all the Hamilton cycles,  $a \rightarrow c \rightarrow b \rightarrow d$  is the shortest, and is therefore a TSP solution for this graph.

A formal description of the TSP is given in figure 11.6. Any reasonable input and output formats could be used, but we will stick with the conventions first suggested in figure 4.3 (page 48). In particular, a solution is just an ordered list of the nodes separated by commas. For the example of figure 11.5, one solution is therefore “ $a, c, b, d$ ”. But note that this same cycle can be written in different ways (e.g., “ $c, b, d, a$ ” and “ $d, b, c, a$ ”), and some graphs will have distinct Hamilton cycles that tie for the shortest. That's why we refer to *a* shortest Hamilton cycle rather than *the* shortest Hamilton cycle.

A slight variant of TSP is TSPPATH, where we ask to find a Hamilton *path* rather than a cycle. The start and ending points of the path are given as part of the problem, and a solution is a shortest path from the source to the destination that passes through every other node exactly once. See the middle panel of figure 11.6 for a formal statement and example.

Finally, recall the problem SHORTESTPATH, defined formally in the bottom panel of figure 11.6. The input is the same as TSPPATH, but now we want a shortest path from the source to the destination, without requiring the path to go through all the other nodes. (Note that this version of SHORTESTPATH allows the input to be a weighted graph, which is a generalization of the unweighted SHORTESTPATH problem described on page 55. We use the same name for both problems and rely on context to distinguish between them.)

What kind of algorithms are available to solve SHORTESTPATH and TSPPATH? SHORTESTPATH can be solved (at least in the case of positive weights) by a classic technique called Dijkstra's algorithm, which is in  $O(n^2)$ . So SHORTESTPATH is clearly a member of Poly. On the other hand, no efficient algorithm is known for TSPPATH. One obvious approach is to explicitly enumerate all possible Hamilton paths from the source to the destination. From figure 10.3 (page 204), we know that there are  $O(n!)$  possible Hamilton paths. And from list (★) on page 199, we know  $n! \in O(2^{n^2})$ . Hence, TSPPATH is in Expo. As usual, it's a good exercise to implement programs that solve these problems yourself, but the book resources do provide them too: see `shortestPath.py` and `tspPath.py`.

### PROBLEM TSP

- **Input:** An undirected, weighted graph  $G$ . Example: The graph in figure 11.5 can be represented as “ $a,b,3\ a,c,5\ a,d,6\ b,c,1\ b,d,2\ c,d,9$ ”.
- **Solution:** A shortest Hamilton cycle of  $G$ , or “no” if none exists. Example: “ $a,c,b,d$ ” is a solution to the above input.

### PROBLEM TSPPATH

- **Input:** An undirected, weighted graph  $G$  and two of  $G$ ’s nodes  $v, w$  (the *source* and the *destination*).  $G, v$ , and  $w$  are separated by semicolons and optional whitespace. Example based on figure 11.5: “ $a,b,3\ a,c,5\ a,d,6\ b,c,1\ b,d,2\ c,d,9\ ;\ a\ ;\ c$ ”.
- **Solution:** A shortest Hamilton path starting at  $v$  and ending at  $w$ , or “no” if none exists. Example: “ $a,d,b,c$ ” is a solution to the above input.

### PROBLEM SHORTESTPATH

- **Input:** An undirected, weighted graph  $G$  and two of  $G$ ’s nodes  $v, w$  (the *source* and the *destination*).  $G, v$ , and  $w$  are separated by semicolons and optional whitespace. Usually, we also assume the weights are positive. Example based on figure 11.5: “ $a,b,3\ a,c,5\ a,d,6\ b,c,1\ b,d,2\ c,d,9\ ;\ a\ ;\ c$ ”.
- **Solution:** A shortest path starting at  $v$  and ending at  $w$ , or “no” if none exists. Example: “ $a,b,c$ ” is a solution to the above input.

**Figure 11.6:** Description of the computational problems TSP, TSPPATH, and SHORTESTPATH.

## Multiplying and factoring

As our third and final pair of related problems, let’s consider the tasks of multiplying numbers and factoring numbers. Both have already been defined formally as computational problems: MULTIPLY on page 58, and FACTOR on page 215. For programs solving these problems see `multiply.py` (page 212) and `factor.py` (page 216). We’ve already discussed the complexity of these problems too. MULTIPLY can be solved in polynomial time (specifically,  $O(n^2)$ —see page 213). In contrast, the best known algorithms for FACTOR require superpolynomial time (see section 10.4, page 215). Note that we can think of these problems as inverses of each other: MULTIPLY inputs two numbers  $M_1, M_2$  and produces their product  $M = M_1 M_2$ ; whereas FACTOR inputs the product  $M$  and outputs one of  $M_1$  or  $M_2$  or possibly another factor.

## Back to the boundary between Poly and Expo

Figure 11.7 summarizes the three pairs of problems we are using to get an intuitive feeling for the boundary between Poly and Expo. In each pair, one of

<b>Obviously in Poly</b> (and also in Expo)	<b>Obviously in Expo</b>
<p><b>ALL3SETS:</b> Input is a list of distinct numbers; solution is a list of all possible 3-sets (i.e., sets containing three of the input numbers). Complexity: <math>O(n^3)</math>.</p>	<p><b>ALLSUBSETS:</b> Input is a list of distinct numbers; solution is a list of all possible subsets of the input numbers. Complexity: <math>O(n^{2^n})</math> and requires superpolynomial time (at least <math>2^{O(n/\log n)}</math> steps); <b>therefore outside Poly.</b></p>
<p><b>SHORTESTPATH:</b> Input is a weighted graph, source node, and destination node; solution is a shortest path from source to destination. Complexity: <math>O(n^2)</math>.</p>	<p><b>TSPPATH:</b> Input is a weighted graph, source node, and destination node; solution is a shortest path from source to destination that visits each node exactly once. Complexity: <math>O(n!)</math> upper bound, but <b>not known to be outside Poly.</b></p>
<p><b>MULTIPLY:</b> Input is two integers; solution is their product. Complexity: <math>O(n^2)</math>.</p>	<p><b>FACTOR:</b> Input is an integer; solution is a nontrivial factor. Complexity: <math>O(n^2 10^n)</math> upper bound, but <b>not known to be outside Poly.</b></p>

Figure 11.7: Some examples of Poly and Expo problems.

the problems is obviously in Poly and the other is obviously in Expo. (Remember, however, that all Poly problems are also in Expo.) The six problems in figure 11.7 illustrate several points about the boundary between Poly and Expo:

1. **Problems that seem similar can have very different complexities.** In each row of figure 11.7, it doesn't take much of a tweak to send us from a Poly problem in the left column to an apparently superpolynomial Expo problem in the right column. Specifically, in the first row we ask for *all* subsets rather than just 3-sets; in the second row we ask for a shortest path through *all* nodes rather than a shortest path through *any* nodes; and in the third row we ask to break a *product into factors* rather than combine *factors into a product*. There are several other well-known examples of this phenomenon of small tweaks to easy problems producing hard problems. Some of these problems won't be defined until later, and others aren't covered in this book at all, but we mention them here for completeness: SHORTESTPATH is easy, but LONGESTPATH is hard; MINCUT (page 310) is easy, but MAXCUT (page 359) is hard; EULERCYCLE (which seeks a cycle traversing every *edge* in a graph) is easy, but HAMILTONCYCLE (page 277) is hard.
2. **Some Expo problems are easily proved to be outside Poly.** ALLSUBSETS provides an example of this: as already noted above, the solution has

### PROBLEM FACTORINRANGE

- **Input:** An integer  $M$ , and two additional integers  $a, b$ . The integers are in decimal notation separated by whitespace. Example: “55 10 20”.
- **Solution:** A nontrivial factor of  $M$  in the range  $[a, b]$  inclusive, if one exists, and “no” otherwise. Examples: “55 10 20” is a positive instance with solution “11”, but input “55 15 20” is a negative instance.

**Figure 11.8: Description of the computational problem FACTORINRANGE.** The problem FACTORINRANGE and its decision variant are both believed to require superpolynomial time. This contrasts with FACTOR, which appears to be strictly harder than its polynomial-time decision variant.

superpolynomial length, so any program that solves the problem requires superpolynomial time just to print (or return) the solution. Hence, ALL-SUBSETS lies outside Poly.

3. **Many problems seem to require superpolynomial time, but haven’t been proved to require superpolynomial time.** Our two examples of this are FACTOR and TSPPATH. The provided programs `factor.py` and `tspPath.py` require exponential time. But it is one of the most remarkable facts of computer science that no one has ever *proved* that these two problems require superpolynomial time. And it turns out that there are many, many more problems in this category, including all of the NP-complete problems we will meet in chapter 14.

### Primality testing is in Poly

This is a good time to detour briefly into another problem closely related to FACTOR: the decision problem ISPRIME, which determines whether an integer  $M$  is prime. The famous *AKS algorithm* solves ISPRIME in polynomial time. The AKS algorithm was discovered in 2002 and formally published by Agrawal, Kayal, and Saxena in 2004. We won’t study any details of the algorithm, but it’s important to know of its existence. Recall that an integer is *composite* if it has nontrivial factors and *prime* otherwise. So the decision problem ISCOMPOSITE is the negation of ISPRIME, and it is also solved efficiently by the AKS algorithm.

Note that ISCOMPOSITE is *precisely* the decision version of FACTOR: ISCOMPOSITE asks the question, does  $M$  have a nontrivial factor? This means the same as the question, is  $M$  a positive instance of FACTOR? And yet, as discussed above, the best known algorithm for FACTOR is superpolynomial. The strange, unusual, and important conclusion here is that FACTOR seems to be strictly harder than its decision version!

To work around this anomaly, complexity theorists often consider a modified version of factoring that examines only a range of possible factor values. The problem FACTORINRANGE is defined formally in figure 11.8. The best known algorithm for FACTORINRANGE is superpolynomial, and this is also true of

its decision variant. (Note the contrast with FACTOR, and see exercises 11.9 and 10.13 for more details on this contrast.)

To summarize, we know how to efficiently *detect the existence* of nontrivial factors, but not how to *compute their values*, or even *detect them in a given range*. This is one of the most surprising situations in complexity theory, and it has important applications in cryptography (see page 216).

## 11.4 Poly AND Expo DON'T CARE ABOUT THE COMPUTATIONAL MODEL

We know from figure 10.15 (page 218) that switching between any two classical computational models (e.g., from a real computer to a standard Turing machine) requires, at worst, a polynomial slowdown. This has a remarkable consequence: it turns out that the definitions of Poly and Expo do not depend on which computational model is used (as long as we exclude quantum computers).

For example, suppose we have an algorithm that runs in time  $p(n)$  on a real computer, for some polynomial  $p$ . And suppose that a standard Turing machine requires  $q(t)$  steps to simulate  $t$  instructions on this real computer, for some other polynomial  $q$ . By substituting  $t = p(n)$ , we see that the time required for the standard Turing machine to execute the algorithm is  $q(p(n))$ . But the composition of two polynomials produces another polynomial (see claim 10.1, page 203). So  $q(p(n)) = r(n)$  for some polynomial  $r$ . Obviously, the same argument applies for switching between any two classical computational models. So we've proved that the definition of Poly doesn't depend on which classical model is used.

The same argument works for Expo. For example, an  $O(2^{p(n)})$  algorithm on a real computer is at worst  $O(2^{r(n)})$  on a standard Turing machine, which stays within the definition of Expo. Thus, we have proved the following claim.

**Claim 11.4.** The complexity classes Poly and Expo remain unchanged when defined with respect to any of the following underlying computational models: standard Turing machines, multi-tape Turing machines, random-access Turing machines, real computers (with unlimited memory), and Python programs.

## 11.5 HALTEX: A DECISION PROBLEM IN Expo BUT NOT Poly

As mentioned above, there are many problems like TSP, TSPPATH, and FACTOR that *seem* to require superpolynomial time, but have not been *proved* to require superpolynomial time. This leads to the obvious question: What kind of problems have been proved to lie outside Poly but inside Expo? We've already seen one example: ALLSUBSETS (figure 11.3, page 232). But this is an unsatisfying example: since its output has superpolynomial length, there's no possibility of a clever, polynomial-time algorithm solving ALLSUBSETS. What happens if we restrict our attention to problems with *short* outputs? Can we find any problems with short outputs that are still provably outside Poly but inside Expo? As we will see in this section, the answer is yes.

Our first example of a problem with short solutions that nevertheless requires exponential time is a decision problem called HALTSINEXPTIME, or HALTEX for

**PROBLEM HALTSINEXPTIME (HALTEX)**

- **Input:** A program  $P$  and a string  $I$ .
- **Solution:** “yes” if  $P$ , on input  $I$ , executes fewer than  $2^n$  computational steps before halting; and “no” otherwise. Here,  $n = |I|$  denotes the length of  $I$ —it does *not* include the length of  $P$ .

**Figure 11.9: Description of the computational problem HALTEX.** The program  $P$  could be a Turing machine or a Python program.

short. HALTEX is described formally in figure 11.9. Informally, HALTEX analyzes its input  $(P, I)$  and decides whether program  $P$  halts within exponential time on input  $I$ . More specifically, the solution is “yes” if  $P$  halts before executing  $2^n$  computational steps, where  $n$  is the length of  $I$ . The program  $P$  can be a Turing machine or a Python program. As discussed in section 10.3, the running time of Python programs is measured in CPU instructions and the running time of Turing machines is measured in transitions or “steps.”

To get an initial, intuitive feel for HALTEX, let’s examine some specific instances of the problem. First, we consider analyzing  $P = \text{rf}('containsGAGA.py')$  with the 1000-character input string  $I = "TTTTTT...T"$ . Direct experimentation shows that, on my computer, containsGAGA.py takes only a few dozen milliseconds to run on this input:

```
>>> from time import time
>>> start = time(); containsGAGA(rf('1000Ts.txt'));
       duration = time()-start; print(duration)
'no'
0.06240081787109375
```

The 0.062... number represents the number of seconds required to run the program in this case. This is a very approximate and amateurish way of obtaining the running time of the program, but we don’t need any better precision for our purposes. Let’s just observe that the CPU time required for the program is undoubtedly less than 1 second. My 2.67 GHz CPU executes at most 3 billion instructions per second. Therefore,  $3 \times 10^9$  is an upper bound on the number of computational steps. For this input  $n = 1000$ , so we are interested in whether the program completed in less than  $2^{1000} \approx 10^{301}$  steps. Clearly,  $3 \times 10^9 < 10^{301}$ , so we conclude this is a positive instance of the problem.

Next, consider an instance of HALTEX with  $P = \text{rf}('factor.py')$  and input string  $I = "1129187"$ . Note that 1,129,187 is a seven-digit prime number. So  $n = 7$ , and the main loop of factor.py (figure 10.14, page 216) will be executed over one million times. Hence, the program executes at least 1 million CPU instructions (this is a gross underestimate, but good enough for our purposes). Because  $n = 7$ , we are interested in whether the number of instructions is less than  $2^7 = 128$ . In this case, the answer is obviously “no”.

The above techniques work fine for some instances of HALTEX. But can we write a program that solves the problem in general? The answer is yes. We can solve HALTEX with the following strategy: Use a simulator to simulate  $P$  with

```

# We import the TuringMachine class for fine-grained
# control of Turing machine simulation.
from turingMachine import TuringMachine
def haltExTuring(progString, inString):
    # construct the Turing machine simulator
    sim = TuringMachine(progString, inString)
    # simulate for at most  $2^n - 1$  steps
    for i in range(2**len(inString)-1):
        sim.step()
        if sim.halted:
            return 'yes'
    return 'no'

```

**Figure 11.10:** The Python program `haltExTuring.py`, which solves the problem HALTEX.

input  $I$ , and allow the simulator to run for at most  $2^n - 1$  simulated transitions or CPU instructions. If  $P$  has halted by this time, output “yes”; otherwise output “no”.

Figure 11.10 shows an implementation of this strategy using the Turing machine simulator `turingMachine.py` provided with the book materials. Line 8 uses the Python exponentiation operator “`**`” to represent  $2^n$ , so it’s clear that `haltExTuring.py` does indeed return “yes” if the simulated program halts in fewer than  $2^n$  steps, and “no” otherwise. Note that this approach could also be made to work with Python programs as input, by substituting a Python simulator for the Turing machine simulator.

Next, we must attend to a technical detail about the definition of  $n$ , both in the problem HALTEX and in the program of figure 11.10. For the remainder of this section,  $n$  will continue to denote the length of  $I$ :

$$n = |I| = \text{len}(\text{inString}).$$

This deviates from our usual convention of using  $n$  for the total length of all inputs to a program, because `haltExTuring.py` has two parameters  $P$  and  $I$ . For the rest of this section, we will instead use  $N$  as the total length of these parameters and  $m$  as the length of  $P$ . So we have

$$\begin{aligned} N &= \text{len}(\text{progString}) + \text{len}(\text{inString}) \\ &= \text{len}(P) + \text{len}(I) \\ &= m + n. \end{aligned} \tag{*}$$

Now we can prove our first claim about HALTEX:

**Claim 11.5.** HALTEX is in Expo.

*Sketch proof of the claim.* We sketch a proof for the Turing machine version of HALTEX. Looking at `haltExTuring.py` (figure 11.10), it’s clear we need

```

# Import H, which is assumed to solve HaltsInExpTime in polynomial
2 # time. (In reality, we will prove that H cannot exist.)
from H import H
4 def weirdH(progString):
    if H(progString, progString) == 'yes':
        # deliberately enter infinite loop
        print('Entering infinite loop...')
        utils.loop()
    else:
        # The return value is irrelevant, but the fact that we return
        # right away is important.
10       return 'finished already!'
12

```

**Figure 11.11:** The Python program `weirdH.py`. This program produces contradictory behavior, proving that the program  $H$  can't exist.

running-time estimates for (i) the `TuringMachine()` constructor at line 6, and (ii) the `step()` method at line 9. In both cases, we do not attempt rigorous proofs. However, inspection of the Python source code for `turingMachine.py` demonstrates that both pieces of code run in polynomial time. Specifically, it turns out that (i) the `TuringMachine()` constructor is in  $O(p(N))$  for some polynomial  $p$ , and (ii) the `step()` method is in  $O(q(m))$  for some polynomial  $q$ . The constructor is executed only once, and the `step()` method is executed  $O(2^n)$  times. So the total running time of the program is in  $O(p(N) + q(m)2^n)$ , which satisfies the definition of `Expo`.  $\square$

Now let's proceed to the really interesting thing about HALTEX: we can prove that this problem doesn't just look hard, it really is hard. As the following claim shows, there is no way of solving it in polynomial time.

**Claim 11.6.** HALTEX is not in Poly.

*Overview of the proof.* This proof has close similarities to earlier ones, such as the undecidability of YESONSTRING (figure 3.9, page 38). In earlier proofs, to show that some program  $X$  couldn't exist, we created a new “weird” version of the program, say `weirdX.py`. The weird version is crafted to have self-contradictory behavior: `weirdX.py` first analyzes itself to see if it should have behavior  $B$ , then does the opposite of  $B$ . For example, in figure 3.9, `weirdYesOnString.py` first determines whether it should return “yes”, and then does the opposite. Similarly, one way of showing that the halting problem is undecidable is by crafting `weirdHaltsOnString.py`: it first determines whether it should halt, and if so goes into an infinite loop. In this proof, we take the same approach, but the contradiction involves running times. Specifically, we craft `weirdH.py` (figure 11.11) as follows: It first analyzes itself to find out if its running time should be less than  $2^n$ . If so, it runs forever; if not, it terminates rapidly (within polynomial time). The only tricky part of the proof is this last part: showing that `weirdH.py` does in fact terminate in polynomial time. So, when you read the following proof for the first few times, don't worry too much

about this detail. Concentrate on understanding the high-level self-contradictory behavior of `weirdH.py`, then later focus on the proof of polynomial running time.

*Proof of the claim.* We prove this for the Python program version of HALTEX. Let's assume HALTEX is in Poly, and argue for a contradiction. By our assumption, we can assume the existence of a polynomial-time Python program  $H(P, I)$  that solves HALTEX, so it has the following properties:

1.  $H(P, I)$  returns yes if the Python program  $P$ , with the given input  $I$ , halts before  $2^n$  instructions have been executed. Otherwise  $H(P, I)$  returns no.
2.  $H(P, I)$  runs in polynomial time. Specifically, let  $N = m + n$  be the length of all data passed to  $H$ , as in equation  $(\star)$  on page 240. Then there exists a polynomial  $r(N)$  such that  $H(P, I)$  always halts before executing  $r(N)$  instructions.

Now let's put our program  $H$  to work. We'll call it as a subroutine from another Python program, `weirdH.py`, as shown in figure 11.11. The program `weirdH.py` does three deliberately weird things:

- It takes a single string `progString` as input (line 4), then invokes  $H$  using `progString` as both parameters  $P$  and  $I$  (line 5). In other words, `weirdH.py` asks the question

Does `progString` terminate in fewer than  $2^n$  steps, when given itself as input? (Here,  $n$  is the length of `progString`.)

- If the answer to this question is “yes”, then `weirdH.py` enters an infinite loop (line 8).
- Otherwise the answer to the boxed question above is “no”, and `weirdH.py` returns immediately with the silly return value “finished already!” (line 12).

The next step should be a familiar one: we ask ourselves what happens when `weirdH.py` is run with itself as input. In IDLE,

```
>>> weirdH(rf('weirdH.py'))
```

If you do this for real, beware: the results you get won't be correct, because `weirdH.py` assumes that `H.py` works correctly, and we'll soon prove that this is impossible. Instead, let's work out what should happen if `H.py` did work as advertised. Well, there are two possibilities. Either `weirdH.py` halts in fewer than  $2^n$  steps, or it runs for at least  $2^n$  steps. We will examine these two possibilities after defining them more carefully. Note that in this context,  $n$  is the length of the program `weirdH.py`. This file happens to be 926 ASCII characters long, so  $n = 926$  for the program shown in figure 11.11. But we will also consider the possibility of adding an arbitrary number of space characters to the end of `weirdH.py`, in order to make the program longer without changing its behavior.

Specifically, for any  $n \geq 926$ , let  $\text{weirdH}_n$  be a version of  $\text{weirdH.py}$  created by appending spaces until the program is  $n$  characters long.

Let  $T(n)$  be the running time of  $\text{weirdH}_n$  when given itself as input. Note that  $T(n)$  is defined only for values of  $n \geq 926$ , but that's not a problem since we will be interested in large values of  $n$ . Given  $n$ , there are two possibilities for  $T(n)$ : either (i)  $T(n) < 2^n$ , or (ii)  $T(n) \geq 2^n$ . Both cases will lead to contradictions, and we examine each case separately.

**Case (i):**  $T(n) < 2^n$ . In this case,  $H$  returns “yes” at line 5, and  $\text{weirdH}_n$  enters an infinite loop. So if  $T(n) < 2^n$ , then  $T(n)$  is infinite, which is a contradiction. In plain English, if  $\text{weirdH}_n.py$  halts in fewer than  $2^n$  instructions, then it runs forever, which is a contradiction.

**Case (ii):**  $T(n) \geq 2^n$ . In this case,  $H$  returns “no” at line 5, and  $\text{weirdH}_n.py$  returns immediately. What is the total number of instructions used by the program in this case? Line 4 and lines 9–12 all run in constant time. Let’s assume those lines use at most  $C$  instructions in total, for some constant  $C > 0$ . The only expensive operation is the call to  $H(\text{progString}, \text{progString})$  at line 5. How many instructions does this use? For this, we use the second property of  $H$  assumed above:  $H$  always terminates within  $r(N)$  instructions, where  $r$  is a fixed polynomial and  $N$  is the length of all parameters to  $H$ . In this case,  $N = 2n$ , because both parameters of  $H(\text{progString}, \text{progString})$  are the program  $\text{weirdH}_n.py$ , which has length  $n$ .

Hence, the total number of instructions used is at most  $C + r(2n)$ . Write  $s(n) = C + r(2n)$ , and note that because  $s(n)$  is formed by adding and composing polynomials,  $s(n)$  is itself a polynomial (see claim 10.1, page 203). That is,  $T(n) \in O(s(n))$  for some polynomial  $s(n)$ . But this contradicts the fact that  $T(n) \geq 2^n$ , which was assumed for case (ii). If the contradiction isn’t immediately obvious, think of it this way:  $2^n$  dominates any polynomial (see list (★), page 199). So the assumption that  $T(n) \geq 2^n$  means that  $T$  dominates any polynomial. But the statement  $T(n) \in O(s(n))$  means that  $T$  is bounded by some polynomial—a contradiction, for all sufficiently large  $n$ . (Technical note: We needed to obtain a contradiction for a single value of  $n$  only, but we in fact proved a contradiction for all sufficiently large  $n$ .) In plain English, if  $\text{weirdH}_n.py$  has an exponentially long running time, then its running time is bounded by a polynomial, which is a contradiction.

The only two possibilities led to contradictions, so the proof of the claim is complete.  $\square$

## 11.6 OTHER PROBLEMS THAT ARE OUTSIDE Poly

So now we know there are decision problems that can be solved in exponential time, but not in polynomial time. Our only example of this is HALTEX. It’s an interesting problem, but a somewhat artificial one—it’s pretty obvious that computer scientists cooked up this problem just so the proof by contradiction in the previous section could be made to work.

So the question remains: Are there any realistic, practical decidable problems that are known to be unsolvable in polynomial time? The answer is yes, although

we won't look into proofs or even formal definitions here. One easily described example is *generalized chess*. Regular chess takes place on an 8-by-8 board. Given a particular position of all the black and white pieces on the board, we can pose the decision problem "Is White guaranteed to win from this position?" This is a very tough problem, but because it is defined for a board of fixed size, the size of the input is bounded, and it doesn't make sense to investigate the complexity of a problem with bounded input size. So instead, consider generalizing this problem to a version of chess played on an  $n$ -by- $n$  board with  $O(n)$  pieces. As  $n$  increases, the size of the input increases too, and now we can ask about the complexity of the problem. It turns out that there is no polynomial-time algorithm for determining whether White can force a win from a generalized  $n$ -by- $n$  chess position.

## 11.7 UNREASONABLE ENCODINGS OF THE INPUT AFFECT COMPLEXITY

When using a standard Turing machine, what is the time complexity of the computational problem ISODD, in figure 7.1 (page 117)? The machine must examine the last digit of the input  $M$ , which requires at least  $n$  steps just to move the head into position over the last non-blank cell. So the best possible asymptotic bound on the time complexity is  $O(n)$ . But what would happen if we defined this problem in a slightly different way, by specifying that  $M$  would be provided with its digits in *reverse* order? In this case, a standard Turing machine can solve the problem by looking at only the first cell of the tape, so the running time is in  $O(1)$ .

This trivial example demonstrates that the encoding of the input can affect the complexity of a computational problem. To avoid ambiguity, the description of a problem should, in principle, include a description of the input encoding. In practice, however, it turns out that for most problems of practical interest, switching between any reasonable encoding of the input changes a problem's complexity by at most a small polynomial factor such as  $O(n)$  or  $O(n^2)$ . If we are interested only in broad classes such as Poly and Expo, these polynomial factors are irrelevant and hence the input encoding becomes irrelevant too—as long as the encoding is "reasonable."

There is one particular *unreasonable* encoding that sometimes plays a role in complexity theory: *unary* notation for integers. The unary notation for a positive integer  $M$  consists of  $M$  copies of the character "1". For example, the decimal "5" becomes "11111" in unary. The significance of unary is that unary numerals are exponentially longer than their equivalent binary or decimal representations: the length  $n$  of the input is given by  $n = M$  instead of  $n = O(\log M)$ , as in equation (★) on page 211. This difference is large enough to have a major impact on a problem's complexity class.

For example, consider the problem FACTORUNARY, which is identical to FACTOR (figure 10.13, page 215) except that the input integer  $M$  is given in unary. Recall that the naïve algorithm `factor.py` (figure 10.14, page 216) runs in polynomial time as a function of  $M$ . And since  $n = M$  for FACTORUNARY, we immediately deduce that FACTORUNARY is in Poly—in

stark contrast to the fact that no polynomial-time algorithm is known for FACTOR.

At first glance, this conclusion appears to defy common sense, since we can apparently solve FACTORUNARY more efficiently than FACTOR. It is true that we know how to solve FACTORUNARY, but not FACTOR, in polynomial time. But the absolute amount of time to solve both problems is similar. It's just that we chose an unreasonably inefficient method of encoding the input to FACTORUNARY, so that the running time when measured as a function of this absurdly large input length turns out to be small. If this still seems strange to you, run some experiments on `factorUnary.py` and `factor.py`, which are both provided with the book materials (and see exercise 11.14).

Because of its inefficiency, unary is never used in practice. But unary notation is sometimes useful in theoretical proofs, which is why you should be aware of it.

## 11.8 WHY STUDY Poly, REALLY?

In this chapter, we've established a basic understanding of the most fundamental distinction in complexity theory: the distinction between problems that are in Poly and those that are outside Poly. Why are complexity theorists obsessed with this distinction?

One common answer is to say that Poly problems are typically “tractable” (i.e., solvable in practice) and non-Poly problems are typically “intractable.” Although there is a certain amount of truth to this answer, it is potentially misleading for at least three reasons.

The first reason is that some published polynomial-time algorithms are not practical. For example, at the time this book was written (late 2010s), the best known variant of the AKS primality-testing algorithm discussed on page 237 requires  $cn^6$  operations (for some constant  $c$ ). This is polynomial, but is wildly impractical for modern cryptographic applications, where the integers involved can be thousands of digits in length. That’s why real-world cryptography implementations use a probabilistic primality test, which happens to be vastly more efficient than AKS. For a more extreme example, consider a 2013 paper by Austrin, Benabbas, and Georgiou, which describes a polynomial-time algorithm for approximating a problem known as *max bisection*. This algorithm is estimated to require roughly  $cn^{10^{100}}$  operations, which is polynomial but useless in practice for any  $n > 1$ . Computer scientists discover improvements to algorithms with surprising rapidity. So by the time you read this, the particular algorithms just mentioned may have been significantly improved. But it seems plausible there will always be plenty of Poly problems whose best known algorithms require too much time to be useful in practice—that is, Poly problems that are “intractable.”

A second reason that Poly problems may not be tractable has already been mentioned on page 223: if the typical input to a problem consists of millions or billions of symbols, even an  $O(n^2)$  algorithm may be far too slow. Hence, quadratic-time algorithms are unacceptable for applications that must process, say, an entire human genome.

Finally, the third reason for not equating Poly with “tractable” is that superpolynomial-time algorithms can be useful in practice. In principle, this could include algorithms that run in quasipolynomial time (see page 202) and even—for appropriately modest values of  $n$ —some outright exponential algorithms with running times like  $2^{n^{1/k}}$ . Indeed, in writing this book, a simple double-exponential-time algorithm was used to verify that all our examples of Turing machines and finite automata produce correct outputs on all strings up to a given maximum length  $M$ .

For these reasons, despite the strong correlation between tractability and membership in Poly, it is certainly too simplistic to equate “Poly” with “tractable.” The connection to tractability is just one among several reasons for the importance of Poly as a complexity class.

So what are the other reasons that complexity theorists focus on Poly? One was mentioned earlier (claim 11.4, page 238): Poly is invariant to the computational model. Another reason is that Poly leads naturally to the extraordinarily important theory of NP-completeness, which is addressed in chapter 14. But perhaps the most fundamental property of Poly is that it defines a particular computational methodology. Polynomial-time algorithms employ loops and nested loops, iterating over sets of possibilities whose size may increase additively but not geometrically. In contrast, algorithms that require exponential time usually explore sets of possibilities whose size increases geometrically, such as the nodes on successive levels of a tree or the elements in a list that occasionally doubles in size. In the words of Christopher Moore and Stephan Mertens in their magisterial book, *The Nature of Computation*, Poly “is not so much about tractability as it is about mathematical insight into a problem’s structure.”

## EXERCISES

**11.1** A *polylogarithmic function* is a function of the form  $p(\log n)$ , for some polynomial  $p$ . For example,  $6(\log n)^3 + 3 \log n + 7$  is a polylogarithmic function. Define the complexity class PolyLogTime to be the set of computational problems that are solved by a polylogarithmic time program. (Using the soft- $O$  notation of page 203, this corresponds to  $\tilde{O}(1)$  programs.)

- (a) Is PolyLogTime a subset of Poly? Give a brief proof of your answer.
- (b) Is PolyLogTime a *strict* subset of Poly? Give a brief proof of your answer.

**11.2** Define the complexity class PolyPolyLog to be the set of computational problems that can be solved by a program with time complexity in  $O(p(n)q(\log n))$ , for some polynomials  $p$  and  $q$ . (Using the soft- $O$  notation of page 203, this corresponds to  $\tilde{O}(p(n))$ .) Prove that Poly and PolyPolyLog are the same complexity class.

**11.3** Consider the input string  $I = "1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10"$ . How many sets would be in the solution of the following problem instances?

- (a) ALL3SETS( $I$ )
- (b) ALLSUBSETS( $I$ )

**11.4** Give a solution to each of the following problem instances:

- (a) TSP("a,b,3 b,c,2 c,d,1 d,e,2 e,a,1 b,d,5")
- (b) TSPPATH("a,b,3 b,c,2 c,d,1 d,e,2 e,a,1 b,d,5 ; a c")
- (c) SHORTESTPATH("a,b,3 b,c,2 c,d,1 d,e,2 e,a,1 b,d,5 ; a c")
- (d) TSPPATH("a,b,3 b,c,2 c,d,1 d,e,2 e,a,1 b,d,5 ; a d")

**11.5** Give the entire solution set for each of the following problem instances:

- (a) MULTIPLY("5 10")
- (b) FACTOR("23")
- (c) FACTOR("30")
- (d) ISPRIME("30")
- (e) ISCOMPOSITE("30")
- (f) FACTORINRANGE("300 10 100")

**11.6** Define the problem ALLPERMUTATIONS as follows. The input  $I$  is an ASCII string, and a solution is a list of all possible permutations of  $I$ , separated by newlines. For example, on input "cat", a solution is "cat $\leftarrow$ cta $\leftarrow$ act $\leftarrow$ atc $\leftarrow$ tca $\leftarrow$ tac"; on input "bee", a solution is "bee $\leftarrow$ ebe $\leftarrow$ eeb".

- (a) Write a Python program that solves ALLPERMUTATIONS, assuming all the characters in  $I$  are distinct.
- (b) Improve your program so that it works correctly for any ASCII string  $I$ .
- (c) Give a reasonable estimate of the running time of your program.
- (d) Prove that ALLPERMUTATIONS  $\in$  Expo.
- (e) Prove that ALLPERMUTATIONS  $\notin$  Poly.

**11.7** For each of the following problems, describe our state of knowledge about whether the problem is in Poly and/or Expo. For example, the answer for the problem FACTOR is "FACTOR  $\in$  Expo, and FACTOR is widely believed but not proved to be outside Poly."

- (a) FACTORINRANGE: see figure 11.8, page 237.
- (b) ISCOMPOSITE: see page 237
- (c) FACTORUNDERONEMILLION (FUOM): Input is an integer  $M$ . Solution is a nontrivial factor of  $M$  that is less than 1 million, or "no" if no such factor exists.
- (d) FACTORLESSTHAN1PERCENT (F1PCT): Input is an integer  $M$ , solution is a nontrivial factor of  $M$  that is less than  $M/100$ , or "no" if no such factor exists.
- (e) HALTSINCUBICTIME: Input is a program  $P$  and input  $I$ . Solution is "yes" if  $P$  halts within  $n^3$  steps on input  $I$  (where  $n = |I|$ ), and "no" otherwise.
- (f) HALTSIN10ONTIME: Input is a program  $P$  and input  $I$ . Solution is "yes" if  $P$  halts within  $10^n$  steps on input  $I$  (where  $n = |I|$ ), and "no" otherwise.
- (g) POWER: Input is two integers  $M_1, M_2$ . Solution is  $M_1^{M_2}$ .

**11.8** Consider the generic problem instance FACTORINRANGE( $M a b$ ) for positive integers  $M, a, b$ . For each of the following special cases, state whether

it is known how to solve the instance in polynomial time, and explain your reasoning:

- (a)  $a = 0$
- (b)  $b = M$
- (c)  $a = 0, b = M$
- (d)  $a = 0, b = \text{floor}(\sqrt{M})$
- (e)  $a = 0, b = \text{floor}(M/2)$
- (f)  $a = \text{floor}(M/2), b = M$

**11.9** Let  $\text{FIRD}$  denote the decision variant of  $\text{FACTORINRANGE}$ , so  $\text{FIRD}(M, a, b) = \text{"yes"}$  if and only if  $M$  has a nontrivial factor in the range  $[a, b]$  inclusive.

- (a) Write a program `fird.py` that solves  $\text{FIRD}$ . (Hint: No polynomial-time algorithm is known for this, so don't worry about efficiency.)
- (b) Suppose, hypothetically, that `fird.py` did actually run in polynomial time. Under this hypothetical assumption, write a program `factorInRange.py` that solves  $\text{FACTORINRANGE}$  in polynomial time. (Hint: Invoke `fird.py` and use binary search.)
- (c) What happens if we follow the same series of steps for  $\text{FACTOR}$ ? Specifically, if we are given a hypothetical polynomial-time program `factorD.py` that solves the decision variant of  $\text{FACTOR}$ , can we use the same technique to obtain a polynomial-time program for  $\text{FACTOR}$  as a nondecision problem?

**11.10** Consider the decision problem  $\text{HALTSINSOMEPOLY}$  ( $\text{HISP}$ ), defined as follows. The input is a program  $P$ , and the solution is "yes" if and only if there exists some polynomial  $q(n)$  such that, for every  $n$ ,  $P$  halts after at most  $q(n)$  steps on all inputs of length  $\leq n$ .

- (a) State, with proof, which of the following statements are true: (i)  $\text{HISP}$  is undecidable, (ii)  $\text{HISP} \in \text{Expo}$ , (iii)  $\text{HISP} \in \text{Poly}$ .
- (b) Would your answer change if we restrict the domain of  $\text{HISP}$ , so that the input  $P$  is guaranteed to halt on all inputs? (Note: This is an example of a *promise problem*, in which the input is promised to have a certain property, which may itself be undecidable.)

**11.11** Recall from page 202 that a quasipolynomial function is of the form  $2^{p(\log n)}$  for some polynomial  $p$ . Note that quasipolynomial functions are superpolynomial and subexponential. Define the complexity class  $\text{QuasiPoly}$  to be the set of all problems solvable by a program with time complexity in  $O(2^{p(\log n)})$  for some polynomial  $p$ . Give an example of a decision problem in  $\text{QuasiPoly}$  that is not in  $\text{Poly}$ , and sketch a proof of your claim.

**11.12** Give an example of a decidable decision problem that is outside  $\text{Expo}$ . Explain informally why your example is outside  $\text{Expo}$ , without giving a rigorous proof.

**11.13** Section 11.6 mentioned that solving generalized chess requires exponential time. Do some brief research to find out some other generalized games that require exponential time to solve, and briefly describe them. Can you

think of any suitably generalized games that do not require exponential time to solve?

**11.14** Time the execution of the commands `factor('10000019')` and `factorUnary(10000019*'1')`. Compare the absolute running times and the running times per character of input. Which program has lower complexity? In your opinion, which program is more “efficient”?

# 12



## PolyCheck AND NPoly: HARD PROBLEMS THAT ARE EASY TO VERIFY

It has yet to be proved that more time is required to find mathematical proofs than is required to test them.

— Leonid Levin, *Universal Search Problems* (1973)

The motivation for defining complexity classes Poly and Expo is reasonably clear: problems in Poly are usually “easy” and problems in Expo but outside Poly are usually “hard.” In contrast, there is no such obvious motivation for the complexity class PolyCheck, which is the main topic of this chapter. The true motivation behind PolyCheck becomes clear only after we have understood the theory of NP-completeness, in chapter 14. For now, it’s best to take it on faith that PolyCheck is interesting because it contains many of the genuinely useful yet apparently intractable computational problems that scientists would like to solve. This includes many problems in engineering, biology, chemistry, physics, and mathematics.

Let’s start with an informal definition: PolyCheck is the class of problems for which (a) it might take a long time to *compute* a solution; but (b) once you have a solution, it’s easy for a separate computer program to *check* (or *verify*) that solution quickly.

The computational problem FACTOR (figure 10.13, page 215) provides a concrete example of this. As far as anyone knows, there’s no efficient way of finding the factors of a large number  $M$ ; the best known existing algorithms take superpolynomial time. However, once you are given a nontrivial factor  $m$ , there’s an extremely efficient way of checking that the answer is correct: simply divide  $M$  by  $m$  and check that the remainder is zero. Hence, at least according to our initial informal definition, FACTOR is in PolyCheck.

### 12.1 VERIFIERS

A *verifier* is a program that checks solutions to computational problems. Figure 12.1 gives an example of a verifier for the problem FACTOR. This verifier

```

1  def verifyFactor(I, S, H):
2      if S == 'no': return 'unsure'
3      M = int(I); m = int(S)
4      if m>=2 and m<M and M % m == 0:
5          # m is a nontrivial factor of M
6          return 'correct'
7      else:
8          # m is not a nontrivial factor of M
9          return 'unsure'

```

**Figure 12.1:** The Python program `verifyFactor.py`.

takes three string parameters: an instance  $I$  of the problem, a proposed solution  $S$ , and a hint  $H$ . The use of the hint will be explained later—for now, please ignore  $H$ , since it is not used by this verifier.

The execution of `verifyFactor.py` is easy to understand. The instance will be a string representation of an integer, such as  $I = "1400"$ . And the proposed solution will be either “no” or a string representation of a factor to be verified, such as  $S = "200"$ . The program converts its string inputs into integers  $M$  and  $m$  at line 3. It then checks to see whether  $m$  is a nontrivial factor of  $M$ . If  $m$  is a nontrivial factor, then we know  $S$  is a correct solution for the input  $I$ , and the verifier outputs the string “correct”. So for our current example ( $I = "1400"$  and  $S = "200"$ ), the verifier’s output is “correct”.

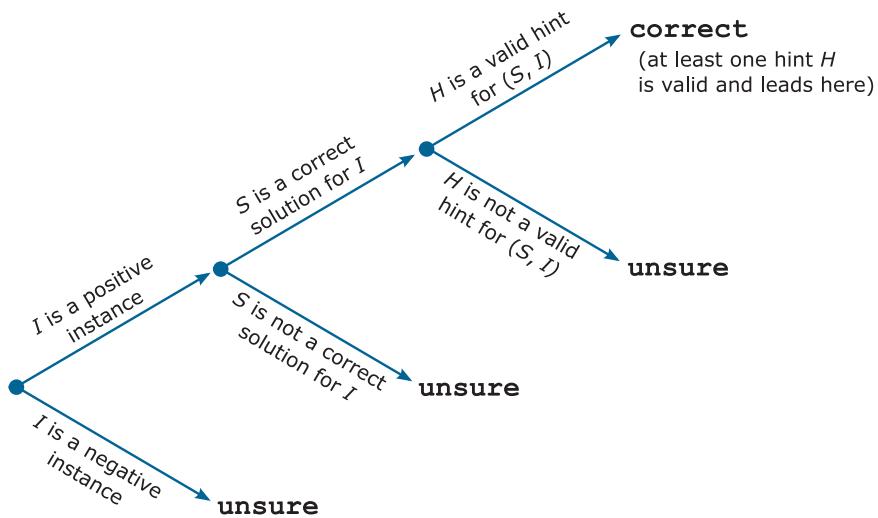
But note carefully what happens if  $m$  is not a nontrivial factor of  $M$ . In this case, you would probably expect the output to be “incorrect”, since it’s clear that  $m$  is an incorrect solution. However, because of a technicality explained later (page 253), verifiers don’t operate in this way. Instead, if the solution can’t be verified as correct, a verifier returns “unsure”. For example, if the parameters had been  $I = "1400"$  and  $S = "500"$ , the verifier’s output would be “unsure”.

There is some potential for confusion about our terminology for solutions in this chapter. The word “solution” was defined on page 54: given a problem  $F$ , a *solution* for the instance  $I$  is any element of the solution set  $F(I)$ . So, by definition, any solution is a “correct solution.” But the *proposed* solution  $S$  given as input to a verifier may not be an actual (“correct”) solution. So to prevent confusion, we will sometimes refer to solutions as “correct solutions.”

Now we are ready for the formal definition of a verifier. This is a complex definition. The first time you read it, ignore the hint  $H$ . Then study the TSPD example after the definition, which illustrates the use of hints. Finally, return to this definition and read it carefully in conjunction with the explanatory diagram 12.2.

**Definition of a verifier.** Let  $F$  be a computational problem. A *verifier* for  $F$  is a program  $V$  with the following properties:

- $V$  receives three string parameters: an instance  $I$ , a proposed solution  $S$ , and a hint  $H$ .
- $V$  halts on all inputs, returning either “correct” or “unsure”.



**Figure 12.2:** The output of a verifier program depends on three strings: instance  $I$ , proposed solution  $S$ , and hint  $H$ .

- **Every positive instance can be verified.** If  $I$  is a positive instance of  $F$ , then  $V(I, S, H) = \text{"correct"}$  for **some** correct positive solution  $S$  and **some** hint  $H$ .
- **Negative instances can never be verified.** If  $I$  is a negative instance of  $F$ , then  $V(I, S, H) = \text{"unsure"}$  for **all** values of  $S$  and  $H$ .
- **Incorrect proposed solutions can never be verified.** If  $S$  is not a correct solution (i.e.,  $S \notin F(I)$ ), then  $V(I, S, H) = \text{"unsure"}$  for **all**  $H$ .

Figure 12.2 explains this behavior diagrammatically. Note carefully the asymmetry between positive and negative instances. When checking a positive instance, a verifier must return “correct” for at least one correct solution and hint, but it can be “unsure” about other correct solutions and/or other hints. When checking a negative instance, the verifier cannot return “correct” for any solution or hint—it must be “unsure” for all possibilities. (Does this asymmetry remind you of anything? It should! Look back to the definition of the output of a nondeterministic computation on page 152. The similarity of these definitions is not a coincidence.) It’s easy to check that `verifyFactor.py` (figure 12.1) is indeed a verifier for FACTOR, according to the formal definition above.

For our next example, we will use a new problem called TSPD, defined formally in figure 12.3. The “D” in the name here indicates that this is a *decision* variant of TSP, which was originally defined on page 235. We convert TSP from an optimization problem to a decision problem by introducing a threshold  $L$  and declaring the solution to be “yes” if the input graph  $G$  has a Hamilton cycle of length at most  $L$ . (For a refresher on the relationship between optimization, search, and decision problems, look back to section 4.3 and especially figure 4.13.)

The interesting new feature of this example is that our verifier will need to use the hint string  $H$  if we want it to work in a reasonable amount of time. Why? Because TSPD is a decision problem, so the positive solution  $S$  to be verified will

### PROBLEM TSPD

- **Input:** An undirected, weighted graph  $G$  and integer threshold  $L$  separated by a semicolon. Example: “ $a,b,5\ b,c,6\ c,a,3\ ;\ 20$ ”.
- **Solution:** “yes” if  $G$  has a Hamilton cycle of length at most  $L$ , and “no” otherwise. Example: With the above input, the solution is “yes”, because “ $a,b,c$ ” is a Hamilton cycle of length 14, and  $14 \leq 20$ .

**Figure 12.3:** Description of the computational problem TSPD.

```

1   def verifyTspD(I, S, H):
2       if S == 'no': return 'unsure'
3           # extract G,L from I, and convert to correct data types etc.
4           (G,L) = I.split(';')
5           G = Graph(G, directed=False); L = int(L)
6
7           # split the hint string into a list of vertices, which will
8           # form a Hamilton cycle of length at most L, if the hint is correct
9           cycle = Path(H.split(','))
10
11          # verify the hint is a Hamilton cycle, and has length at most L
12          if G.isHamiltonCycle(cycle) and \
13              G.cycleLength(cycle) <= L:
14              return 'correct'
15          else:
16              return 'unsure'
```

**Figure 12.4:** The Python program verifyTspD.py.

simply be “yes”. Theoretically, we could verify this “yes” solution by searching through every potential cycle—but there are exponentially many possibilities, so in practice this approach would take too long. Instead, we need a hint telling us *why* the solution is “yes”. Specifically, the hint should consist of a Hamilton cycle of length at most  $L$ . Once we are given this cycle, we can easily check that it is indeed a Hamilton cycle and that its length is at most  $L$ . That’s exactly what the program verifyTspD.py does, in figure 12.4. To understand this program, consider the example inputs  $I = “a,b,1\ b,c,2\ c,a,3;8”$ ,  $S = “yes”$ ,  $H = “a,b,c”$ . The hint  $H$  is a Hamilton cycle of length 6, and since  $6 \leq 8$ , the program concludes that  $S = “yes”$  is a correct solution and outputs “correct”.

Now that we’ve seen hints in action, review the formal definition of verifier on page 251, matching each bullet point to the relevant part of figure 12.2.

#### Why “unsure”?

Now we can give an explanation that was promised earlier: Why would a verifier output “unsure”, even for an incorrect proposed solution? The answer is that,

```

1  def verifyFactorPolytime(I, S, H):
2      # reject excessively long solutions and hints
3      if len(S) > len(I) or len(H) > len(I):
4          return 'unsure'
5      # the remainder of the program is identical to verifyFactor.py
6      # ...

```

**Figure 12.5:** The Python program `verifyFactorPolytime.py`.

for a positive instance  $I$ , there are two separate reasons that verification can fail—these are the upper two “*unsure*” branches in figure 12.2. The first reason is that the proposed solution actually is incorrect. The second reason is that the hint was wrong. In general, there is no way of distinguishing these two cases, which is why we only ever permit verifiers to output “*correct*” or “*unsure*”, and never “*incorrect*”.

## 12.2 POLYTIME VERIFIERS

We’ll be most interested in verifiers that check instances efficiently. As usual, the informal notion of “efficient” is formalized as “polynomial time,” which we will often abbreviate to *polytime* from now on:

**Definition of a polytime verifier.** Suppose  $V(I, S, H)$  is a verifier for a computational problem  $F$ . We say that  $V$  is a *polynomial-time verifier* (or *polytime verifier*) if the running time of  $V$  is bounded by a polynomial as a function of  $n$ , the length of  $I$ .

It’s crucial to note that the lengths of  $S$  and  $H$  play no role in this definition. We care only about the length  $n$  of the instance  $I$ . Because of this, it turns out that `verifyFactor.py` (page 251) is not a polytime verifier. To see why, consider what happens if we give the program a proposed solution that has length  $2^n$ . Then the conversion from  $S$  to  $m$  at line 3 requires at least  $2^n$  steps, so the running time is not polynomial as a function of  $n$ .

Fortunately, it’s easy to improve our verifier and obtain one that is guaranteed to run in polynomial time. The program `verifyFactorPolytime.py` in figure 12.5 shows how to do this. We simply add a check at the start of the program, rejecting unreasonably long proposed solutions and hints.<sup>1</sup> It’s easy to see that the remaining lines of the program all run in polynomial time, so this new program does satisfy the definition of a polytime verifier for FACTOR. Similarly, the program `verifyTspd.py` is not a polytime verifier as written, but can be converted into one with a small change to reject long solutions and hints (see `verifyTspDPolytime.py` in the book materials).

The remainder of this section covers technical details that will be needed later. On first reading, skip ahead to section 12.3 on page 256.

<sup>1</sup> Technicality: What if `str(len(S))` is itself exponentially long? What we really need here is a special “bounded” version of `len`, say `bLen(S, M)`, that examines at most  $M$  characters of  $S$ .

## Bounding the length of proposed solutions and hints

It will be important for us to know that polytime verifiers aren't affected by excessively long proposed solutions or hints. As the next claim shows, even if  $S$  and/or  $H$  are excessively long, we can ignore all but a small (polynomial-sized) substring at the start of  $S$  and  $H$ .

**Claim 12.1.** Suppose  $V(I, S, H)$  is a polytime verifier for a computational problem  $F$ , and let  $n$  be the length of  $I$ . Then there is some polynomial  $p(n)$  such that the output of  $V$  depends only on the first  $p(n)$  characters of  $S$  and the first  $p(n)$  characters of  $H$ .

*Proof of the claim.* For this proof, it is best to think of  $V$  as a multi-tape transducer Turing machine with the inputs  $S$  and  $H$  provided on separate, special input tapes.<sup>2</sup> By definition,  $V$  runs in polynomial time as a function of  $n$ , so there exists some polynomial  $p(n)$  bounding the number of computational steps that  $V$  executes. This means the heads of the special input tapes move at most  $p(n)$  cells along these tapes before the machine halts. Therefore, at most the first  $p(n)$  characters of  $S$  and the first  $p(n)$  characters of  $H$  can affect the output of the machine.  $\square$

## Verifying negative instances in exponential time

Recall that polytime verifiers output either “correct” or “unsure”, and they only ever output “correct” on positive solutions. So, at first glance, it seems we can't use verifiers to check that instances are negative—wouldn't we always be “unsure” about negative instances? In fact, the opposite is true. We *can* verify negative instances, but only if we are willing to spend an exponentially long time doing so.

Here's how to verify a negative instance  $I$  of length  $n$  using a polytime verifier  $V$ . First, compute the value  $N = p(n)$  in the claim above, which tells us the maximum number of characters we need to consider in the solution and/or hint. Now run the verifier, obtaining  $V(I, S, H)$ , for every possible solution  $S$  and hint  $H$  up to length  $N$ . (When using the 128-symbol ASCII alphabet, there are about  $128^{2N}$  possible solutions and hints, which could be an unimaginably large number. But it is a finite number.) If the verifier returns “unsure” in every case, we can conclude that  $I$  is a negative instance. Why? Because if  $I$  were positive, then  $V$  must return “correct” for some correct solution  $S$  and some value of  $H$ . But we tested every relevant value of  $S$  and  $H$ , and none of them returned “correct”.

## Solving arbitrary instances in exponential time

Incidentally, the algorithm just described can also be used to solve arbitrary instances using a polytime verifier, even without the help of a hint. Given  $I$ , we compute  $N = p(n)$  as before, and then compute  $V(I, S, H)$  for all  $S$  and  $H$  up

<sup>2</sup> This proof also applies directly to random-access Turing machines, since the RAM tape must be initially blank;  $S$  and  $H$  are on “ordinary” tapes whose heads move only one cell per step. See page 92.

to length  $N$ . If any result is “correct”, then  $I$  is positive and the corresponding  $S$ -value is a solution. Otherwise  $I$  is negative and the solution is “no”. Thus, a polytime verifier can always be converted into a “solver” program that solves all instances. But the solver will, in general, require exponential time to run.

## 12.3 THE COMPLEXITY CLASS PolyCheck

Once we have a good understanding of verifiers, it’s easy to give a formal definition of the complexity class PolyCheck.

**Definition of PolyCheck.** The complexity class PolyCheck consists of all computational problems that have polytime verifiers.

Informally, a computational problem  $F$  is in PolyCheck if we can verify (or “check”) its positive instances in polynomial time. FACTOR provides our first example: we have already seen (in figure 12.5) that FACTOR has a polytime verifier, so  $\text{FACTOR} \in \text{PolyCheck}$ . This makes good sense: it’s difficult to *find* a nontrivial factor of a large number, but easy to *check* whether a particular integer is a nontrivial factor of a large number. TSPD is another example of a PolyCheck problem. Again, this is intuitively reasonable because it’s difficult to find a short Hamilton cycle, but easy to check whether a given Hamilton cycle is short enough.

### Some PolyCheck examples: PACKING, SUBSETSUM, and PARTITION

The easiest way to gain a deeper understanding of PolyCheck is via further examples. Figure 12.6 gives details of three more classic PolyCheck problems: PACKING, SUBSETSUM, and PARTITION. Informally, for PACKING, we are given a bunch of packages with weights, and we want to load them onto a delivery truck that needs at least weight  $L$  to make a delivery worthwhile, and can carry at most a total weight  $H$ . If this can’t be done, the output is no. Otherwise the output is a list of which packages can be used to produce a feasible packing of the truck. For SUBSETSUM, the lower threshold  $L$  is omitted and we require the truck to be packed with exactly total weight  $H$ . This explains the name of the problem, since we are looking for a *subset* of the packages that *sums* to  $H$ . For PARTITION, both thresholds are omitted and we require the truck to be packed with exactly half the total weight of the packages. The name of this problem makes sense too, since we are looking for a *partition* of the packages into two sets of equal weight.

We leave it as an exercise to prove that PACKING, SUBSETSUM, and PARTITION are all in PolyCheck. Test your understanding now: write programs that verify each of these problems in polynomial time (see exercise 12.1). Of course, it is easy to see in an informal way that the three packing problems are in PolyCheck. It might be very hard to find a feasible packing. But once we are given an allegedly feasible packing as a proposed solution, we can quickly check whether the packing is in fact feasible, by adding up its weights.

### PROBLEM PACKING

- **Input:** A list of integer *weights* denoted  $w_1, w_2, \dots$ , and two further integer *thresholds* denoted  $L, H$ . The weights  $w_i$  are separated by whitespace. The thresholds  $L$  and  $H$  are separated from each other and from the weights by semicolons. Example: “12 4 6 24 4 16 ; 20 ; 27”. We think of the  $w_i$  as the weights of some objects to be loaded into a delivery truck. We think of  $L$  as the minimum weight that makes it worthwhile for the truck to make a delivery, and  $H$  as the maximum weight that the truck can carry. (So  $L$  stands for Low and  $H$  for High.)
- **Solution:** A *feasible packing* is a subset of the input weights whose total weight  $W$  lies between the low and high thresholds:  $L \leq W \leq H$ . If no feasible packing exists, the solution is no. Otherwise a solution is a subset  $S$  of the weights that represents a feasible packing. For the above example, one solution is “4 6 16”. The total weight is therefore  $W = 4 + 6 + 16 = 26$ , and  $L \leq 26 \leq H$ , as required.

### PROBLEM SUBSETSUM

- **Input:** Same as for PACKING, except that  $L$  is omitted. Example: “12 4 6 24 4 16 ; 32”.
- **Solution:** Same as for PACKING, except that the weight of a feasible packing must exactly equal the threshold  $H$ . For the above example, one solution is “12 4 16”.

### PROBLEM PARTITION

- **Input:** Same as for PACKING, except that  $L$  and  $H$  are omitted. Example: “12 4 6 14 4 16”.
- **Solution:** Same as for PACKING, except that the weight of a feasible packing must equal exactly half the total weight of the packages. That is, the packing splits the packages into two subsets of equal weight. For the above example, one solution is “12 16”.

**Figure 12.6:** Description of the computational problems PACKING, SUBSETSUM, and PARTITION.

### The haystack analogy for PolyCheck

We can often think of a problem in PolyCheck as being a search for a positive solution among a vast space of possible solutions. Metaphorically, it’s a search for a needle in a haystack. Think of searching for the factors of a 10,000-digit number: each factor is a needle, and each nonfactor is a piece of hay. In this example, there are far more pieces of hay than atoms in the observable universe.

Pushing our metaphor a little further, we could say that a problem is in PolyCheck if there is an efficient way of verifying needles. We need to be able to glance at a needle and quickly conclude “yes, that’s a needle, it’s definitely not a piece of hay.” Of course, the ability to efficiently verify needles does not allow

us to find them efficiently. To determine that there are no needles in a given haystack (i.e., that the haystack is a negative instance), we might need to examine each piece of hay, which would take an extraordinarily long time.

## 12.4 THE COMPLEXITY CLASS NPoly

Our next objective is to define a new complexity class called **NPoly**. We will later find that **NPoly** and **PolyCheck** are identical, but their definitions are very different and we pursue **NPoly** separately for now. **NPoly**'s definition uses nondeterministic running times, so we first need a clear definition of the running time of a nondeterministic program. Informally, the nondeterministic running time is the time taken by the longest thread. Here's the formal definition:

**Definition of the nondeterministic running time.** Let  $P$  be a nondeterministic program and  $I$  an input string. If one or more threads of  $P$  do not halt on input  $I$ , the *nondeterministic running time* of  $P$  on input  $I$  is infinite. Otherwise we may assume all threads halt and the *nondeterministic running time* of  $P$  on input  $I$  is the running time of the last thread to halt. More precisely, it is the maximum number of computational steps in any path from the root of the computation tree to a leaf.

Another way of thinking about this is to say that when many different threads simultaneously take a computational step, this is counted as only a single step of nondeterministic time. Because the threads all run in parallel, the total amount of nondeterministic time consumed is just the time for the longest thread to complete.

For an example of nondeterministic running time, refer back to `ndContainsNANA.py`, in figure 8.3 (page 146). Almost every line of this program runs in constant time. The only non-constant-time operation is the string search at line 37, which runs in  $O(n)$  time. This line is executed simultaneously in several different threads, but all of them will terminate within  $O(n)$ . So the nondeterministic running time of the program is in  $O(n)$ .

Now we are ready to define the complexity class **NPoly**:

**Definition of NPoly.** The complexity class **NPoly** consists of all computational problems that can be solved by a nondeterministic, polynomial-time program.

**FACTOR** (page 215) is a good example of an **NPoly** problem. We can easily imagine a nondeterministic program that launches a separate thread for every possible factor of the input, and tests them all simultaneously. This nondeterministic program will terminate in roughly the time it takes to test a single factor, which is certainly polynomial. The details of writing a real Python program to do this are a little messy, so we won't examine the source code in detail. If you're interested, take a look at `ndFactor.py`, which is provided with the book materials. The `ndFactor.py` program eventually launches  $O(M) = O(10^n)$  threads. But although it launches exponentially many threads, it does so in a manageable way: each thread launches at most two additional threads. So the computation tree

ends up being a binary tree of depth  $O(\log M) = O(n)$ . And each thread runs in time  $O(n^2)$  since the most expensive single operation is an integer division. So the nondeterministic running time is in  $O(n \times n^2) = O(n^3)$ .

Other examples of NPoly problems include TSPD, PACKING, SUBSETSUM, and PARTITION. In each case, we can imagine launching a vast number of threads—one for each possible solution—and testing each solution in parallel. In general, the threads are launched via a computation tree of polynomial depth. And each “leaf” thread in this tree tests one particular solution within polynomial time. Thus, the problem is solved in nondeterministic polynomial time. (Technical note: If you’re concerned about these trees being too “bushy,” recall from the discussion on page 152 that we may assume that computation trees are in fact binary trees.)

This is only a very brief sketch of how to solve TSPD, PACKING, SUBSETSUM, and PARTITION in nondeterministic polynomial time. But it turns out there’s no need to go into more detail. We’ve already seen that each of these problems is a member of PolyCheck, and in the next section we will prove that PolyCheck and NPoly are identical. This will immediately imply that all our PolyCheck problems are also NPoly problems.

## 12.5 PolyCheck AND NPoly ARE IDENTICAL

Recall the definitions of complexity classes PolyCheck and NPoly:

- PolyCheck: problems whose positive instances can be verified in polynomial time (using a hint, if needed).
- NPoly: problems that can be solved in polynomial time by a nondeterministic program.

These two definitions appear to rely on completely different properties. We will now prove the remarkable result that the two definitions yield exactly the same complexity class:

**Claim 12.2.** PolyCheck and NPoly are identical. That is, any problem in PolyCheck is also in NPoly, and vice versa.

*Proof of the claim.* We prove this claim by splitting it into two subclaims: (i) PolyCheck  $\subseteq$  NPoly (claim 12.3, page 260), and (ii) NPoly  $\subseteq$  PolyCheck (claim 12.4, page 261). These two subclaims are proved below, and the main claim above then follows immediately.  $\square$

### Every PolyCheck problem is in NPoly

In this section, we will be proving that every PolyCheck problem is in NPoly. Let’s first run through an informal overview of the proof. We are given a polytime *deterministic* program that can *verify* solutions, perhaps with the help of a hint. And we need to construct a polytime *nondeterministic* program that can *construct* solutions, with no hint. For any particular problem, it’s usually obvious how to do this. As examples, consider nondeterministic programs for the following two problems: (i) for FACTOR, launch a thread for each possible factor, and check

them all in parallel (see `ndFactor.py` for details); (ii) for TSPD, launch a thread for each possible permutation of the graph's nodes, and check each permutation in parallel to see if it's a Hamilton cycle that's short enough.

But now we want to prove that this approach works for any problem in PolyCheck. The idea is the same, but the details are more challenging, because we don't know what kinds of solutions should be searched in parallel (factors?, permutations?). The trick is to think of the possible solutions and hints merely as strings. Then we can launch a separate thread for each possible solution and hint, using the verifier to check each one in parallel. That's exactly how we prove the following claim:

**Claim 12.3.** Every PolyCheck problem is in NPoly.

*Proof of the claim.* Let  $F$  be a problem in PolyCheck. Because  $F \in \text{PolyCheck}$ , we have a (deterministic) polytime verifier  $V(I, S, H)$ . From this, we will create a nondeterministic polytime program  $P(I)$  that solves  $F$ .

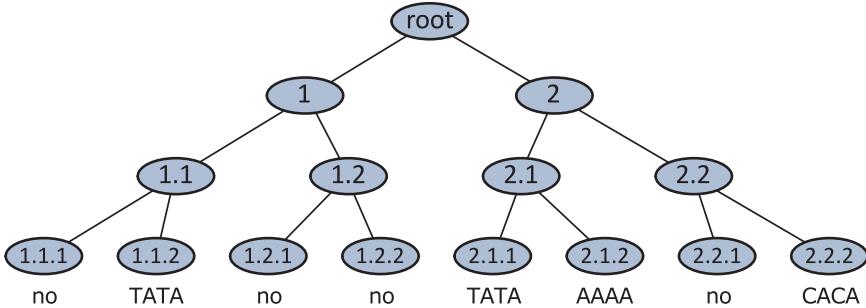
Here is how  $P(I)$  operates: It runs exactly like  $V(I, S, H)$ , with one exception. Every time one of the characters in  $S$  or  $H$  is read by the program,  $P(I)$  launches a new thread for every symbol in the alphabet. This part of the proof is easiest to understand in the Turing machine model. As in the proof of claim 12.1 on page 255, think of  $V$  as a deterministic multi-tape Turing machine with  $S$  and  $H$  provided on separate, special input tapes. The machine  $P$  mimics  $V$ , except that every time it consults a previously unread cell  $C$  on one of the two special input tapes, it launches a set of new clones—one for each possible symbol that could be in cell  $C$ .

(This proof is rather abstract. To help your understanding, think about the FACTOR example above, and for simplicity assume we use the 10-symbol alphabet of digits. Every time  $V$  reads a new digit of the proposed factor  $S$ , then  $P$  launches 10 new clones—one for each possible digit. After  $V$  has read, say, 5 cells of the  $S$ -tape,  $P$  has  $10^5$  clones, each of which is ready to verify a different 5-digit factor.)

In this way,  $P(I)$  considers all possible strings for  $S$  and  $H$ , up to some maximum length bounded by the polynomial running time of  $V$ . The return value of a thread depends on the value of  $V(I, S, H)$  computed. There are only two possible cases: if  $V(I, S, H) = \text{"correct"}$ , the return value is the correct solution  $S$ ; if  $V(I, S, H) = \text{"unsure"}$ , the return value is  $\text{"no"}$ . Because  $V(I, S, H)$  terminates in polynomial time for every  $S$  and every  $H$ , every thread of  $P(I)$  also terminates in polynomial time. And from the construction of  $P(I)$ , it's clear that  $V(I, S, H) = \text{"correct"}$  for some  $S, H$  if and only if  $P(I)$  returns a correct positive solution for  $F(I)$ . (For deeper understanding, let's continue the example of FACTOR. If  $P$  has  $10^5$  clones checking factors  $m$  from 1 to  $10^5$  in parallel, one of the clones will successfully detect and return a factor if and only if there exists some  $m$  which  $V$  would verify as  $\text{"correct"}$ .)  $\square$

## Every NPoly problem is in PolyCheck

Before proving our next claim, we need to recall an important concept about nondeterministic computations: these computations can be represented by



**Figure 12.7: A computation tree representing the threads launched by a nondeterministic program.** Note that every path from the root to a leaf is described unambiguously by the leaf's label, such as “2.1.2” for the thread that returns “AAAA”.

*computation trees*. For detailed explanations, look back to figure 8.6 (page 150); one of those examples is repeated here in figure 12.7, for easy reference. The key point is that every leaf of the tree can be given a unique label, such as “2.1.2” for the thread that returns “AAAA” in figure 12.7. This label describes exactly the path from the root to the chosen leaf. We think of this label as a list of integer *components*  $c_0, c_1, \dots$ , separated by periods. So for the specific example “2.1.2”, we have  $c_0 = 2, c_1 = 1, c_2 = 2$ . The  $c_i$  represent ID numbers of the threads launched at each node. We can replicate the computation leading to the chosen leaf by following the thread with ID number  $c_i$  at level  $i$  in the tree. For example, to reach the “AAAA” leaf node in figure 12.7, we follow thread 2 at depth 0, thread 1 at depth 1, and thread 2 at depth 2.

We are now ready to prove our next claim.

**Claim 12.4.** Every NPoly problem is in PolyCheck.

*Proof of the claim.* Let  $F$  be a problem in NPoly. That means there is a nondeterministic program  $P(I)$  computing solutions to  $F$  in polynomial time. We need to construct a (deterministic) polytime verifier  $V(I, S, H)$ .

The program  $V$  will actually mimic program  $P$ , except that it will choose exactly one path from the root of the computation tree to a leaf. So  $V$  will indeed be deterministic, because it performs the same computation as exactly one of the threads from  $P$ .

Initially, we will describe  $V$ 's behavior for an instance  $I$  when verifying a correct positive solution  $S$  using a correct hint  $H$ . In this case,  $S$  can be any one of the positive solutions returned by a leaf of the computation tree (e.g., we might have  $S = “AAAA”$  in the tree of figure 12.7). The hint will be the label of the chosen leaf (e.g.,  $H = “2.1.2”$  when  $S = “AAAA”$ ). As suggested above, we think of this hint as a list of integer *components*  $c_0, c_1, c_2, \dots$ , separated by periods.

Recall that the verifier  $V$  mimics  $P$ , with one exception. The exception occurs every time  $P$  makes a nondeterministic choice and launches some new threads. The first time this happens,  $V$  uses the first component of the hint,  $c_0$ , to make a single deterministic choice—instead of simulating all of  $P$ 's new threads, it simulates only the new thread with ID number  $c_0$ . To simulate the next

nondeterministic choice by  $P$ , then  $V$  simulates the new thread with ID number  $c_1$ , and so on for the remainder of the hint. Assuming for the moment that the hint is correct,  $V$  will travel down the computation tree to the chosen leaf and compute the return value  $R$  at that leaf. (So in our running example, we would have  $R = \text{“AAAA”}$ .) Then  $V$  checks that  $S = R$ , returning “correct” if  $S = R$ , and “unsure” otherwise.

Now consider the various cases where  $S$  and/or  $H$  are not correct. If the hint is obviously wrong at any point,  $V$  immediately returns “unsure”. This happens, for example, if  $H$  is incorrectly formatted, or provides a nonexistent thread ID, or is too short so that the hint is exhausted before  $R$  is computed. If  $S$  is an incorrect proposed solution, the final check that  $S = R$  fails, and  $V$  again returns “unsure”.

One final technicality should be mentioned: if  $S = \text{“no”}$ , then  $V$  returns “unsure” immediately, without bothering to simulate any of  $P$ ’s computations.

Let’s now check that  $V$  satisfies the three boldface conditions for the definition of a verifier on page 252. Positive instances can always be verified by choosing  $S, H$  leading to a positive leaf. Negative instances can never be verified, since the final  $S = R$  check will fail (unless  $S = \text{“no”}$ , but  $V$  returns “unsure” immediately in such cases). Incorrect proposed solutions can never be verified for exactly the same reason: the final  $S = R$  check must fail.

So it remains to show only that  $V$  runs in (deterministic) polynomial time. This follows immediately from the fact that  $P$  runs in nondeterministic polynomial time. By definition,  $P$ ’s threads always terminate in polynomial time. And since  $V$  simulates just one path from the root to a leaf of  $P$ ’s computation tree,  $V$  runs in deterministic polynomial time.  $\square$

We can add a little more detail to the above proof by thinking about what changes would need to be made to  $P$ ’s code to produce  $V$ . It’s reasonable to assume that any part of the nondeterministic program  $P$  that launches new threads looks something like this:

```

treeDepth = treeDepth + 1
2 for nondetChoice in range(numChoices):
    t = threading.Thread(target=someFunction, \
4     args = (inString, treeDepth, nondetChoice, otherParams))
    t.start()

```

In the verification program  $V$ , we change the above code to

```

treeDepth = treeDepth + 1
2 nondetChoice = hint[treeDepth]
someFunction(inString, treeDepth, nondetChoice, otherParams)

```

It is strongly recommended that you study and understand these two code snippets—they contain the key idea from the previous proof.

Combining the last two claims, we’ve now proved one of the key results of complexity theory: PolyCheck and NPoly are identical. We’ll sometimes emphasize this fact by referring to the single complexity class PolyCheck/NPoly.

## 12.6 THE PolyCheck/NPoly SANDWICH

We saw in chapter 11 that  $\text{Poly} \subseteq \text{Expo}$ . But where does PolyCheck/NPoly fit in this hierarchy? It turns out that PolyCheck/NPoly is sandwiched between Poly and Expo, so we have

$$\text{Poly} \subseteq \text{PolyCheck/NPoly} \subseteq \text{Expo}. \quad (\star)$$

We prove the two subset relations in  $(\star)$  via two separate claims below. But first, recall that by claim 11.6 on page 241, we know Poly is a *proper* subset of Expo. This implies that at least one of the two subset relations in  $(\star)$  is proper. The properness of the first inclusion ( $\text{Poly} \subseteq \text{PolyCheck/NPoly}$ ) is equivalent to the P versus NP question discussed in chapter 14. The properness of the second inclusion ( $\text{PolyCheck/NPoly} \subseteq \text{Expo}$ ) also interests complexity theorists but is not discussed in this book. Both inclusions are believed to be proper, and we know that at least one of them must be. But neither of them has been proved to be proper! This strange situation is one of the many reasons for the importance of the P versus NP question.

Let's now proceed with a proof of  $(\star)$  in two separate claims.

**Claim 12.5.** Poly is a subset of PolyCheck/NPoly.

*Proof of the claim.* It would be sufficient to use either the definition of PolyCheck or NPoly, since they actually define the same thing. But to deepen our understanding of these definitions, we will give two separate proofs.

*Proof using PolyCheck.* Let  $F \in \text{Poly}$ . Then there is a polytime program  $P$  solving  $F$ . To show  $F \in \text{PolyCheck}$ , we need a polytime verifier  $V(I, S, H)$  for  $F$ . It's very easy to construct  $V$ : first,  $V$  runs  $P$  to obtain  $P(I)$ , then  $V$  returns “correct” if  $S = P(I)$ , and “unsure” otherwise. Note that  $V$  ignores  $H$ . Another minor technicality is that if  $S = \text{“no”}$ ,  $V$  immediately returns “unsure”. It's easy to check that  $V$  satisfies the conditions of a verifier, and it's clear that  $V$  runs in polynomial time because  $P$  is polytime.

*Proof using NPoly.* As above, let  $F \in \text{Poly}$ , and let  $P$  be a polytime program solving  $F$ . To show  $F \in \text{NPoly}$  we need a nondeterministic polytime program  $P'$  that solves  $F$ . This is easy: we take  $P' = P$ . (This works because we can regard  $P$  as a nondeterministic program that happens not to employ any nondeterminism.)  $\square$

**Claim 12.6.** PolyCheck/NPoly is a subset of Expo.

*Proof of the claim.* As in the previous claim, we deepen our understanding by giving separate proofs for the two definitions of PolyCheck/NPoly.

*Proof using PolyCheck.* Let  $F \in \text{PolyCheck}$ . So we have a polytime verifier  $V(I, S, H)$  for  $F$ , and we need to construct an exponential-time program  $P(I)$  that solves  $F$ . The high-level idea is that  $P$  runs  $V$  for every relevant choice of  $S$  and  $H$ . If a positive solution exists, it will verify as “correct” and  $P$  can return this solution. If not, every run of  $V$  will return “unsure”, so  $P$  returns “no”.

It remains to show that  $P$  always terminates within exponential time. The time for each run of  $V$  is bounded by some polynomial  $q(n)$ , because  $V$  is a polytime verifier. But how many runs could there be? That depends on how many “relevant” choices there are for  $S$  and  $H$ . Looking back to claim 12.1 on

page 255, we see that there is some other polynomial  $p(n)$  bounding the number of characters in  $S$  and  $H$  that can affect the output. If there are  $K$  symbols in the alphabet, the total number of relevant possibilities for  $S$  and  $H$  is  $K^{2p(n)}$ . Hence, the total running time for program  $P$  is bounded by  $q(n)K^{2p(n)}$ . This is dominated by, say,  $(K+1)^{2p(n)}$ , which meets the definition of  $\text{Expo}$  on page 228.

*Proof using NPoly.* Let  $F \in \text{NPoly}$ . So we have a nondeterministic polytime program  $P$  that solves  $F$ . We need to produce a new, deterministic exponential-time program  $P'$  that also solves  $F$ . The basic idea is to simulate  $P$  deterministically. This can be done by executing each node in the computation tree in breadth-first order, remembering any required state so that computations in the next level can resume correctly when they are reached. The main challenge is to prove that this approach runs in exponential time. By the definition of nondeterministic polynomial time, we can assume the threads of  $P$  all have running times bounded by some polynomial  $q(n)$ . As discussed on page 152, we may assume the computation tree is in fact a binary tree, so the number of leaves is bounded by  $2^{q(n)}$ . This gives a total running time for  $P'$  of at most  $q(n)2^{q(n)}$ . This is dominated by, say,  $3^{q(n)}$ , which meets the definition of  $\text{Expo}$  on page 228.  $\square$

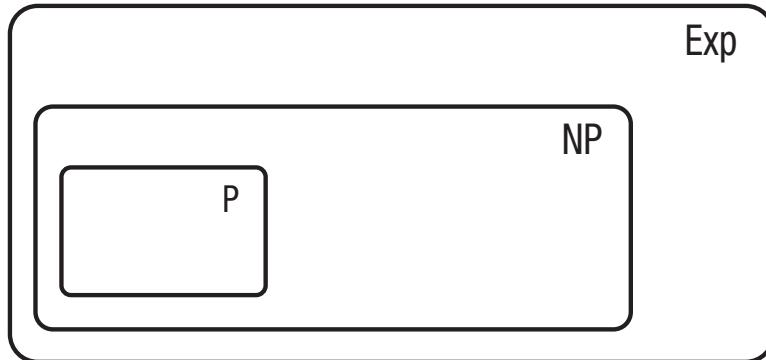
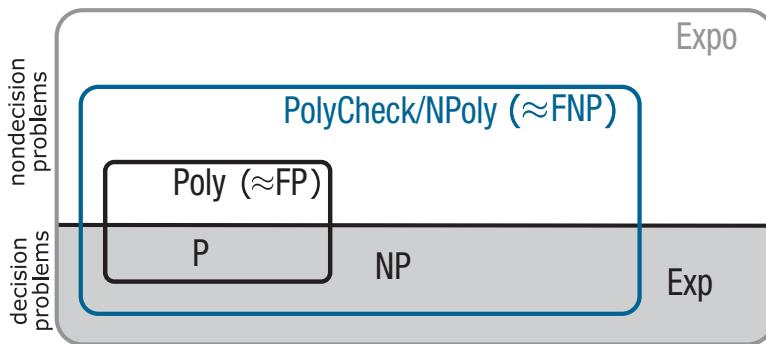
Figure 12.8 summarizes the relationships between the main complexity classes we have encountered so far, clearly demonstrating the “sandwich” of PolyCheck/NPoly between Poly and Expo.

## 12.7 NONDETERMINISM DOES SEEM TO CHANGE WHAT IS COMPUTABLE EFFICIENTLY

From section 8.4, we know that adding nondeterminism to our computational model doesn’t change *what* we can compute. But does it change *how fast* we can compute it? In a certain sense, the answer is clearly yes: ever since the multicore era began (sometime after the year 2000), most CPU-intensive software has been written to take advantage of multiple cores simultaneously. This is for the obvious reason that if your computer has, say, 8 cores, multithreaded code can run up to 8 times faster than the equivalent single-threaded code. Similarly, if your data center has hundreds of thousands of cores, certain types of programs can be run simultaneously on all of them, obtaining a speedup that could be in the hundreds of thousands, compared to running the same computation on a single core.

But complexity theorists demand more. These speedups, whether a factor of 8 or 100,000, are just constants—and as we know, the theoretical analysis of running times ignores constants. To a complexity theorist, then, the real question is more abstract: Does the use of nondeterministic programs allow us to efficiently solve problems that can’t be solved efficiently with deterministic programs? More precisely, we refer back to the first half of the sandwich relationship ( $\star$ ) on page 263: Is  $\text{NPoly}$  a proper superset of  $\text{Poly}$ ? The answer is widely believed, but has not been proved, to be yes. Chapter 14 addresses this in much greater detail, but for now let’s summarize the most important properties of nondeterminism in a single slogan:

Nondeterminism does not alter computability, but it is widely believed to improve computational efficiency.

(a) Relationship between classes of **decision problems**(b) Relationships between classes of both **decision problems** (shaded), and **nondescription problems** (unshaded)

**Figure 12.8: Relationships between some complexity classes.** Both diagrams adopt the widely believed but unproven assumptions  $P \neq NP$  (which will be discussed in the next chapter) and  $NP \neq Exp$ . The top panel (a) shows the relationships for decision problems only, whereas (b) shows the relationships between all types of problems. The shading in (b) reminds us that  $P \subset Poly$ ,  $NP \subset NPoly$ , and  $Exp \subset Exp$ . Many treatments of complexity theory discuss FP and FNP, which are roughly equivalent to Poly and NPoly respectively. See the discussions on pages 229 and 266 for more detailed explanations.

## 12.8 THE FINE PRINT ABOUT NPoly

This is an optional section that can be skipped by most readers. The objective is to provide more details and motivation for this book's definitions of the non-standard complexity classes NPoly and PolyCheck, especially for readers who have encountered the more standard class FNP in other sources.

### An alternative definition of NPoly

A very perceptive reader may have noticed a strange inconsistency between our definition of the *output* of a nondeterministic computation (page 152) and our

definition of the *running time* of a nondeterministic computation (page 258). The inconsistency arises when the computation tree is infinite but possesses at least one positive leaf. In this case, the output is defined but the running time is infinite. Wouldn't it be more natural to define the running time in this case as the number of computational steps until the first positive leaf is encountered? Yes, it would be more natural to do that, because it mirrors the way multithreaded programs actually work: as soon as one thread has found a satisfactory solution, all the other threads can be killed and the program terminates.

Unfortunately, this alternative definition of nondeterministic running time makes the other definitions and proofs in this chapter somewhat more complicated. Nevertheless, it's worth noting that the alternative definition leads to exactly the same complexity class **NPoly**, and all our other claims remain true too.

Let us briefly sketch the reason for this. Suppose we have a nondeterministic program  $P$  that runs in polynomial time according to our alternative definition. That is, there is some polynomial  $q(n)$  that bounds the number of computational steps until *either* some thread terminates positively *or* all threads terminate negatively. We can convert  $P$  into a new nondeterministic program  $P'$  that runs in polynomial time according to our original definition. Program  $P'$  simulates all the threads of  $P$  simultaneously, but it also keeps a count of the number of computational steps that have been simulated in each thread. If any thread exceeds  $q(n)$  steps, it is immediately terminated. There is some overhead introduced by the simulation, but the amount of overhead is only a small polynomial factor. It's easy to check that  $P'$  produces the same outputs as  $P$ , but now all threads are guaranteed to terminate in polynomial time, as required by the original definition.

By the way, did it bother you that the construction of  $P'$  seems to require knowledge of the *value* of  $q(n)$ , not merely the *existence* of  $q$ ? This kind of “constructivist” objection to certain proofs does seem rather reasonable here. Nevertheless, the above argument does prove the existence of  $P'$ , even if the algorithm for constructing  $P'$  is unclear.

## NPoly compared to NP and FNP

Let's now try to understand our definition of **NPoly** in terms of the classes **NP** and **FNP**, which are frequently used by other books. The relationship between **NPoly**, **NP**, and **FNP** resembles the relationship between **Poly**, **P**, and **FP** which has already been discussed on page 229.

As usual, we are working with general computational problems, so **NPoly** was defined as a class of general problems. If we restrict **NPoly** to decision problems only, we obtain the class known as **NP**, which will be discussed at considerable length in chapter 14. If we are restricting to decision problems, the definition of verifiers can be simplified by eliminating the proposed solution parameter  $S$ . A verifier then becomes a program  $V(I, H)$  such that  $V(I, H) = \text{"yes"}$  for some  $H$  if and only if  $I$  is a positive instance. In this context, the hint string  $H$  is sometimes called a *witness, certificate, or proof*.

**NPoly/PolyCheck** is similar to the complexity class known as **FNP**, for “function, nondeterministic, polynomial time.” In fact, it turns out that **FNP** is a strict subset

of PolyCheck, but both classes capture the idea of problems whose solutions are easy to verify. The rest of this optional section provides the relevant technical details.

One standard way of defining FNP is as follows. We fix an alphabet  $\Sigma$  and consider binary relations  $R(I, S)$  on  $\Sigma^* \times \Sigma^*$ . The relation  $R$  is *polynomially bounded* if there is some polynomial  $q(n)$  such that  $|S| \leq q(|I|)$  whenever  $R(I, S)$  holds. A polynomially bounded relation  $R$  is in FNP if there is a deterministic polytime program  $\hat{V}(I, S)$  that decides  $R(I, S)$  for all  $I, S$  with  $|S| \leq q(|I|)$ .

Note that it does not matter whether  $\hat{V}(I, S)$  runs in polynomial time as a function of  $|I|$  only or as a function of  $|I| + |S|$ . This is because we insisted  $|S| \leq q(|I|)$ . (Technical note: We could instead define FNP for all  $R$ , rather than just polynomially bounded  $R$ . In this case,  $\hat{V}$  would need to run in polynomial time as a function of  $|I|$  alone, and the definition of FNP would need an existential quantifier on  $S$ . This would lead to a definition of FNP somewhat closer to PolyCheck, as we will see below.)

In this book, our definitions used solution set functions rather than binary relations, but these notions are equivalent: given a computational problem  $F$ , let  $R_F$  be the relation where  $R_F(I, S)$  holds if and only if  $S \in F(I)$ . Similarly, given binary relation  $R$  on  $\Sigma^* \times \Sigma^*$ , write  $F_R$  for the function mapping  $I$  to  $\{S \mid R(I, S)\}$ . For complete compatibility with this book's definitions, we would need to insist that negative instances are related to the fixed string “no”, or to some other fixed string if a non-ASCII alphabet is used. But for convenience, we relax that convention here.

Members of PolyCheck are functions, whereas members of FNP are relations. Therefore, in the remainder of this section, we assume the above conversion between functions and relations is applied whenever necessary. For example, the statement “ $\text{FNP} \subsetneq \text{PolyCheck}$ ” implicitly requires that the elements  $R$  of FNP be converted to functions  $F_R$  in PolyCheck.

### **Claim 12.7.** $\text{FNP} \subsetneq \text{PolyCheck}$ .

*Proof of the claim.* First we show that  $\text{FNP} \subseteq \text{PolyCheck}$ . This is immediate because the FNP verifier  $\hat{V}(I, S)$  for a given problem can be used as the PolyCheck verifier  $V(I, S, H)$ , with an arbitrary value such as the empty string for the hint  $H$ .

Next we show that PolyCheck is a strict superset of FNP. There are in fact two separate reasons and both are worth mentioning. The first reason is a minor technicality: FNP is defined only for polynomially bounded relations, whereas computational problems in PolyCheck might have solutions in their solution sets whose length is not polynomially bounded. (Positive instances always have at least one polynomially bounded solution, however.) The second reason is more fundamental: FNP-style verifiers (denoted  $\hat{V}$  above) are not allowed to use hints. Therefore, PolyCheck problems that require a hint for polytime verification are not in FNP. For example, let UHP be the problem asking for a Hamilton path in an undirected graph if one exists, or “no” otherwise. Solutions can be efficiently checked without needing a hint, so UHP is in both PolyCheck and FNP. But consider the problem UHPOIGIN, which asks for just the first node of some Hamilton path if one exists, or “no” otherwise. There is no known polytime algorithm for checking solutions to UHPOIGIN, so it's widely believed

that UHPOIGIN  $\notin \text{FNP}$ . But it's easy to check a solution with the right hint, so UHPOIGIN  $\in \text{PolyCheck}$ .  $\square$

## EXERCISES

**12.1** Write polytime verifiers in Python for (a) PACKING, (b) SUBSETSUM, and (c) PARTITION.

**12.2** Explain why each of the following problems is in PolyCheck. You must use the definition of PolyCheck, not NPoly.

- (a) The decision version of the undirected Hamilton cycle problem. That is, the input consists of an undirected graph  $G$ . The solution is “yes” if  $G$  has a Hamilton cycle, and “no” otherwise.
- (b) Let's call this problem INTEGERDIVISION. The input consists of two integers,  $M$  and  $N$ , written in decimal notation and separated by a space character. If  $M$  is divisible by  $N$ , the solution is  $M/N$ , again written in decimal notation. If  $M$  is not divisible by  $N$ , the solution is “no”.
- (c) Let's call this problem ANONYMOUSFRIENDS. The input consists of two graphs,  $L$  (for Labeled) and  $A$  (for Anonymous). The graph  $L$  consists of nodes labeled with peoples' names, and edges between any two people that are friends on some particular social network, such as Facebook. The graph  $A$  has the same format, but the nodes have been permuted randomly and are labeled with anonymous strings rather than peoples' names. The ANONYMOUSFRIENDS problem asks, could  $L$  and  $A$  represent the same set of people? (Note: ANONYMOUSFRIENDS is just a disguised version of a famous problem called GRAPHISOMORPHISM. Look ahead to page 309 for a definition.)
- (d) The problem UHPOIGIN, defined on page 267.

**12.3** Repeat the previous question, this time proving membership in NPoly. You must directly use the definition of NPoly in each case.

**12.4** Consider the following decision problem, called IMPOSSIBLETOBALANCE. The input is a list of integers, such as “45 23 4 3 72 12”. The solution is “yes” if it is impossible to partition the integers into two sets that *balance*—that is, the sum of each set is the same. Otherwise the output is “no”. For example, on input “1 2 4” the solution is “yes” (because it's impossible to partition the input into two balanced sets), and on input “1 2 4 1 2” the solution is “no” (because we can balance  $1 + 2 + 2 = 4 + 1$ ). Your friend claims that IMPOSSIBLETOBALANCE is in PolyCheck, because we can provide as a hint a list of all partitions of the integer inputs, together with the weights of these partitions. Explain why your friend's reasoning is not correct.

**12.5** Consider the proof of claim 12.4 on page 261. Assume we are applying the method described in this proof to construct a verifier  $V(I, S, H)$  from a nondeterministic program  $P$  whose computation tree for some particular input  $I_0$  is shown in figure 12.7. Give the value of  $V(I_0, S, H)$  when

- (a)  $S = \text{"TATA"}, H = \text{"2.1.2"};$
- (b)  $S = \text{"TATA"}, H = \text{"TATA"};$

```

1  def verifyCheckMultiply(I, S, H):
2      M1, M2, K = [int(x) for x in I.split()]
3      if M1*M2==K and S=='yes':
4          return 'correct'
5      elif M1*M2!=K and S=='no':
6          return 'correct'
7      else:
8          return 'unsure'

```

**Figure 12.9:** The Python program `verifyCheckMultiply.py`. This program contains a deliberate error—see exercise 12.7.

```

1  def verifyAdd(I, S, H):
2      if len(S)>2*len(I) or len(H)>0:
3          return 'unsure'
4      M1, M2 = [int(x) for x in I.split()]
5      S = int(S)
6      total = M1
7      for i in range(M2):
8          total += 1
9      if total == S:
10         return 'correct'
11     else:
12         return 'unsure'

```

**Figure 12.10:** The Python program `verifyAdd.py`. See exercise 12.8.

- (c)  $S = \text{"TATA"}, H = \text{"1.1.2"};$
- (d)  $S = \text{"no"}, H = \text{"1.2.2"};$
- (e)  $S = \text{"no"}, H = \text{"1.1.2"}.$

**12.6** While taking an exam for a college course on computational theory, a student encounters the following question: “Give an example of a computational problem that is in  $\text{NPoly}$  but outside  $\text{Poly}$ .” Explain why this is not a reasonable exam question, and suggest a way of altering the question so that it is reasonable. Give an answer for your new version of the question.

**12.7** Consider the program `verifyCheckMultiply.py` shown in figure 12.9. This program contains a mistake: it is *not* a correct implementation of a verifier for the problem `CHECKMULTIPLY` defined in figure 4.13, page 60. Explain how to fix the program so that it is a verifier for `CHECKMULTIPLY`.

**12.8** Define the function problem `ADD` as follows. The input is an ASCII string containing two positive integers  $M_1, M_2$  in decimal notation separated by whitespace. (Example: “223 41”.) The unique solution is  $M_1 + M_2$ , expressed as an ASCII string in decimal notation. (Example using the previous input: “264”.) Now consider the program `verifyAdd.py` in figure 12.10. Is this program a polytime verifier for `ADD`? Explain your answer. (Note: In answering this

question, you may assume that all parameters are correctly formatted—so  $I$  is a string containing two positive integers separated by whitespace, and  $S$  is a string containing a positive integer.)

**12.9** Let SUBSETSUMD be the decision version of SUBSETSUM, and suppose  $V(I, S, H)$  is a verifier for SUBSETSUMD. What is the maximum length of (a)  $S$  and (b)  $H$  that  $V$  needs to consider?

**12.10** Let  $V(I, S, H)$  be a verifier for PACKING that does not employ the hint  $H$ . That is, the output of  $V$  is independent of  $H$ .

- (a) What is  $V("5 5 5 5; 15; 25", "5 5 5", \epsilon)$ ?
- (b) What is  $V("5 5 5 5; 20; 25", "5 5 5", \epsilon)$ ?
- (c) What is  $V("5 5 5 5; 16; 19", "5 5 5", \epsilon)$ ?
- (d) Is there any value of  $S$  such that  $V("5 5 5 5; 16; 19", S, \epsilon) = \text{"correct"}$ ? If so, give an example. If not, explain why not.

**12.11** Consider the formal definition of a verifier on page 251:

- (a) In which bullet point is  $H$  existentially quantified? (That is, a statement applies to some  $H$ , not all  $H$ .) Explain why the definition would be incorrect if this quantifier were reversed (“all” instead of “some”).
- (b) In which two bullet points is  $H$  universally quantified? (That is, a statement applies to all  $H$ , not some  $H$ .) For each of the two locations, explain why the definition would be incorrect if this quantifier were reversed.
- (c) Repeat the above exercises for the proposed solution  $S$ : find all places where  $S$  is universally or existentially quantified, and explain why the definition fails if the quantifier is reversed.

**12.12** Examine the first proof of the claim that  $\text{Poly} \subset \text{PolyCheck}$  (claim 12.5, page 263). Suppose that the solution set  $F(I)$  contains multiple elements, and that  $P(I)$  happens to compute one of the solutions that is not equal to  $S$ . (a) What does  $V$  return in this case? (b) Explain why the proof is still correct, despite your answer to (a). (Hint: This is easy if you have completed exercise 12.11.)

**12.13** Define the computational problem DIFFERENTPARTITIONS as follows: The input is the same as for PARTITION. If the weights can be partitioned equally, a solution is a list of two weights that come from different “sides” of the partition (and otherwise the solution is “no”). More precisely, if the weights can be partitioned into sets  $W_1$  and  $W_2$  with  $\sum_{w_i \in W_1} w_i = \sum_{w_j \in W_2} w_j$ , then a solution is any pair  $w_i, w_j$  such that  $w_i \in W_1$  and  $w_j \in W_2$ . For example, solutions to the instance “2 4 6 8” include “2 4”, “2 6”, and “8 4”.

- (a) Prove that DIFFERENTPARTITIONS is in PolyCheck.
- (b) What is the maximum length of hint that a verifier for DIFFERENTPARTITIONS needs to consider?

**12.14** Consider the **decision** problem SUBSETSUMWITHFIVES (abbreviated to SSWITH5S), defined as follows: SSWITH5S is identical to the **decision** variant of SUBSETSUM, except we may use up to 10 extra packages with weight 5 when constructing the desired subset. Write a polytime verifier for SSWITH5S in Python.

**12.15** Section 12.8 described the possibility of defining nondeterministic polynomial time in terms of positive leaves only. In fact, we can take this approach for deterministic polynomial time also. Let us say that a deterministic program  $P$  runs in *alternative polynomial time* if the running time of  $P$  is bounded by a polynomial for all inputs  $I$ , where  $P(I)$  is defined and positive. Given a  $P$  that runs in alternative polynomial time, prove that there exists another deterministic program  $P'$  that runs in polynomial time according to our standard definition, such that (a) for all  $I$ ,  $P'(I)$  is defined and positive if and only if  $P(I)$  is defined and positive, and (b)  $P'(I) = P(I)$  for all such positive instances  $I$ .

# 13



## POLYNOMIAL-TIME MAPPING REDUCTIONS: PROVING $X$ IS AS EASY AS $Y$

What makes polynomial time reducibility an important tool in the study of computational complexity is the fact that *if  $A$  is polynomial time reducible to  $B$  and  $A$  is not computable in polynomial time then neither is  $B$ .*

—Ladner, Lynch, and Selman, *A Comparison of Polynomial Time Reducibilities* (1975)

We begin this chapter by adopting an important new convention:

**We work only with *decision* problems for the next two chapters.**

Why? Because we are building towards the definition in the next chapter of a crucial concept called *NP-completeness*. It turns out that NP-completeness is defined only for decision problems. Therefore, in this chapter and the next chapter, we will work almost exclusively with decision problems. *All computational problems, regardless of how they were defined earlier, will be considered as decision problems unless stated otherwise.* For example, the problems PARTITION and PACKING were defined as nondecision problems on page 257. But in the next two chapters, these problems have only the solutions “yes” (if there is a feasible packing), and “no” (if there is no feasible packing). All of the problems discussed will have simple, obvious conversions between their decision and nondecision variants. Consult figure 4.13 (page 60) for a reminder of these standard conversion methods.

### 13.1 DEFINITION OF POLYTIME MAPPING REDUCTIONS

Recall that in chapter 7, we defined reductions as follows:

$F$  reduces to  $G$

means (informally)

if you can solve  $G$ ,  
you can solve  $F$

The above definition ignores efficiency, and considers only computability. For that reason, this type of reduction is sometimes called a *Turing reduction*. We are now going to give a similar definition that takes efficiency into account:

$F$  polyreduces to  $G$

means (informally)

if you can solve  $G$  in polynomial time, you can solve  $F$  in polynomial time

The terminology here can produce ugly tongue twisters. Formally, this new type of reduction is called a “polynomial-time mapping reduction.” We will use the phrases “polynomial-time mapping reduction,” “polytime mapping reduction,” and “polyreduction” interchangeably. Similarly, the verb “polyreduce” will be used as a shorthand for both “perform a polyreduction” and “has a polyreduction.” And we will write  $F \leq_p G$  for “ $F$  polyreduces to  $G$ .” This echoes the notation for Turing reductions, since we write  $F \leq_T G$  for “ $F$  Turing-reduces to  $G$ .”

Recall that the definition of Turing reductions is rather flexible. If we are asked to prove that  $F$  reduces to  $G$ , we need to write a program  $F.py$  that solves  $F$ . At any point in the program, we can import and use a function  $G$  from  $G.py$ , which is assumed to solve problem  $G$ . (That is,  $G$  terminates with a correct solution on any input.) We are free to invoke the function  $G$  as often as we like, even inside a loop that might be executed exponentially often. And we can pass parameters to  $G$  that might be exponentially longer than the original input to  $F.py$ .

Once we begin considering efficiency and switch our attention to polytime reductions, there is much less flexibility in the appropriate ways to use function  $G$ . We are certainly still able to import and use  $G$  from  $G.py$ . And we can assume that  $G$  runs in polynomial time, as a function of the length of its input parameters. But beyond this, we have to be rather careful how we use  $G$ . If we invoke it too often, or send it parameters that are too long, the total running time of  $F.py$  might not be polynomial anymore.

For this reason, our definition of polytime reductions is much more restrictive. At the heart of the definition is a *mapping* of problem instances. Specifically, we need to map the original input (an instance of problem  $F$ ) into some new input (an instance of problem  $G$ ), and this mapping must run in polynomial time.

This leads us to a formal definition of polynomial-time mapping reductions:

**Definition of polyreduction.** A decision problem  $F$  has a *polynomial-time mapping reduction* to problem  $G$  (denoted  $F \leq_p G$ ) if there exists

- a polytime program  $C$  (for Convert) that converts instances of  $F$  into instances of  $G$ ,

such that

- $C$  maps positive instances of  $F$  to positive instances of  $G$ , and negative instances of  $F$  to negative instances of  $G$ .

(Note: As already commented above, we use *polyreduction* as an abbreviation of “polynomial-time mapping reduction.”) Mapping reductions

```

1  def convertPartitionToPacking(inString):
2      weights = [int(x) for x in inString.split()]
3      totalWeight = sum(weights)
4      if totalWeight % 2 == 1:
5          # if the total weight is odd, no partition is possible,
6          # so return any negative instance of Packing
7          return '0;1;1'
8      else:
9          # use thresholds that are half the total weight
10         targetWeight = str(int(totalWeight / 2))
11         return inString+';'+targetWeight+';'+targetWeight

```

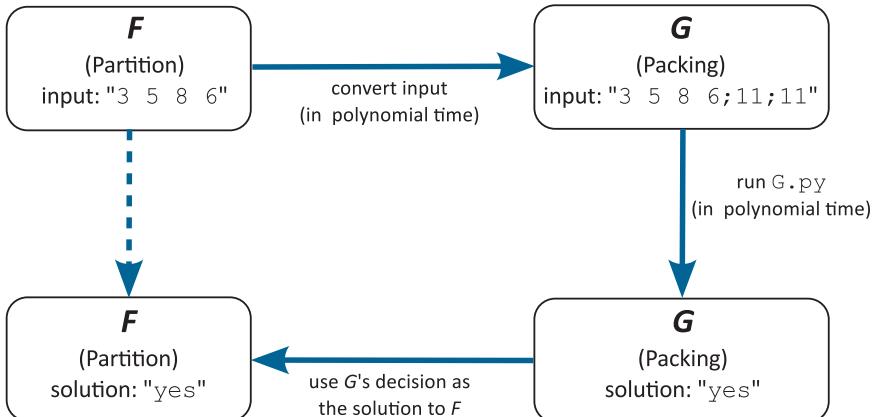
Figure 13.1: The Python program `convertPartitionToPacking.py`.

are also known as *Karp* reductions and *many-one* reductions. Polytime *Turing* reductions can also be defined, but they are not equivalent to our polytime mapping reductions. We don't study polytime Turing reductions in this book.

Let's see how the above definition of polyreduction works in practice, by polyreducing PARTITION to PACKING. Recall from figure 12.6 (page 257) that PACKING takes a set of integer weights  $w_i$  as input, and it receives two additional integers  $L, H$ , which are the low and high threshold values. The solution is "yes" if there's a subset of the weights that sums to a value between  $L$  and  $H$ , or "no" if this is impossible. PARTITION also takes a set of integer weights as input, but there are no threshold values. Instead, the solution is "yes" if the weights can be partitioned into two sets of equal total weight, and "no" otherwise. (Don't forget that all problems in the next two chapters are regarded as decision problems unless noted otherwise. That's why "yes" and "no" are the only solutions considered here.)

So, how do we polyreduce PARTITION to PACKING? The idea is simple. Suppose we are given a PARTITION instance with package weights  $w_1, w_2, \dots, w_m$ . We want to separate the packages into two piles of equal weight. Therefore, the weight of each pile must be half of the total weight. Thus, we set both threshold values of PACKING to be half of the sum of the weights. More precisely, we first calculate the total weight  $S = \sum_{k=1}^m w_k$ . If  $S$  is odd, then it's clearly impossible to split the packages into two equal piles. So we can convert the PARTITION instance into *any* negative instance of PACKING, such as "0;1;1". If  $S$  is even, set  $L = H = S/2$ , and convert the input into the new PACKING instance  $w_1, w_2, \dots, w_m; L; H$ . The code for this conversion is given in figure 13.1. This program is in  $O(n)$ , so certainly runs in polynomial time. Also, the returned instance of PACKING is positive if and only if the input instance of PARTITION was positive.

Figure 13.2 gives a concrete example of this polyreduction, demonstrating that *if* PACKING has a polytime method of solution then PARTITION also does. The PARTITION instance in this example is "3 5 8 6". When we run the conversion program (figure 13.1) this instance gets converted to the PACKING instance



**Figure 13.2: An example of a polynomial-time mapping reduction.** We can achieve the effect of following the dashed arrow (i.e., solving  $F$ ) by following the three solid arrows. *Top arrow:* Problem  $F$  (PARTITION) has its input converted into an equivalent input for problem  $G$  (PACKING), in polynomial time. *Right arrow:* Problem  $G$  is solved using a program  $G.py$ , which is assumed to exist and run in polynomial time. *Bottom arrow:*  $G$ 's output is used as the solution to  $F$ .

“3 5 8 6 ;11;11”. By assumption, this PACKING instance can be solved by running the polynomial-time program  $G.py$ . (Of course,  $G.py$  need not really exist. It is *assumed* to exist for the purposes of proving the polyreduction.) In this example,  $G.py$  returns the value “yes”, because a feasible packing exists (e.g., we can choose  $5 + 6 = 11$ ). Therefore, we can map this solution back to  $F$ , outputting “yes” as the solution for  $F$  also. This is guaranteed to work correctly, because the original conversion from  $F$  to  $G$  maps positive instances to positive instances, and negative instances to negative instances.

### Polyreducing to nondecision problems

Although we are primarily considering decision problems in the next two chapters, it’s worth noting a subtlety in the definition of a polyreduction on page 273. It’s trivial to extend this definition to problems  $G$  that are general problems ( $F$ , however, should still be a decision problem). In fact, the definition applies without any changes whatsoever. But we would need a slight change in the workings of figure 13.2: the bottom arrow still maps “no” to “no”, but now it maps *any* positive solution to “yes”. This extended definition will be used in our definition of NP-hardness on page 298.

## 13.2 THE MEANING OF POLYNOMIAL-TIME MAPPING REDUCTIONS

Figure 13.3 will help us understand the meaning and consequences of polyreductions. We start by recalling three important properties of Turing reductions, listed as T1, T2, and T3 in the left column of figure 13.3. For each of these

Turing reductions	Polyreductions
(T1) If $F$ Turing-reduces to $G$ , then $F$ is “no harder” than $G$ . The notation $F \leq_T G$ reminds us of this.	(P1) If $F$ polyreduces to $G$ , then $F$ can be solved “as fast as” $G$ (neglecting any polynomial slowdown factor). The notation $F \leq_P G$ reminds us of this.
(T2) If $F$ Turing-reduces to $G$ , and $G$ is computable, then $F$ is computable.	(P2) If $F$ polyreduces to $G$ , and $G$ is in Poly, then $F$ is in Poly.
(T3) If $F$ Turing-reduces to $G$ , and $F$ is uncomputable, then $G$ is uncomputable.	(P3) If $F$ polyreduces to $G$ , and $F$ requires superpolynomial time, then $G$ requires superpolynomial time.

**Figure 13.3:** Left: Three familiar properties of Turing reductions (T1, T2, T3). Right: The corresponding properties for polynomial-time mapping reductions (P1, P2, P3).

properties, there is an intuitively similar property for polyreductions, listed as P1, P2, and P3 in the right column of figure 13.3. It is easy to prove properties P1–P3, as follows:

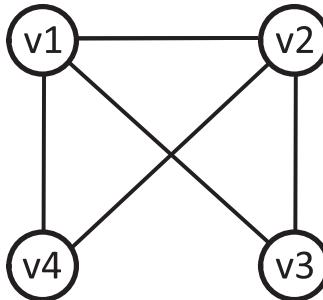
- P1 follows immediately from the definition of polyreductions. To see this concretely, follow the solid arrows in figure 13.2 and note that the top solid arrow requires an additional polynomial factor only.
- P2 follows from essentially the same reasoning: to solve  $F$  in polynomial time, we just follow the solid arrows in figure 13.2. Now *every* step takes only polynomial time, since the statement of P2 guarantees that  $G$  has a polytime method of solution.
- P3 is logically equivalent to P2 (by taking the contrapositive), but for increased understanding let’s prove it directly by contradiction. So, suppose  $G$  can be solved in polynomial time. Then by following the arrows in figure 13.2, we have a way of solving  $F$  in polynomial time, and this contradicts the fact that  $F$  requires superpolynomial time.

### 13.3 PROOF TECHNIQUES FOR POLYREDUCTIONS

Proving that one problem polyreduces to another is one of the most important skills you should learn from this book, so we provide a brief overview of the technique before presenting further examples. We’ve already seen one example of a polyreduction proof, summarized in figures 13.1 and 13.2. But these figures provide much more detail than necessary for a typical polyreduction proof. Usually, we assume the reader of our proof understands the overall framework of figure 13.2, so it’s not necessary to explain it. Instead, we give details only of the conversion algorithm  $C$  corresponding to the top arrow of figure 13.2.

**PROBLEM HAMILTONCYCLE (UHC)**

- **Input:** An undirected, unweighted graph, formatted as in figure 4.3 (page 48). For example, the graph below would be input as “v1, v2 v2, v3 v3, v1 v2, v4 v1, v4”.
- **Solution:** “yes” if the input graph has a Hamilton cycle (as defined on page 47), and “no” otherwise. For example, the graph below has the Hamilton cycle “v1, v4, v2, v3”, so the solution is “yes” for that graph.



**Figure 13.4:** Description of the computational problem (undirected) HAMILTONCYCLE, also known as UHC.

Algorithm  $C$  can be described using a real computer program (as in figure 13.1), but it’s more usual to give a prose description of  $C$ . Either way, it’s essential to then prove several facts about  $C$ . Specifically, we must prove that

- $C$  maps positive instances to positive instances;
- $C$  maps negative instances to negative instances;
- $C$  runs in polynomial time.

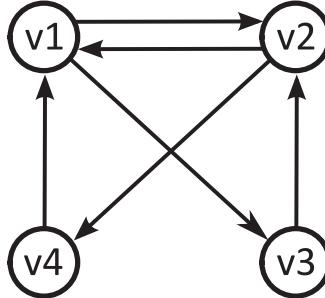
Important note: For part (b) of the proof, it’s often easier to prove the contrapositive version of statement (b). Specifically, instead of the statement “If  $I$  is negative, then  $C(I)$  is negative,” we prove the equivalent statement “If  $C(I)$  is positive, then  $I$  is positive.” Several examples of this approach are given later, such as in the proofs of the next two claims.

## 13.4 EXAMPLES OF POLYREDUCTIONS USING HAMILTON CYCLES

The problems HAMILTONCYCLE (also known as UNDIRECTEDHAMILTONCYCLE or UHC) and DIRECTEDHAMILTONCYCLE (DHC) provide some good examples of polyreductions. The basic idea is, given a (directed or undirected) graph, to find a (directed or undirected) cycle that visits every node exactly once. Figures 13.4 and 13.5 describe these problems in detail. These problems don’t have many direct real-world applications, but they are simplifications of important and practical problems such as TSP. For that reason, it’s important to understand the complexity of UHC and DHC.

**PROBLEM DIRECTEDHAMILTONCYCLE (DHC)**

- **Input:** A directed graph, formatted as in figure 4.3 (page 48). For example, the graph below would be input as “v1, v2 v2, v1 v3, v2 v1, v3 v2, v4 v4, v1”.
- **Solution:** “yes” if the input graph has a directed Hamilton cycle, and “no” otherwise. A directed Hamilton cycle is a directed cycle that contains each node exactly once. For example, the graph below has the Hamilton cycle “v1, v3, v2, v4”, so the solution is “yes” for that graph.



**Figure 13.5:** Description of the computational problem DHC.

### A polyreduction from UHC to DHC

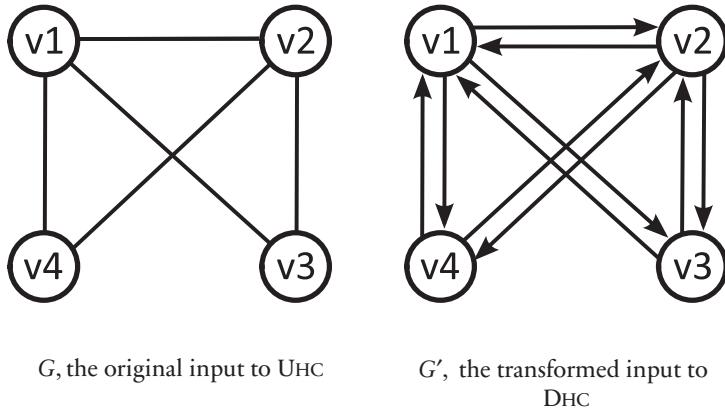
As the following claim shows, there is a simple polyreduction from the undirected Hamilton cycle problem to the directed version.

**Claim 13.1.**  $\text{UHC} \leq_p \text{DHC}$ .

*Proof of the claim.* Let  $I$  be an arbitrary input string for UHC. We need to convert this into an equivalent input string  $I'$  for DHC. It's trivial to convert between graphs and their string representations, so instead of converting  $I$  to  $I'$ , let's instead aim to convert an undirected graph  $G$  into a new directed graph  $G'$ . The key trick is to change each edge of  $G$  into two edges of  $G'$ —one in each direction. For example, if  $G$  was the graph shown in the left panel of figure 13.6, then  $G'$  would be the graph shown in the right panel. If we translated this example into ASCII strings, we would have

$$\begin{aligned} I &= \text{“v1, v2 v2, v3 v3, v1 v2, v4 v1, v4”}, \\ I' &= \text{“v1, v2 v2, v1 v2, v3 v3, v2 v3, v1 v1, v3} \\ &\quad \text{v2, v4 v4, v2 v1, v4 v4, v1”}. \end{aligned}$$

There is a minor technicality here. If  $G$  consists of exactly 2 nodes connected by 1 edge, then our trick doesn't work:  $G$  has no Hamilton cycle, but the transformed graph with 2 edges would have a Hamilton cycle from one node to the other and back again. Therefore, we treat 2-node graphs as a special case, transforming the single edge into a single directed edge.

**Figure 13.6:** An example of polyreducing UHC to DHC.

We need to prove that (a) if  $G$  has an undirected Hamilton cycle, then  $G'$  has a directed Hamilton cycle; and (b) if  $G$  has no undirected Hamilton cycle, then  $G'$  has no directed Hamilton cycle. For (a), it's clear that any Hamilton cycle  $c$  in  $G$  translates directly to the same cycle (but now with directed edges) in  $G'$ . For example, in figure 13.6, the  $G$ -cycle “ $v_1, v_4, v_2, v_3$ ” becomes the directed  $G'$ -cycle “ $v_1, v_4, v_2, v_3$ ”. For (b), it's easier to prove the contrapositive statement: we claim that if  $G'$  has a directed Hamilton cycle, then  $G$  has an undirected Hamilton cycle. Again, there is a direct translation: if  $c$  is a directed cycle in  $G'$ , then exactly the same undirected edges must exist in  $G$ , so the same cycle (now undirected) exists in  $G$ . For example, if  $c$  is the  $G'$ -cycle “ $v_2, v_4, v_1, v_3$ ”, then “ $v_2, v_4, v_1, v_3$ ” is also a  $G$ -cycle. (Technicality: This argument works only for cycles of 3 or more nodes, but recall that we treated the 2-node case separately.)

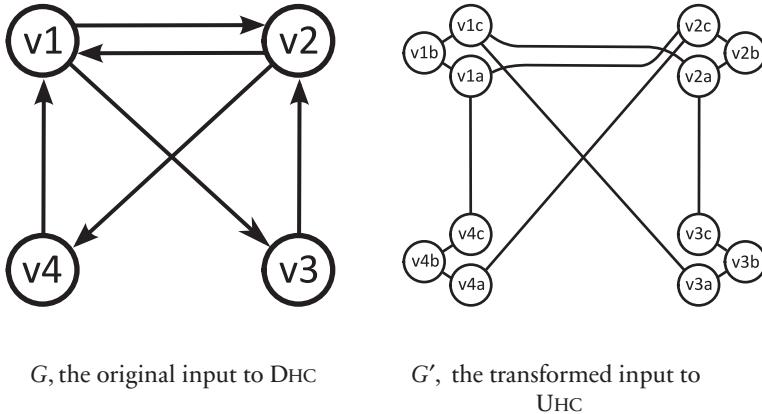
Finally, we show that this transformation can be done in polynomial time. First note that the ASCII form of  $G'$  is twice as long as  $G$ , since it contains twice as many edges. And we can compute  $G'$  from  $G$  from a single scan of the description of  $G$ , by copying and reversing each edge that we encounter. So the transformation requires only  $O(n)$  time, which is certainly polynomial. (It would be an easy and useful exercise to write this program for yourself. One possible implementation is provided with the book resources as `convertUHC-toDHC.py`.)  $\square$

## A polyreduction from DHC to UHC

Interestingly, we can go back the other way: there is a polyreduction from DHC to UHC too. The reduction in this direction is a little more elaborate, but it involves a beautiful idea.

**Claim 13.2.**  $DHC \leq_p UHC$ .

*Proof of the claim.* As in the proof of the previous claim, we work with graphs  $G, G'$  instead of their string representations  $I, I'$ . This time we are given a *directed* graph  $G$ , such as the one in the left panel of figure 13.7. We need to convert  $G$

**Figure 13.7:** An example of polyreducing DHC to UHC.

into an undirected graph  $G'$ , such that  $G'$  has an undirected Hamilton cycle if and only if  $G$  has a directed Hamilton cycle.

Here's how the conversion works. Each node  $v$  of  $G$  becomes three nodes in  $G'$ , labeled  $v_a, v_b, v_c$ . Let's call  $v_a, v_b, v_c$  the *children* of  $v$  (and similarly for the other nodes). Each triplet of children with the same parent is connected by an edge between the  $a$ -child and  $b$ -child, and between the  $b$ -child and  $c$ -child. And every directed edge in  $G$ —say, from node  $v$  to node  $w$ —is transformed into a corresponding undirected edge in  $G'$ , between the  $c$ -child of  $v$  and the  $a$ -child of  $w$ .

As an example of this construction, suppose  $G$  is the graph in the left panel of figure 13.7. The string description of  $G$  is “ $v1, v2 \rightarrow v2, v1 \rightarrow v3, v2 \rightarrow v1, v3 \rightarrow v2, v4 \rightarrow v4, v1$ ”. The resulting  $G'$  is in the right panel. Node  $v1$  becomes  $v1a, v1b, v1c$ , and similarly for the other nodes. Edges “ $v1a, v1b$ ” and “ $v1b, v1c$ ” are inserted, and similarly for the other triplets. Finally each directed edge in  $G$  (e.g., “ $v2, v4$ ”) is converted into a corresponding  $c-a$  edge in  $G'$  (e.g., “ $v2c, v4a$ ”).

We need to prove that (a) if  $G$  has a directed Hamilton cycle, then  $G'$  has an undirected Hamilton cycle, and (b) if  $G$  has no directed Hamilton cycle, then  $G'$  has no undirected Hamilton cycle. For (a), suppose we have a Hamilton  $G$ -cycle, containing the nodes  $u, v, w, \dots$ . Then  $G'$  contains the Hamilton cycle  $u_a, u_b, u_c, v_a, v_b, v_c, w_a, w_b, w_c, \dots$ . For a specific example based on figure 13.7, the cycle beginning  $v4, v1, \dots$  is mapped to  $v4a, v4b, v4c, v1a, v1b, v1c, \dots$ .

For (b), it's easier to prove the contrapositive statement: if  $G'$  contains an undirected Hamilton cycle, then  $G$  contains a directed Hamilton cycle. To see this, first note that any Hamilton cycle in  $G'$  must have a very specific form. Starting from any  $b$ -child (e.g.,  $v1b$  in figure 13.7), we can choose to follow the cycle in either direction—towards the corresponding  $a$ -child or  $c$ -child ( $v1a$  or  $v1c$  in our running example). Let's choose to follow it in the direction of the  $c$ -child. Then, the cycle must proceed to the  $a$ -child of a new triplet, visiting all other triplets in  $a, b, c$  order, eventually returning to the  $a$ -child of the original triplet. Thus, any Hamilton cycle in  $G'$  has the form  $u_b, u_c, v_a, v_b, v_c, w_a, w_b,$

$w_c, \dots$ . (In our running example, this could be v1b, v1c, v3a, v3b, v3c, ...) And any Hamilton cycle of this form in  $G'$  corresponds to a directed Hamilton cycle in  $G$ , obtained by mapping triplets back to their parents:  $u, v, w, \dots$  (In our running example, this would be v1, v3, ...)

Finally, we claim this conversion runs in polynomial time. The number of nodes and edges increases by at most a factor of 3, and the edges to be added can be determined by a single scan through the description of  $G$ . So the conversion runs in  $O(n)$  time, which is certainly polynomial.  $\square$

As usual, it would be an excellent idea to implement the conversion from  $G$  to  $G'$  yourself, as a Python program. An implementation is also provided with the book materials as `convertDHCToUHC.py`.

## 13.5 THREE SATISFIABILITY PROBLEMS: CIRCUITSAT, SAT, AND 3-SAT

This section describes three decision problems that are essential for understanding the “hardness” of many PolyCheck problems: CIRCUITSAT, SAT, and 3-SAT. All three belong to the general category of *satisfiability* problems.

### Why do we study satisfiability problems?

When you first encounter them, satisfiability problems seem abstract, and it is hard to understand why they are so central to complexity theory. But there are at least three reasons we study these problems. First, satisfiability problems have many practical applications, including planning and scheduling (e.g., air-traffic control, crop rotation), hardware and software verification, drug design, and genetics (e.g., identifying haplotypes from genotypes). Second, SAT has an important role in the history of complexity theory, since (as we will discover in the next chapter) it was the first problem proved to be NP-complete. Third, the satisfiability problems are all about Boolean algebra. Because computers are made of circuits, and circuits are a physical representation of Boolean algebra, there is a surprisingly direct connection between satisfiability and computers, especially for CIRCUITSAT, which is discussed next.

### CIRCUITSAT

CIRCUITSAT is also known as *circuit satisfiability*. The informal idea behind this problem is that we are given an electrical circuit with a large number of inputs and one output. The circuit is a *Boolean* circuit, which means that the inputs and output are binary: 0 or 1. Then we ask the question, is it possible for the output of the circuit to be 1? In more detail, we ask, is there any setting of the inputs that produces an output of 1? Any setting that does produce a 1 is said to *satisfy* the circuit, which explains why this problem is called circuit satisfiability. Note that no previous knowledge of circuits is required for understanding the material in this book. The circuits involved in CIRCUITSAT are very simple, and everything you need to know about these circuits will now be explained.

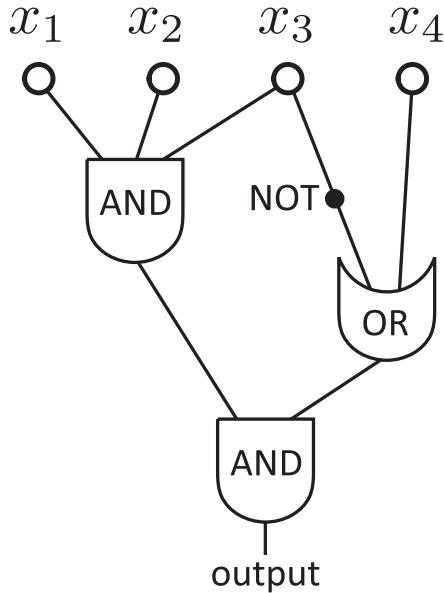


Figure 13.8: A CIRCUITSAT instance.

Let's fix some more formal notation. There will be  $k$  binary inputs labeled  $x_1, x_2, \dots, x_k$ . The circuit will consist of *wires* and three types of *gates*: AND gates, OR gates, and NOT gates. AND gates output a 1 only if *all* the inputs are 1. OR gates output a 1 if *any* input is 1. A NOT gate reverses its single input (0 becomes 1, and 1 becomes 0).

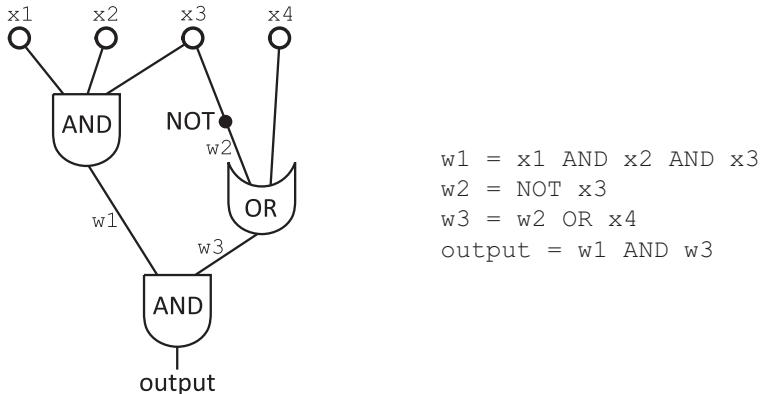
Figure 13.8 gives an example. You should check for yourself that we can make the output 1 by setting all inputs to 1. In this case, no other setting satisfies the circuit. But in general, there can be zero, one, or many ways of satisfying a circuit.

Of course, a formal definition of a computational problem requires the input to be a string. We can agree on how to describe CIRCUITSAT instances using any reasonable conventions. One simple way is as follows. First, we write  $x_1, x_2, \dots$  in ASCII as  $x1, x2, \dots$ . Then, we label all the connecting wires with arbitrary unique labels such as  $w1, w2, \dots$ . The new labels are shown in the left panel of figure 13.9. Finally, we write out an explicit computation for each wire in ASCII, as in the right panel of figure 13.9. Figure 13.10 summarizes this discussion with a formal definition of CIRCUITSAT.

## SAT

The famous problem SAT, also known as *satisfiability*, is similar to CIRCUITSAT. The main difference is that we deal with Boolean *formulas* instead of Boolean circuits. As with circuits, no prior knowledge is required. We will now give a complete description of the simple Boolean formulas used by SAT.

Any Boolean formula contains some number (say  $k$ ) of Boolean variables, denoted  $x_1, x_2, \dots, x_k$ . By definition, each Boolean variable can take the values



**Figure 13.9:** *Left:* The CIRCUITSAT instance of figure 13.8, with extra labels so that it can be converted into ASCII. *Right:* The same CIRCUITSAT instance represented as an ASCII string.

### PROBLEM CIRCUITSAT

- **Input:** An ASCII description of a Boolean circuit  $C$  that has a single output. The format of the input string is explained in figure 13.9.
- **Solution:** “yes” if  $C$  is satisfiable (i.e., it is possible to produce the output 1), and “no” otherwise.

**Figure 13.10:** Description of the computational problem CIRCUITSAT.

1 or 0, which we can think of as “true” and “false” respectively. The Boolean variables are combined using the symbols  $\wedge$ ,  $\vee$ ,  $\neg$ , which stand for AND, OR, NOT respectively. These symbols operate as you would expect:  $a \wedge b$  is true when  $a$  and  $b$  are both true;  $a \vee b$  is true when  $a$  is true or  $b$  is true;  $\neg a$  is true when  $a$  is false. Parentheses are used to eliminate ambiguity.

Here’s an example of a Boolean formula:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (x_5). \quad (\star)$$

In this example, we can make the formula true by taking  $x_1 = 0$ ,  $x_2 = 0$ ,  $x_3 = 0$ ,  $x_4 = 1$ ,  $x_5 = 1$ . We say that this assignment of the variables *satisfies* the formula. It turns out that there are other ways of satisfying this formula (e.g., it remains true when we switch  $x_1$  to 1, and/or  $x_4$  to 0). In general, there can be zero, one, or many ways of satisfying a given formula.

As you have probably guessed already, SAT takes a Boolean formula  $B$  as input and asks, is it possible to satisfy  $B$ ? But there is an important technical restriction, discussed next.

### PROBLEM SAT

- **Input:** An ASCII description of a Boolean formula  $B$  in conjunctive normal form. An example of the format of the input string is given in equation (†) on this page.
- **Solution:** “yes” if  $B$  is satisfiable (i.e., it is possible to produce the value 1), and “no” otherwise. The solution is also “no” if the input is not a correctly formatted Boolean formula in CNF.

### PROBLEM 3-SAT

3-SAT is identical to SAT, except that all clauses in the input can have at most 3 literals. (If the input is a CNF formula with a clause that’s too big, the solution is “no”.)

**Figure 13.11:** Description of the computational problems SAT and 3-SAT.

## Conjunctive normal form

To be a legitimate instance of SAT, a Boolean formula must be in a special format that we call *conjunctive normal form*, or CNF. To define CNF, we will need some other terminology first. A *literal* is a variable by itself (e.g.,  $x_7$ ), or a negated variable by itself (e.g.,  $\neg x_3$ ). A *clause* is a collection of literals OR’ed together (e.g.,  $x_1 \vee x_2 \vee \neg x_3$ ). Finally, a formula is in CNF if it consists of a collection of clauses AND’ed together. For example, the formula (★) given above is in CNF, whereas the formula

$$(x_1 \wedge \neg x_2) \vee (x_2 \vee \neg x_3 \wedge \neg x_4)$$

is not in CNF.

## ASCII representation of Boolean formulas

It’s easy to represent Boolean formulas in ASCII. One simple convention would represent the formula (★) above as

$$(x1 \text{ OR } x2 \text{ OR } \text{NOT } x3) \text{ AND } (\text{NOT } x2 \text{ OR } x4) \text{ AND } (x5) \quad (\dagger)$$

Figure 13.11 gives a formal definition of SAT. You should also note that complexity theory textbooks are inconsistent in their definition of SAT: some follow our definition and insist that SAT instances are in CNF; but other books permit arbitrary Boolean formulas as input for SAT and define a separate CNFSAT problem for CNF formulas. The non-CNF variant of SAT polyreduces to CNFSAT, so for some purposes the distinction doesn’t matter.

### 3-SAT

3-SAT is identical to SAT, except that we have an even more stringent requirement on the types of inputs that are acceptable. Instead of merely insisting that the input is in CNF, 3-SAT insists that the input is in 3-CNF, which means that every clause in the CNF formula has no more than 3 literals. For example, formula (★) on page 283 is in 3-CNF, because the clauses have 3, 2, and 1 literals respectively. On the other hand, the following formula is in CNF but not 3-CNF:

$$(x_1 \vee x_2 \vee \neg x_3 \vee \neg x_5) \wedge (\neg x_2 \vee x_4) \wedge (x_5).$$

You can easily check that this formula is satisfiable. Thus, it's a positive instance of SAT, but a negative instance of 3-SAT.

Formal definitions of SAT and 3-SAT are given in figure 13.11. Note that we can give similar definitions for problems such as 2-SAT and 5-SAT—in each case, the name of the problem indicates the maximum number of literals permitted in a clause.

## 13.6 POLYREDUCTIONS BETWEEN CIRCUITSAT, SAT, AND 3-SAT

It turns out that there are reasonably simple polyreductions in both directions between any two of the following problems: CIRCUITSAT, SAT, and 3-SAT. The next few claims describe these polyreductions, without always giving completely rigorous proofs. To describe our first reduction, from CIRCUITSAT to SAT, we need a technique known as the *Tseytin transformation*.

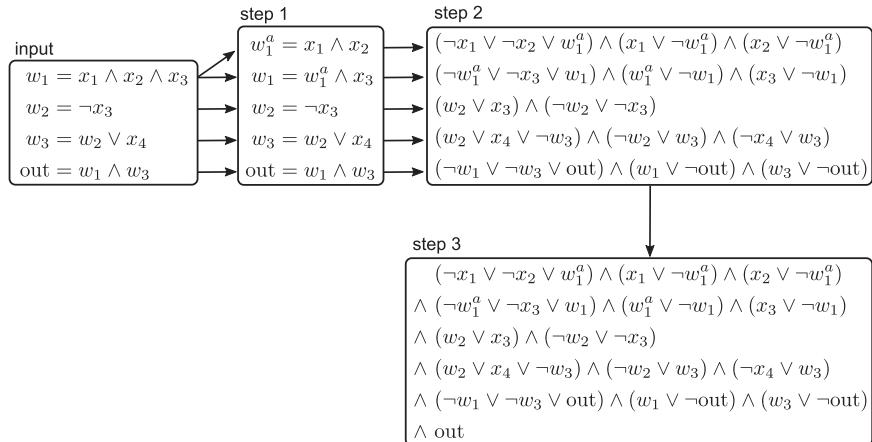
### The Tseytin transformation

The Tseytin transformation converts any Boolean circuit into a 3-CNF formula in linear time. The transformation consists of the following three steps, which are also illustrated in figure 13.12:

**Step 1.** Transform gates with three or more inputs into a sequence of gates with only two inputs each. For example, “ $w_1 = w_2 \text{ AND } w_3 \text{ AND } w_4 \text{ AND } w_5$ ” becomes

$$\begin{aligned} w_{1a} &= w_2 \text{ AND } w_3, \\ w_{1b} &= w_{1a} \text{ AND } w_4, \\ w_1 &= w_{1b} \text{ AND } w_5. \end{aligned}$$

A similar transformation can be applied to OR gates.



**Figure 13.12: An example of the Tseytin transformation.** The input represents the circuit in figure 13.9. To improve readability, formulas are written in mathematical notation rather than ASCII, and “output” is abbreviated to “out.”

**Step 2.** Transform each gate into an equivalent CNF expression, using the following templates for each type of gate:

Gate	Equivalent CNF formula
$w_1 = w_2 \text{ AND } w_3$	$(\neg w_2 \vee \neg w_3 \vee w_1) \wedge (w_2 \vee \neg w_1) \wedge (w_3 \vee \neg w_1)$
$w_1 = w_2 \text{ OR } w_3$	$(w_2 \vee w_3 \vee \neg w_1) \wedge (\neg w_2 \vee w_1) \wedge (\neg w_3 \vee w_1)$
$w_1 = \text{NOT } w_2$	$(w_1 \vee w_2) \wedge (\neg w_1 \vee \neg w_2)$

These formulas may seem a little magical when you first see them, but there is nothing mysterious here. By checking each possible value for  $w_1$ ,  $w_2$ ,  $w_3$ , you can verify that each CNF formula is true exactly when the corresponding gate’s output is correct. As an example, let’s verify the third line of the table. Write  $f = (w_1 \vee w_2) \wedge (\neg w_1 \vee \neg w_2)$ . By checking the 4 possibilities for  $w_1$  and  $w_2$ , we find that  $f = 1$  if and only if  $w_1 = \neg w_2$ , as required.

**Step 3.** Join all the CNF formulas from the previous step with AND’s, producing a single CNF formula. Then, AND this with a final clause containing the single literal “output”. This produces our final Boolean formula in conjunctive normal form.

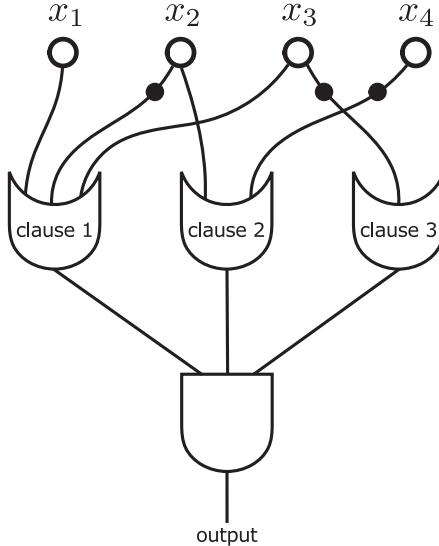
Verify your understanding of the Tseytin transformation by checking the example in figure 13.12. With this understanding, we are ready to polyreduce CIRCUITSAT to SAT:

**Claim 13.3.** CIRCUITSAT  $\leq_p$  SAT.

*Proof of the claim.* Let  $C$  be an instance of CIRCUITSAT; that is,  $C$  is an ASCII string describing a Boolean circuit. We need to convert  $C$  into a Boolean formula  $B$  in CNF, such that  $B$  is satisfiable if and only if  $C$  is satisfiable. We do this by applying the Tseytin transformation described above.

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge \neg x_3$$

clause 1                    clause 2                    clause 3



**Figure 13.13: An example of converting a SAT instance into a CIRCUITSAT instance.**  
The boxed CNF formula is satisfiable if and only if the circuit below it is satisfiable.

We claim that  $B$  is a satisfiable formula if and only if  $C$  is a satisfiable circuit. Break this down as (a)  $B$  is satisfiable implies  $C$  is satisfiable, and (b)  $C$  is satisfiable implies  $B$  is satisfiable. For (a), take a satisfying assignment for  $B$  and copy the input ( $x_i$ ) values from  $B$  to the corresponding inputs for  $C$ . We know all the gates behave correctly because of the transformation in step 2 above, and the output equals 1 because of the final clause (see, for example, the last line of the “step 3” box in figure 13.12). So this is a satisfying assignment for  $C$ . For (b), we simply copy all values in a satisfying assignment for  $C$  into the corresponding variables for  $B$ , and note that each clause is satisfied.

Finally, we claim that the transformation from  $C$  to  $B$  runs in polynomial time. In fact, it’s easy to check that each of the three steps above executes in linear time, so the entire transformation runs in linear time.  $\square$

Because the Tseytin transformation always results in a 3-CNF formula, the above proof in fact shows that  $\text{CIRCUITSAT} \leq_p \text{3-SAT}$ .

Next, we reverse our previous claim and polyreduce SAT to CIRCUITSAT:

**Claim 13.4.**  $\text{SAT} \leq_p \text{CIRCUITSAT}$ .

*Proof of the claim.* Given a Boolean formula  $B$  in CNF, we can easily convert it into an equivalent circuit  $C$ . Because  $B$  is in CNF, we know the precise structure of the circuit to build: each clause becomes an OR gate, with a wire feeding down to a single AND that produces the output. It’s easy to see that the description of this

circuit is the same size as the original formula (ignoring multiplicative constants), so the conversion process easily runs in polynomial time. It's also clear that  $B$  is satisfiable if and only if  $C$  is satisfiable: a satisfying assignment for one translates without alteration into a satisfying assignment for the other. Figure 13.13 gives an example of this transformation.  $\square$

Proving that 3-SAT polyreduces to SAT is very easy, but for completeness we formalize this as a claim:

**Claim 13.5.** 3-SAT  $\leq_P$  SAT.

*Proof of the claim.* This is a trivial reduction. Given a 3-SAT formula  $B$  in 3-CNF, simply observe that  $B$  is also a SAT formula. So no conversion operation is required, and it follows immediately that  $B$  is a positive instance of 3-SAT if and only if it's a positive instance of SAT.  $\square$

Before tackling our next reduction, we need a new tool for working with Boolean formulas:

**Definition of “splitting a clause.”** Let  $C$  be a clause in a Boolean CNF formula, and suppose  $C$  contains at least 4 literals. Letting  $k$  denote the number of literals in  $C$ , we can write

$$C = l_1 \vee l_2 \vee \cdots \vee l_{k-1} \vee l_k,$$

where  $k \geq 4$ . Recall that each literal  $l_i$  represents some variable  $x_j$  or its negation  $\neg x_j$ . Let  $d$  be a new “dummy” variable that is not present in the original Boolean formula. Then we can *split*  $C$  using  $d$ , obtaining two new clauses  $C'$ :

$$C' = (l_1 \vee l_2 \vee \cdots \vee l_{k-2} \vee d) \wedge (\neg d \vee l_{k-1} \vee l_k).$$

For example, if

$$C = x_1 \vee x_3 \vee \neg x_5 \vee \neg x_7 \vee x_8, \tag{\star}$$

then the result of splitting  $C$  using  $d$  would be

$$C' = (x_1 \vee x_3 \vee \neg x_5 \vee d) \wedge (\neg d \vee \neg x_7 \vee x_8). \tag{\dagger}$$

Note that the clauses in  $C'$  are always shorter than the original clause  $C$ . Specifically, if  $C$  contains  $k$  literals, then the two clauses in  $C'$  contain  $k-1$  and 3 literals respectively. This technique of splitting clauses to shorten them plays a crucial role in the next reduction. But we need to understand another property of clause splitting first: as the next claim shows, splitting a clause preserves satisfiability.

**Claim 13.6.** Let  $C$  be a clause as in the above definition, and let  $C'$  be the two-clause formula that results from splitting  $C$  using  $d$ . Then any satisfying assignment for  $C$  corresponds to a satisfying assignment for  $C'$ , and vice versa.

Moreover, the values of all variables other than the dummy variable are the same in the corresponding assignments.

*Proof of the claim.* First we show that a satisfying assignment for  $C$  leads to a satisfying assignment for  $C'$ . In any satisfying assignment for  $C$ , at least one of the literals  $l_i$  is true. Choose one of these true literals, say  $l_i$ , and consider two cases: either  $l_i$  is in the first clause of  $C'$ , or it's in the second clause of  $C'$ . If it's in the first clause, we set  $d = 0$ , leave all other variables the same, and thus obtain a satisfying assignment for  $C'$ . If  $l_i$  is in the second clause, we set  $d = 1$ , leave all other variables the same, and again obtain a satisfying assignment for  $C'$ .

Second, we show that a satisfying assignment for  $C'$  leads to a satisfying assignment for  $C$ . In any satisfying assignment for  $C'$ , there are two cases: either  $d = 0$  or  $d = 1$ . If  $d = 0$ , then one of the  $l_i$  in the first clause is true. If  $d = 1$ , then one of the  $l_i$  in the second clause is true. So in either case, at least one of the  $l_i$  is true. So we can leave the values of all variables the same, and obtain a satisfying assignment for  $C$ .  $\square$

As a practical example of this, consider  $C$  and  $C'$  defined in equations  $(\star)$  and  $(\dagger)$  above. One particular satisfying assignment for  $C$  is  $x_1 = x_3 = 0$ ,  $x_5 = x_7 = x_8 = 1$ . When we map this assignment into  $C'$ , the second clause of  $C'$  is immediately satisfied by  $x_8$ . None of the literals  $x_1, x_3, \neg x_5$  satisfies the first clause, but we can set  $d = 1$  to achieve this.

Now we are ready for our next polyreduction.

**Claim 13.7.**  $\text{SAT} \leq_P \text{3-SAT}$ .

*Proof of the claim.* Let  $B$  be a Boolean CNF formula, that is, an instance of SAT. We need to convert  $B$  into  $B'$ , an equivalent instance of 3-SAT. The main issue here is that  $B$  could have clauses with many literals, whereas the clauses of  $B'$  are permitted to have only 3 literals at most. So we use the splitting process described above to achieve the conversion. This works iteratively, generating a sequence of formulas  $B_1, B_2, \dots, B_m$ , where  $B_1 = B$  and  $B_m = B'$ . The procedure is as follows: First, scan  $B_1$  to find the first clause that contains 4 or more literals. Split this clause using a new dummy variable  $d_1$ , resulting in a new Boolean formula  $B_2$ . Next scan  $B_2$ , split the first long clause using  $d_2$ , obtaining  $B_3$ . Iterate until all clauses have at most 3 literals. The algorithm is guaranteed to terminate because splitting a clause always generates two shorter clauses.

We also claim the algorithm runs in polynomial time. To see this, note that the total number of splits needed is bounded by the length of  $B$ ; that is, it is in  $O(n)$ . The length of each formula  $B_i$  is also in  $O(n)$ , so each splitting iteration requires  $O(n)$  time. In other words, we have  $O(n)$  iterations of an  $O(n)$  operation, requiring at most  $O(n^2)$  total time, which is certainly polynomial.

Finally, note that each  $B_i$  is satisfiable if and only if  $B_{i-1}$  is satisfiable. (This follows from the previous claim, where we showed that splitting clauses preserves satisfiability.) Hence,  $B$  is satisfiable if and only if  $B'$  is satisfiable.  $\square$

The splitting technique in this proof is an important tool that you should understand. But it's worth noting that we could have avoided the splitting technique and instead proved this claim by combining two of our earlier reductions. Given  $B$ , we first convert it into a circuit  $C$  using the construction

for reducing SAT to CIRCUITSAT (claim 13.4, page 287). Then convert  $C$  into a new Boolean formula  $B'$  using the Tseytin transformation for polyreducing CIRCUITSAT to SAT (claim 13.3, page 286). As mentioned earlier, it's easy to check that the Tseytin transformation always produces a 3-CNF formula.

## 13.7 POLYEQUIVALENCE AND ITS CONSEQUENCES

By now we are familiar with the fact that the composition of polynomials results in another polynomial. Applying this fact to polyreductions, we can immediately deduce that the composition of two polyreductions results in another polyreduction. This proves the following claim:

**Claim 13.8.** Let  $F, G, H$  be decision problems with  $F \leq_p G$  and  $G \leq_p H$ . Then  $F \leq_p H$ .

We often find that there are polyreductions in *both* directions between two problems. In this case, we say the problems are polyequivalent, as in the following formal definition:

**Definition of polyequivalence.** Let  $F, G$  be decision problems with  $F \leq_p G$  and  $G \leq_p F$ . Then we say  $F$  and  $G$  are *polyequivalent*, and write  $F \equiv_p G$ .

For example, claims 13.1 and 13.2 (pages 278 and 279) showed that UHC  $\leq_p$  DHC and DHC  $\leq_p$  UHC, so we can conclude  $UHC \equiv_p DHC$ . Similarly, the other polyreductions in this chapter show that CIRCUITSAT  $\equiv_p$  SAT and SAT  $\equiv_p$  3-SAT.

It follows immediately from the previous claim that polyequivalence is transitive; that is, if  $F \equiv_p G$  and  $G \equiv_p H$ , then  $F \equiv_p H$ . Applying this to the SAT variants, we conclude that all three variants are polyequivalent:

$$\text{CIRCUITSAT} \equiv_p \text{SAT} \equiv_p \text{3-SAT}. \quad (\bullet)$$

In other words, there are polyreductions in both directions between any two of the above three problems.

Note that our three satisfiability problems (CIRCUITSAT, SAT, and 3-SAT) are all in PolyCheck. This follows because if we are given a proposed satisfying assignment as a hint, it's easy to substitute the proposed values and thus check in polynomial time whether the assignment does in fact satisfy the given circuit or formula.

You won't be surprised to hear that no one knows a polytime algorithm for solving any of the three satisfiability problems. But because of the polyreductions summarized in equation  $(\bullet)$  above, we do now have some important information: either all three problems (CIRCUITSAT, SAT, and 3-SAT) can be solved in polynomial time, or none of them can be solved in polynomial time. The full significance of this will become clear only after our study of NP-completeness in the next chapter.

It's worth noting one additional quirk relating to the satisfiability problems: there *is* a polynomial-time algorithm solving 2-SAT. So, the 2-SAT variant of SAT

seems to be strictly easier than the others, and it's widely believed that 2-SAT and 3-SAT are *not* polyequivalent.

## EXERCISES

**13.1** Suppose  $A$  is a decision problem and  $B$  is a general computational problem such that  $A \leq_p B$ . Which of the following statements must be true?

1. If  $A \in \text{Poly}$ , then  $B \in \text{Poly}$ .
2. If  $B \in \text{Poly}$ , then  $A \in \text{Poly}$ .
3.  $A$  and  $B$  are both members of  $\text{Poly}$ .
4. If  $A$  requires exponential time, then  $B$  requires exponential time.
5. If  $B$  requires exponential time, then  $A$  requires exponential time.
6.  $A$  and  $B$  both require exponential time.
7.  $B$  is a decision problem.

**13.2** Suppose that  $A$  and  $B$  are decision problems and it is known that both  $A \leq_p B$  and  $B \leq_p A$ . Which of the following statements must be true?

1.  $A$  and  $B$  are both members of  $\text{Poly}$ .
2.  $A$  and  $B$  both require superpolynomial time.
3. Either both problems ( $A$  and  $B$ ) are in  $\text{Poly}$ , or both require superpolynomial time.
4. Either both problems ( $A$  and  $B$ ) are in  $\text{Poly}$ , or both are outside  $\text{Poly}$ .
5. Either both problems ( $A$  and  $B$ ) are in  $\text{Poly}$ , or both are in  $\text{Expo}$ .

**13.3** Recall the problems **PACKING** and **SUBSETSUM** defined on page 257. Prove that there is a polyreduction from **SUBSETSUM** to **PACKING**.

**13.4** Write a Python program implementing the polytime conversion used in the previous exercise's reduction from **SUBSETSUM** to **PACKING**.

**13.5** The claim that  $\text{DHC} \leq_p \text{UHC}$  (claim 13.2, page 279) describes a polyreduction from **DHC** to **UHC**. Denote the conversion function in this reduction by  $C$ , and let  $I$  be the ASCII string given by

$$I = "u, v \ v, w \ w, u \ v, y \ y, v".$$

- (a) Draw the graph represented by  $I$ .
- (b) Draw the graph represented by  $C(I)$ .
- (c) Give  $C(I)$  as an ASCII string.

**13.6** Write a Python program implementing the polyreduction  $\text{DHC} \leq_p \text{UHC}$  described in claim 13.2, page 279. (Note that the book materials already provide an implementation of this reduction, `convertUHCToDHC.py`. Of course it would be preferable to write your own implementation without consulting the provided one.)

**13.7** Define the “stranded salesperson problem” **SSP**, to be exactly the same as **TSP** (figure 11.6, page 235), except that the salesperson does not have to start and end in the same city. In other words, **SSP** asks for a shortest Hamilton path rather

than Hamilton cycle. Let the decision version of SSP be SSPD. SSPD is defined in a similar way to TSPD on page 253: it takes an additional threshold  $L$  as input, and the solution is “yes” if there is a Hamilton path of length at most  $L$ . Prove that there is a polyreduction from SSPD to TSPD.

**13.8** Write a Python program implementing the polyreduction  $\text{SSPD} \leq_p \text{TSPD}$  in the previous exercise.

**13.9** Determine whether the following instance of CIRCUITSAT is positive or negative, and explain your reasoning:

$$\begin{aligned} w1 &= \text{NOT } x2 \\ w2 &= x1 \text{ AND } w1 \\ w3 &= w2 \text{ OR } x2 \text{ OR } x3 \\ \text{output} &= w1 \text{ AND } w3 \text{ AND } x3 \end{aligned}$$

**13.10** Apply the Tseytin transformation to the CIRCUITSAT instance of exercise 13.9. Is the resulting SAT instance positive or negative? Explain your reasoning.

**13.11** Determine whether the following instance of 3-SAT is positive or negative, and explain your reasoning:

$$\begin{aligned} &(x1 \text{ OR } x2 \text{ OR NOT } x3) \text{ AND } (\text{NOT } x2 \text{ OR NOT } x3) \\ &\quad \text{AND } (x1 \text{ OR } x3) \\ &\quad \text{AND } (\text{NOT } x1 \text{ OR NOT } x2) \\ &\quad \text{AND } (\text{NOT } x1 \text{ OR } x2) \end{aligned}$$

**13.12** Convert the 3-SAT instance of exercise 13.11 into a CIRCUITSAT instance, using the construction of claim 13.4 (page 287). Is the resulting instance positive or negative? Explain your reasoning.

**13.13** Convert the following SAT instance into a 3-SAT instance, using the construction of the claim that  $\text{SAT} \leq_p \text{3-SAT}$  (claim 13.7, page 289):

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4 \vee \neg x_5).$$

**13.14** Explain why 3-SAT cannot be reduced to 2-SAT via the clause splitting defined on page 288.

**13.15** Consider the decision problem REDDHC defined as follows: The input is a directed graph  $G$ , except that in addition to the usual information, every vertex in the graph has a *color*, which is either red or blue. The color information is appended to the usual graph description. For example, the input “a, b b, c c, a; a:red b:red c:blue” represents graph where vertices a and b are red and c is blue. We define a *red Hamilton cycle* to be a cycle that visits every red vertex exactly once, without visiting any blue vertices. The solution to

REDDHC is “yes” if a red Hamilton cycle exists, and “no” otherwise. Describe a polyreduction from REDDHC to DHC.

**13.16** Let us define a *mixed graph* to be a graph containing a mixture of directed and undirected edges. A Hamilton cycle on a mixed graph is a cycle that visits every node exactly once (except for the first node, which is visited again at the end of the cycle). The cycle may traverse undirected edges in either direction but must respect the orientation of directed edges. Cycles may traverse any edge at most once. Let MIXEDHC be a decision problem whose input is a mixed graph  $G$ . The solution is “yes” if  $G$  has a Hamilton cycle, and “no” otherwise.

Give an explicit polyreduction from MIXEDHC to the decision version of DHC, and prove that your construction is a polyreduction.

**13.17** Recall from exercise 12.14 (page 270) the decision problem SUBSET-SUMWITHFIVES: it is identical to the decision variant of SUBSETSUM, except that we may use up to 10 extra packages with weight 5 when constructing the desired subset. Prove that SUBSETSUM  $\equiv_p$  SUBSETSUMWITHFIVES.

**13.18** Consider the decision problem NEARLYSAT defined as follows: The input is a Boolean formula  $B$  in CNF. If  $B$  is satisfiable, the solution is “yes”. If all except one of  $B$ ’s clauses can be simultaneously satisfied, the solution is also “yes”. Otherwise the solution is “no”. For example, the instance “ $(x_1 \text{ OR NOT } x_2) \text{ AND } (x_2) \text{ AND } (\text{NOT } x_1)$ ” is positive, because the truth assignment “ $x_1=1, x_2=1$ ” satisfies two of the three clauses. Prove that NEARLYSAT  $\equiv_p$  SAT.

# 14



## NP-COMPLETENESS: MOST HARD PROBLEMS ARE EQUALLY HARD

One can find many classes of problems...which have algorithms of exponential order but seemingly none better.

—Jack Edmonds, *Paths, Trees, and Flowers* (1965)

### 14.1 P VERSUS NP

Recall that in this chapter and the previous chapter, we are focusing almost exclusively on *decision* problems. In contrast, the two complexity classes Poly and PolyCheck/NPoly were defined back in chapters 11 and 12 as classes of *general* computational problems, not just decision problems. When we restrict these classes to decision problems only, we get the world's two most famous complexity classes: P and NP. Restricting Expo to decision problems gives Exp, another well-known class. Here are the formal definitions:

#### Definitions of P, NP, and Exp

- P is the set of all decision problems in Poly.
- NP is the set of all decision problems in PolyCheck/NPoly.
- Exp is the set of all decision problems in Expo.

Why are P and NP the world's most famous complexity classes? Because they feature in the most famous unsolved mystery in computer science—a mystery known as “P versus NP.” A formal statement of this problem is

#### P VERSUS NP (version 1)

Are P and NP the same complexity class? That is, do they each consist of the same set of decision problems?

We have already seen that two apparently different complexity classes (`PolyCheck` and `NPoly`), defined in completely different ways, turned out to be identical. So even though the definitions of `P` and `NP` are different, it's not unreasonable to wonder whether they are equivalent.

It's obvious that `P` is a subset of `NP`. Why? For the same reason that `Poly` is a subset of `NPoly` (page 263), but let's repeat that reasoning here. Any decision problem `D` in `P` can be solved by some deterministic polytime Python program, say `D.py`. But a deterministic program is just a special case of a nondeterministic one—it's a “multithreaded” program that uses only one thread. So `D.py` can also be regarded as a nondeterministic polytime program that solves `D`, and hence `D` is in `NP`. So `P` is a subset of `NP`, and we can therefore reformulate the `P` versus `NP` question as

### P VERSUS NP (version 2)

Is there any decision problem `D` that lies outside `P` but inside `NP`?

Note that finding even one problem `D` that's in `NP` but not in `P` would resolve the whole `P` versus `NP` question. Computer scientists have catalogued thousands of problems that are (i) known to be in `NP`, and (ii) generally believed to be outside `P`. But not one of these problems has been *proved* to lie outside `P`.

Yet another way of looking at this is to think about converting nondeterministic programs into deterministic ones. If  $P=NP$ , then it must be possible to convert any polytime, nondeterministic program `ND.py` into a polytime, deterministic program `D.py` that computes the same answers. Recalling that programs are *equivalent* if they produce the same answers on the same inputs, we can again reformulate the `P` versus `NP` question:

### P VERSUS NP (version 3)

Is it always possible to convert a polytime, nondeterministic program into an equivalent polytime, deterministic program?

By definition, a polytime nondeterministic program launches a computation tree of threads like the ones in figure 8.6 (page 150). Each leaf of the tree corresponds to a potential solution. The *depth* of the tree is bounded by a polynomial—this follows directly from the definition of a polytime nondeterministic program (see page 258). But the *width* of a given layer of the tree could be exponential, because the number of threads can grow by a constant factor at each layer (see page 152). In particular, the number of leaves could be exponential as a function of the input size. The nondeterministic program can check all the leaves (i.e., the potential solutions) in parallel, and return any positive result. Therefore, the notion of converting a nondeterministic program into a deterministic one is

roughly equivalent to finding a deterministic method of evaluating all the leaves. It is, of course, possible to do this: we just examine the leaves one by one. But this could take an extraordinarily long time, since the number of leaves can be exponential. So the real question is, can we check the leaves in polynomial time? This results in our fourth reformulation of P versus NP:

**P VERSUS NP**  
(version 4, informal)

Is it always possible to search the leaves of a computation tree for an NP problem in polynomial time?

This version of the P versus NP question is labeled “informal” because it uses concepts that are not defined rigorously. But it captures the spirit of the question quite nicely. Problems in NP can always be solved by examining exponentially many leaves in a tree. Problems in P can always be solved by some other, faster technique. A program that actually visits every leaf must take exponential time, and therefore a polytime program *cannot* visit every leaf. Instead, it somehow searches the whole tree by cleverly ignoring the vast majority of possible solutions, and focusing on a tiny, polynomial-sized minority of the possible solutions. This is what we mean by “search the leaves of a computation tree” in version 4 of the P versus NP question.

## 14.2 NP-COMPLETENESS

Recall from section 13.7 that problems  $F$  and  $G$  are *polyequivalent* (written  $F \equiv_p G$ ) if  $F \leq_p G$  and  $G \leq_p F$ . Roughly speaking, polyequivalent problems are “equally hard.” More precisely, their methods of solution have the same running times, if we ignore polynomial factors. The polyreductions of chapter 13 showed that UHC  $\equiv_p$  DHC and that CIRCUITSAT  $\equiv_p$  SAT  $\equiv_p$  3-SAT. And in fact, although we haven’t proved it, there are polyreductions in both directions between SAT and UHC. So we have

$$\text{CIRCUITSAT} \equiv_p \text{SAT} \equiv_p \text{3-SAT} \equiv_p \text{UHC} \equiv_p \text{DHC}.$$

Roughly speaking, these five problems are equally hard. More precisely, either all five problems can be solved in polynomial time or all five problems require superpolynomial time.

Over the last few decades, computer scientists have identified thousands of other problems that are polyequivalent to these five problems. Each one of these thousands of problems is an *NP-complete* problem, and the set of all such decision problems is a complexity class that we’ll refer to as **NPComplete**. The NP-complete problems all rise together, or fall together: it’s been proved that either all can be solved in polynomial time, or none can. And of course, it’s widely believed that none can be solved in polynomial time.

But NP-complete problems have another extraordinary property: the NP-complete problems are the “hardest” problems in NP. That is, given any decision problem  $D$  in NP, and any problem  $C$  in NPComplete, there’s always a polyreduction from  $D$  to  $C$ . Actually, this property is usually used to define NP-completeness. Here is the classical, formal definition:

**Definition 1 of NP-complete (the “hardest” definition).** Let  $C$  be a problem in NP. We say that  $C$  is *NP-complete* if, for all  $D$  in NP,  $D \leq_P C$ .

So, a problem is NP-complete if it’s a “hardest” problem in NP, up to polynomial factors. But this classical definition of NP-completeness is difficult to apply in practice, because we have to find a polyreduction that works for any conceivable problem  $D$  in NP. Fortunately, back in the 1970s, two computer scientists (Stephen Cook and Leonid Levin) achieved this difficult goal: they each found a  $C$  to which any other problem  $D$  in NP can be polyreduced. And once you have a single  $C$  that fits the definition of NP-completeness, you can spread the NP-completeness to other problems by a much easier method.

For example, suppose that we have managed to prove, as Cook and Levin did in the early 1970s, that SAT is NP-complete according to definition 1 above. That is, Cook and Levin tell us that

$$\text{for all } D \in \text{NP}, \quad D \leq_P \text{SAT}. \tag{14.1}$$

But from claim 13.4 on page 287, we already know that

$$\text{SAT} \leq_P \text{CIRCUITSAT}. \tag{14.2}$$

Chaining together equations (14.1) and (14.2), we conclude that

$$\text{for all } D \in \text{NP}, \quad D \leq_P \text{CIRCUITSAT}.$$

In other words, CIRCUITSAT also satisfies the “hardest” definition 1 for NP-completeness.

In general, to show that a problem  $D$  is NP-complete, we just need to find some other problem  $C$  that is already known to be NP-complete, and give a polyreduction from  $C$  to  $D$ . This works because we can chain together two equations like (14.1) and (14.2), with  $C$  and  $D$  in place of SAT and CIRCUITSAT.

Hence, because the really hard work was done for us in the early 1970s, we can adopt the following equivalent—but much more convenient—definition of NP-completeness:

**Definition 2 of NP-complete (easier).** Let  $D$  be a problem in NP. We say that  $D$  is *NP-complete* if  $\text{SAT} \leq_P D$ .

And there’s nothing special about the use of SAT in this definition—SAT just happens to be one of the problems that Stephen Cook and Leonid Levin first proved to be NP-complete. This gives us an even more convenient definition:

**Definition 3 of NP-complete (even easier).** Let  $D$  be a problem in NP.

We say that  $D$  is *NP-complete* if  $C \leq_P D$ , for some problem  $C$  that is already known to be NP-complete.

Finally, it's obvious that any NP-complete problem (according to any of the above equivalent definitions) can be polyreduced to any other NP-complete problem—this follows from definition 1. In other words, the NP-complete problems are all polyequivalent to each other, and this leads to one more way of defining them:

**Definition 4 of NP-complete (via polyequivalence).** A problem is *NP-complete* if it is polyequivalent to SAT, or to any other problem that is already known to be NP-complete.

### Reformulations of P versus NP using NP-completeness

So, now we know that NP-complete problems like SAT are the “hardest” problems in NP. This fact has some remarkable consequences. For example, if someone invented a polynomial-time algorithm for SAT, it would immediately yield polynomial-time algorithms for every other problem in NP. (Why? Because all those other problems are “no harder than” SAT. More formally, the other problems  $D$  all polyreduce to SAT, so we can transform an instance of  $D$  to an instance of SAT in polynomial time, and then solve the SAT instance in polynomial time.) This gives us yet another way of asking whether P equals NP:

**P VERSUS NP**  
(version 5)

Is SAT in P?

And finally, there's nothing special about SAT in version 5 of P versus NP above. We know that all the NP-complete problems are polyequivalent, so P versus NP can be reformulated yet again as

**P VERSUS NP**  
(version 6)

Is any NP-complete problem in P?

## 14.3 NP-HARDNESS

Informally, a problem is NP-hard if it is at least as hard as the NP-complete problems. Formally, we define it as follows:

**Definition of NP-hard.** A computational problem  $F$  is *NP-hard* if there exists some NP-complete problem  $C$  with  $C \leq_P F$ .

Another way of thinking about this is that “NP-hard” and “NP-complete” mean the same thing, except that NP-complete problems must be in NP, whereas NP-hard problems may be outside NP. Definitions 1–3 of NP-completeness

(starting on page 297) all work for NP-hardness if we remove the requirement of membership in NP. Note in particular that NP-hard problems do not have to be decision problems, whereas NP-complete problems must be decision problems.

We also define the complexity class **NPHard** as the set of all NP-hard problems. Obviously, **NPComplete**  $\subseteq$  **NPHard**. But the converse does not hold: it's easy to come up with problems that are NP-hard but not NP-complete. This is because all NP-complete problems are, by definition, decision problems. Therefore, the general, nondecision variants of NP-complete problems are NP-hard but not NP-complete. As specific examples, the search variants of SAT and TSP are NP-hard but not NP-complete.

But these examples are a little unsatisfying: it's hard to imagine a program telling us that a Boolean formula is satisfiable without also finding a satisfying assignment. Similarly, it's hard to imagine a program telling us that a Hamilton cycle of length  $\leq L$  exists, without also finding such a cycle. And in fact, these intuitions are correct for NP-complete problems: if  $D$  is the decision variant of a search problem  $F$ , and  $D$  is NP-complete, then it can be proved that  $F$  has a polytime Turing reduction to  $D$ . This intriguing property is sometimes known as *self-reducibility*. Self-reducibility is beyond the scope of this book, but Goldreich's 2010 book *P, NP, and NP-Completeness* has good coverage of it.

The question remains: Are there any problems that are strictly harder than NP-complete problems? More specifically, are there any decision problems that are NP-hard but not NP-complete? The answer is yes, and we can fall back on our knowledge of computability to see why—uncomputable and undecidable problems provide examples that are strictly harder than NP-complete problems.

For a specific example, think back to the problem YESONSOME (figure 7.7, page 124), which takes a program `P.py` as input and outputs “yes” if and only if `P.py` outputs “yes” on at least one input. We will now give a proof:

**Claim 14.1.** YESONSOME is NP-hard.

*Proof of the claim.* We will polyreduce SAT to YESONSOME. Let  $I$  be an instance of SAT. We need to convert  $I$  into an equivalent instance of YESONSOME, and the conversion must run in polynomial time.

We may assume  $I$  contains  $k$  Boolean variables  $x_1, x_2, \dots, x_k$ . We can easily write a program  $P$  that evaluates the formula  $I$ . Specifically,  $P$  takes as input a string of  $k$  1s and 0s, indicating some proposed values for the  $x_i$ . The output of  $P$  is “yes” if the Boolean formula is true for the given values of the  $x_i$ , and “no” otherwise.

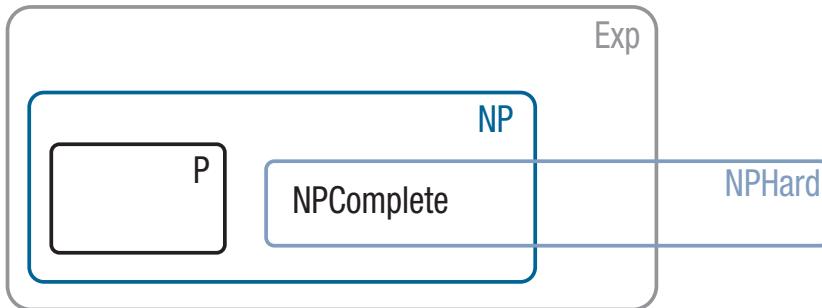
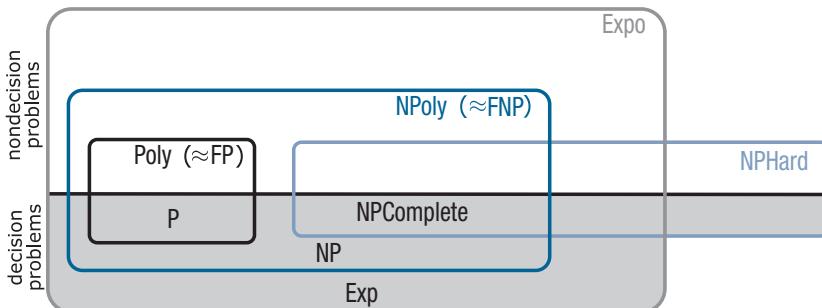
We claim  $P$  can be constructed in polynomial time. To see this, note that almost all of this program can be written ahead of time—figure 14.1 shows the first few lines, and you can consult the rest of the source code in the book resources if you're interested. The only catch is that our Boolean formula  $I$  must be included in the program. To do this, we must replace the string “dummy” at line 3 with the ASCII representation of  $I$ . But this string replacement operation requires only  $O(n)$  time, which is certainly polynomial.

Finally, we need to show that  $I$  is satisfiable if and only if  $P$  returns “yes” on some input. But this follows immediately from the construction of  $P$ , which returns “yes” on an input string of 1s and 0s if and only if the string represents a satisfying assignment for  $I$ . This completes the polyreduction.  $\square$

```

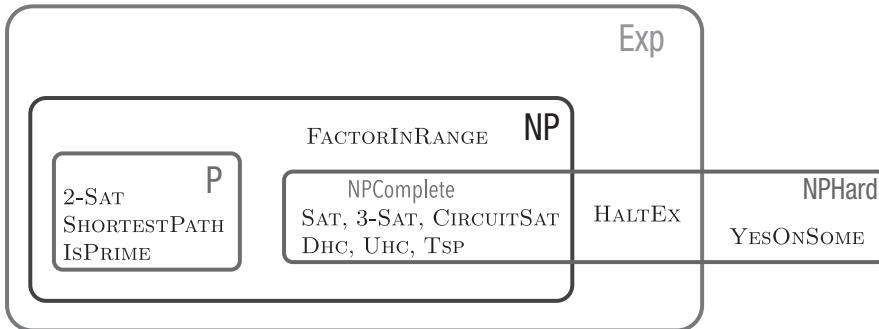
def evaluateSat(inString):
    # replace "dummy" with the desired formula
    booleanFormula = 'dummy'
    clauses = booleanFormula.split('AND')
    # return "yes" iff the clauses are all satisfied by the given inString
    # ...

```

**Figure 14.1:** The first few lines of `evaluateSat.py`.(a) relationship between classes of **decision problems**(b) relationships between classes of both **decision problems** (shaded), and **nondecision problems** (unshaded)

**Figure 14.2: Relationships between common complexity classes.** Both diagrams adopt the widely believed but unproven assumptions  $P \neq NP$  and  $NP \neq Exp$ . Panel (a) shows the relationships for decision problems only, whereas (b) shows the relationships between all types of problems. See the discussions on pages 229 and 266 for explanations of the alternative classes FP and FNP.

Figure 14.2 summarizes the relationships between the complexity classes we've encountered, under the standard but unproven assumption that  $P \neq NP$ . Figure 14.3 repeats figure 14.2(a), this time giving examples of decision problems that belong in each category. This figure again assumes  $P \neq NP$ , and makes the additional widely believed but unproven assumptions that the decision variant of FACTORINRANGE is outside P and HALTEX is outside NP.



**Figure 14.3:** Examples of decision problems lying inside or outside the common complexity classes, assuming  $P \neq NP$  and  $HALTEX \notin NP$  and  $FACTORINRANGE \notin P$ .

## 14.4 CONSEQUENCES OF P=NP

Why are computer scientists obsessed with the P versus NP question? One reason is the potentially miraculous consequences of a positive answer: in the unlikely event that P *does* equal NP, the impact of computers on our society could be dramatically increased. There's a delightful description of this scenario in Lance Fortnow's 2013 book, *The Golden Ticket: P, NP, and the Search for the Impossible*. Fortnow describes a hypothetical "beautiful world" in which someone has discovered a computer program that solves NP-complete problems in polynomial time. Let's denote this beautiful program by  $B$ . Because  $B$  can solve any NP problem in polynomial time, one of the things it can do in polynomial time is find more efficient versions of itself. (We'll omit the details of exactly why this is true.) So Fortnow explains that even if the original version of  $B$  is somewhat inefficient or impractical, we might be able to apply  $B$  to itself over and over again, until we eventually arrive at a new, highly efficient, and practical program  $B'$  that solves large instances of many NP-complete problems.

The beautiful program  $B'$  can probably do many strange and wonderful things, including

- discover new drugs by solving NP-complete protein-folding problems, perhaps curing diseases such as cancer and AIDS;
- develop better models for applications ranging from weather forecasting to logistical scheduling, improving the accuracy of many scientific endeavors and reducing costs for businesses;
- surpass human-level ability in many artificial intelligence tasks, such as face recognition, handwriting recognition, and speech recognition.

This would indeed be a beautiful world. But of course we need to keep it in perspective. Most likely, it will turn out that  $P \neq NP$ , so the beautiful programs  $B$  and  $B'$  don't exist. And even if  $P=NP$ —so a polytime  $B$  does exist—the consequences are unclear. Perhaps  $B$ , even after unlimited optimization, will have a running time of at least  $n^{10}$ . If so, every time we increased the length of the input by a factor of 10, the running time would increase by a factor of at least 10 billion.

So although  $B$  would run in polynomial time, it's possible that it wouldn't help much in the search for a cure for cancer.

## 14.5 CIRCUITSAT IS A "HARDEST" NP PROBLEM

As remarked earlier, the easiest way to prove a problem NP-complete is to use a polyreduction from another NP-complete problem. But to get this process started, we have to prove that at least one problem is NP-complete according to the original “hardest NP problem” definition—definition 1 on page 297. Stephen Cook and Leonid Levin did this for SAT in their famous results from the early 1970s. But here we will take a different route, by sketching a proof of the fact that CIRCUITSAT is a hardest NP problem. First we need a general result about converting Turing machines into circuits. In plain English, this result states that a polytime Turing machine can be implemented in polynomial time by a polynomial-sized circuit. Here are the details:

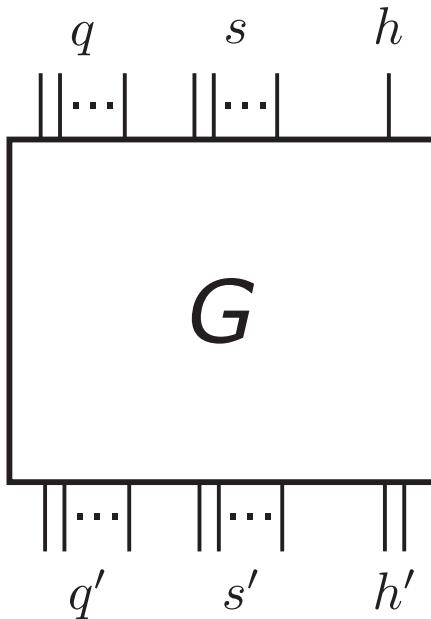
**Claim 14.2.** Let  $M$  be a single-tape Turing machine. Suppose the running time of  $M$  is bounded by a polynomial  $p(n)$  for inputs of length  $n$ . Then for each  $n > 0$ , there exists a circuit of size  $O(p(n)^2)$  computing the same function as  $M$  for all inputs of length  $\leq n$ . Moreover, the circuit can be constructed in  $O(p(n)^2)$  time.

*Sketch proof of the claim.* Any computation by  $M$ , on an input of length  $\leq n$ , can be described in complete detail by a grid as in figure 5.4, page 76. The number of rows in the grid is the number of steps in the computation, which is bounded by  $p(n)$ . What about the number of columns in the grid? Even if the head moves to the right on every step, it cannot move more than  $p(n)$  tape cells. So the grid needs at most  $p(n)$  columns. To summarize, we have at most  $p(n)$  rows and columns, meaning there are at most  $p(n)^2$  entries in the grid.

To complete the proof, we need to show that each entry in the grid can be simulated by a circuit  $G$  (for grid element) of fixed size. We omit the detailed construction of  $G$  here. Instead we give a high-level argument that  $G$  has a fixed size. Note that the tape symbol at each grid entry can be computed from the symbol  $s$  in the cell immediately above, provided we know the state  $q$  of the machine  $M$  and a bit  $b$  telling us whether the read-write head is scanning the current cell. As shown in figure 14.4, this information can be fed in to  $G$  on input wires; the new state  $q'$ , direction of head movement  $b'$ , and new tape symbol  $s'$  can be read from  $G$ 's output wires. Note that each piece of data, such as  $q'$  and  $s'$ , must be encoded in binary, which is why the figure shows clusters of individual wires for each piece of data. For each of the individual output wires, the transition function of  $M$  determines a Boolean function of the inputs  $q, b, s$ .

We now appeal to the well-known fact that any Boolean function can be implemented by a circuit of AND, OR, and NOT gates. So, our grid element circuit  $G$  can be implemented by a circuit of fixed size that computes the Boolean functions for each output wire. Curious readers can see a specific example in figure 14.5, but it's not necessary to understand the details.

The full circuit implementing  $M$  consists of a  $p(n) \times p(n)$  grid containing copies of  $G$ , all linked by suitable wires and gates to propagate the computation from one row to the next. We again omit precise details of the construction, but



**Figure 14.4: The grid element circuit  $G$ .** Inputs are the current state  $q$ , tape symbol  $s$ , and a bit  $h$  specifying whether or not the head is scanning the current cell. Outputs are the new state  $q'$ , new symbol  $s'$ , and head movement  $h'$ . Values of  $q, q', s, s'$  are encoded in binary on multiple wires;  $h$  is a single bit, and  $h'$  uses two bits to represent either  $L$  for left,  $R$  for right, or  $S$  for stay.

figure 14.6 demonstrates the main ideas. In particular, only a fixed number of linking wires and gates are required for each copy of  $G$ .

So we obtain a grid of  $p(n)^2$  copies of  $G$ , yielding a circuit of size  $O(p(n)^2)$ . It’s also clear that the grid can be constructed in  $O(p(n)^2)$  time. This completes our sketch of the proof.  $\square$

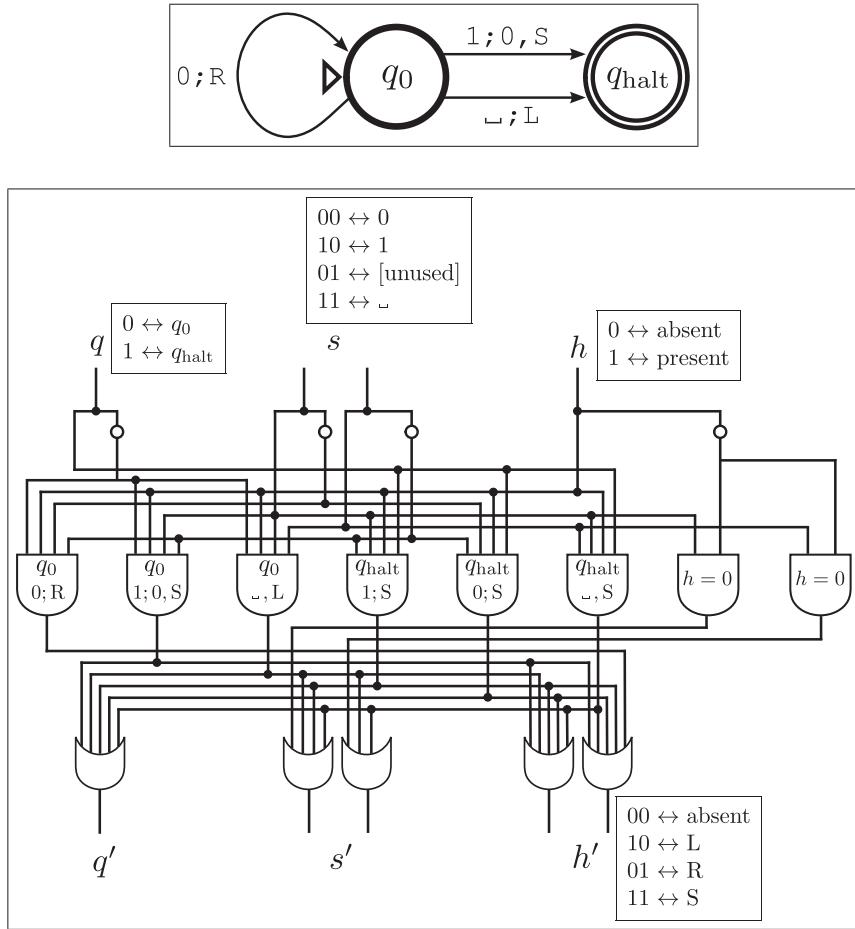
Now we are ready to prove that CIRCUITSAT is NP-complete according to the original “hardest” definition of NP-completeness:

**Claim 14.3.** Let  $D$  be a problem in NP. Then there is a polyreduction from  $D$  to CIRCUITSAT.

*Proof of the claim.* Since  $D$  is in NP, we can verify its solutions in polynomial time. Specifically, there is a polytime program `verifyD.py` whose parameters are an input string  $I$ , a proposed solution  $S$ , and a hint string  $H$ .

In this book, we usually work with ASCII inputs and outputs, but in this proof it will be easier to work with binary. So  $I$  and  $H$  are binary strings, and the output is a single bit: “1” for “correct” (i.e.,  $I$  has been verified as a positive instance of  $D$ ), and “0” for “unsure” (i.e., we failed to verify that  $I$  is a positive instance of  $D$ ).

Because  $D$  is a decision problem, there are only two possible values for the proposed solution  $S$ . In binary, we use “1” for a positive solution and “0” for a negative solution. In particular, the length of  $S$  is exactly 1.

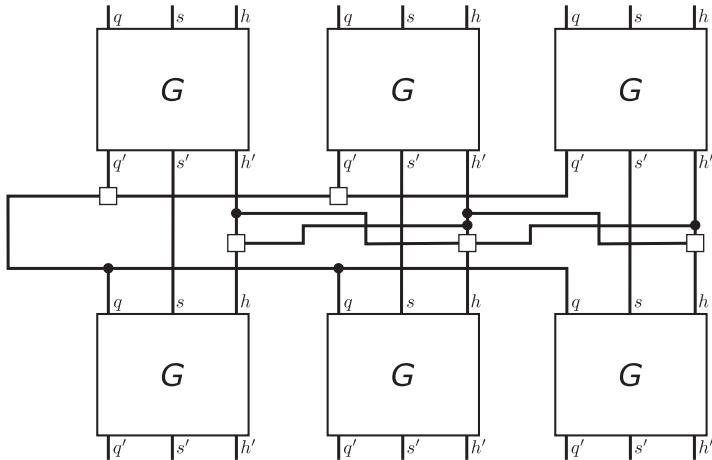


**Figure 14.5: Detailed example of a grid element circuit.** Top: A simple Turing machine  $M$  that converts the first 1 to a 0. Bottom: A grid element circuit  $G$  for the machine  $M$ .

We can assume that the length of  $H$  is bounded by a polynomial—see claim 12.1 on page 255 for details. By definition of a polytime verifier, the running time of `verifyD.py` is also bounded by a polynomial. Combining these two statements, we have some polynomial  $p(n)$  that bounds both the length of  $H$  and the running time of `verifyD.py`. Note that  $p(n)$  is a function of  $n$ , the length of  $I$ . Importantly,  $n$  does *not* include the length of  $S$  or  $H$ .

We are now ready to begin the main proof. We are trying to polyreduce  $D$  to CIRCUITSAT. As a first step, we will convert `verifyD.py` into a circuit  $C$ . Programs run on computers, and computers are made up of circuits, so this seems quite reasonable; additional details will be given later. For now, note that the circuit will have  $n$  inputs for  $I$ , a single bit for  $S$ ,  $p(n)$  inputs for  $H$ , and a single binary output. Let's label the inputs  $i_1, i_2, \dots, i_n$  for the  $I$ -values,  $s$  for the  $S$ -value, and  $h_1, h_2, \dots, h_{p(n)}$  for the  $H$ -values.

The resulting circuit  $C$  is shown in the top panel of figure 14.7. So now we have a circuit  $C$  that mimics `verifyD.py`: by feeding the bits from  $I$ ,  $S$ , and  $H$



**Figure 14.6: The grid elements can be linked by wires to simulate the Turing machine computation.** This simplified example shows a  $2 \times 3$  grid, but the full grid is  $p(n) \times p(n)$ . The “wires” shown here represent clusters of individual wires encoding values in binary, as explained in figure 14.4. Small white squares indicate simple circuits that combine these encoded values using bitwise OR gates.

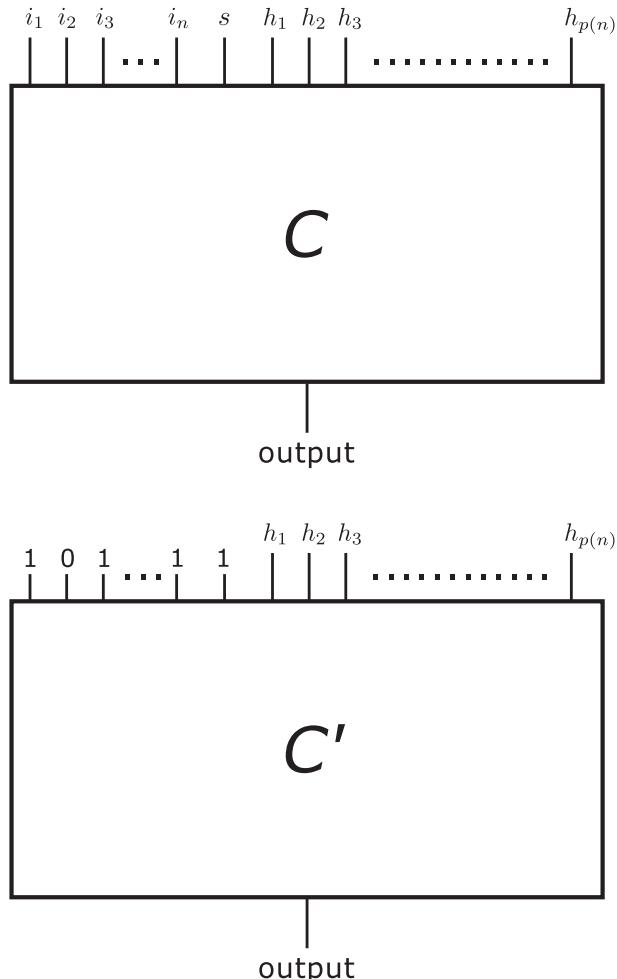
into the top of the circuit, we obtain an output bit at the bottom equal to the output of `verifyD(I, S, H)`.

Again, recall our main objective: polyreduce  $D$  to CIRCUITSAT. Given an instance of  $D$  (i.e., an input  $I$ ), we need to convert  $I$  to an equivalent instance of CIRCUITSAT, and the conversion must run in polynomial time. Here’s how we do it. Given  $I$ , first construct the circuit  $C$  described above. Then modify  $C$  to create a new circuit  $C'$ , by fixing the values of  $i_1, i_2, \dots$  to the specific values of the binary string  $I$ . We also set  $S = 1$ , since we’re interested only in verifying a positive solution. The new circuit has  $p(n)$  inputs labeled  $b_1, b_2, \dots$ , as shown in the lower panel of figure 14.7. Now we ask the following question about  $C'$ : Are there any possible values of the inputs  $b_1, b_2, \dots$  that produce an output of 1? This question is an instance of CIRCUITSAT! And the answer is yes, if and only if there is some hint  $H$  that causes `verifyD.py` to output a 1. This result maps perfectly onto our original problem  $D$ :

Input  $I$  is a positive instance of  $D$   
*if and only if*  
 there exists some hint  $H$  for which `verifyD(I, S, H)` outputs 1  
*if and only if*  
 $C'$  is a positive instance of CIRCUITSAT.

Thus, we certainly have a mapping reduction from  $D$  to CIRCUITSAT.

But the proof is incomplete: we haven’t yet shown that the conversion can be performed in polynomial time. To prove this, it’s easiest to think of `verifyD.py` as a single-tape Turing machine  $V$ . By definition of a polytime verifier,  $V$  always terminates in polynomial time. So claim 14.2 (page 302) can be applied, yielding a circuit of polynomial size in polynomial time.  $\square$

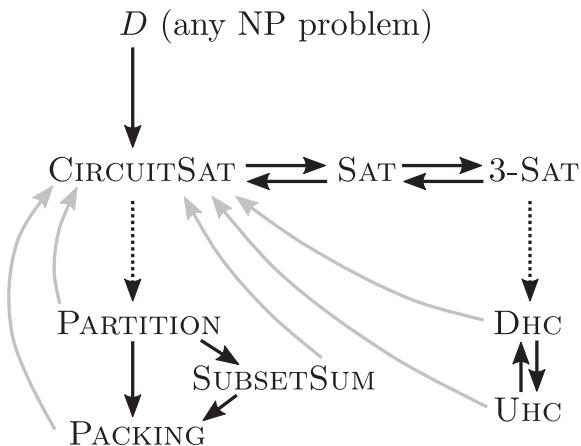


**Figure 14.7:** Top: The circuit  $C$  implementing  $\text{verifyD}(I, S, H)$ . Bottom: The circuit  $C'$  implementing  $\text{verifyD}$  for one particular fixed value of  $I$  (in this case,  $101\dots1$ ) and  $S = 1$ .

So, this claim tells us that for any NP problem  $D$ , we have  $D \leq_p \text{CIRCUITSAT}$ . In other words, when we ignore polynomial factors,  $\text{CIRCUITSAT}$  is at least as hard as any other NP problem. And because all the NP-complete problems are polyequivalent to  $\text{CIRCUITSAT}$  (see definition 4 of NP-completeness, on page 298), this means that *any* NP-complete problem is a “hardest” NP problem.

## 14.6 NP-COMPLETENESS IS WIDESPREAD

NP-completeness is surprisingly widespread. Richard Karp discovered 21 NP-complete problems in 1972 (see chapter 17 for more details) and this set off a burst of publications demonstrating the NP-completeness of many more



**Figure 14.8: The web of NP-completeness.** Arrows represent polyreductions. Solid dark arrows are explicitly proved in this book; dotted arrows can be found in other textbooks; solid light arrows follow from the NP-completeness of CIRCUITSAT.

problems, in essentially all areas of computer science. Only seven years later, in 1979, Michael Garey and David Johnson published a 340-page book called *Computers and Intractability: A Guide to the Theory of NP-Completeness*. This book contained a list of over 300 NP-hard problems, and of course many more are known today.

It would take us too far afield to study more NP-complete problems in any detail. Instead, let's whet our appetites by mentioning just the names of a few more NP-complete problems. We will see all 21 of Karp's original problems in chapter 17, including classic problems such as MAXCLIQUE, MAXCUT, INTEGERPROGRAMMING, and TASKASSIGNMENT. Some of the more interesting-sounding problems in the book by Garey and Johnson include CUBICSUBGRAPH, MINIMUMBROADCASTTIME, SPARSEMATRIXCOMPRESSION, and DEADLOCKAVOIDANCE. Pure mathematics has its own share of NP-complete problems, including QUADRATICDIOPHANTINEEQUATION and COSINEINTEGRAL (for definitions, see Moore and Mertens). And the MULTIPLESEQUENCEALIGNMENT problem from page 5 is a more modern NP-complete problem with applications to genetics.

Figure 14.8 presents another way of looking at the vast range of NP-complete problems. Since these problems are all polyequivalent to each other, we might poetically call this the “web of NP-completeness.” Note that  $D$  represents any NP problem (not necessarily NP-complete), and all other problems in the diagram are in fact NP-complete. Each arrow in the diagram represents a polyreduction. Solid, dark arrows represent polyreductions that have already been described in this book (or, in the case of the two involving SUBSETSUM, are easy exercises). Dotted arrows show polyreductions that are omitted from this book. These omitted polyreductions can be found in many other texts—my favorite would be *The Nature of Computation*, by Moore and Mertens—and in Karp's 1972 paper, which is described in chapter 17. But the key thing to understand in this diagram is the lightly shaded arrows. None of these polyreductions is proved explicitly

in this book, but they all follow immediately from the NP-completeness of CIRCUITSAT, and we *did* prove this in claim 14.3, page 303. In other words, the dark arrow from  $D$  to CIRCUITSAT means that we can draw an arrow from any other NP problem back to CIRCUITSAT. In fact, we could go much further: since every NP-complete problem is polyequivalent to every other NP-complete problem, we could draw arrows between every pair of problems in the diagram (except  $D$ , which isn't NP-complete).

## 14.7 PROOF TECHNIQUES FOR NP-COMPLETENESS

To prove that some problem  $D$  is NP-complete, it's easiest to use definition 3 on page 297. Specifically, we need first to show that  $D \in \text{NP}$ . Then, we choose any problem  $C$  that is already known to be NP-complete, and show that  $C \leq_p D$ . Which  $C$  should we choose for the proof? It is often easiest to choose a  $C$  that is similar to  $D$ , since that is likely to lead to a simple polyreduction.

Consider the following example, which uses the problem HALFUHC. This problem is identical to UHC (page 277), except that we ask for a cycle that visits at least *half* of the nodes, instead of all nodes.

**Claim 14.4.** HALFUHC is NP-complete.

*Proof of the claim.* HALFUHC is clearly in  $\text{NP}$ , since we can verify positive instances in polynomial time, using a hint consisting of a cycle containing half of the nodes in the input graph.

To prove HALFUHC is NP-complete, we will show that  $\text{UHC} \leq_p \text{HALFUHC}$ . This will prove the claim, since UHC is already known to be NP-complete.

So, we need to demonstrate a polyreduction from UHC to HALFUHC. Let  $G$  be a graph representing an arbitrary instance of UHC. We transform  $G$  into  $G'$ , another graph that will be an equivalent instance of HALFUHC, as follows. Let  $G'$  consist of two copies of  $G$ , with no edges joining the two copies. (The nodes in  $G'$  will need to be renamed in some simple way, but we omit this detail.) Clearly,  $G'$  contains a half-Hamilton cycle if and only if  $G$  contains a full Hamilton cycle. So this transformation maps positive instances to positive instances and negative instances to negative instances, as required. In addition, the transformation can be achieved in linear time, which is certainly polynomial. Hence, this is a polyreduction and the claim is proved.  $\square$

An implementation of this transformation is available as `convertUhcToHalfUhc.py`.

### Proving NP-hardness

To prove that a problem  $F$  is NP-hard, use exactly the same techniques as for NP-completeness, except that you don't need to show that  $F$  is in  $\text{NP}$ . For example, consider the search variant of HALFUHC, whose positive solution sets contain actual half-Hamilton cycles instead of "yes". This isn't a decision problem, so it can't be NP-complete. But we can prove it is NP-hard using essentially the same proof as above, except with the first sentence omitted.

## 14.8 THE GOOD NEWS AND BAD NEWS ABOUT NP-COMPLETENESS

Sometimes the rhetoric about NP-completeness starts to sound a little overwhelming. Is it really true that *all* the hard, important problems are NP-complete? Is it really true that it's pointless trying to solve an NP-complete problem? The answers here are "not necessarily," and the next few subsections point out some of the resulting subtleties of NP-completeness.

### Problems in **NPoly** but probably not **NP-hard**

Some problems in **NPoly** have important applications, have been studied intensively, appear to require superpolynomial time, but their decision variants have never been proved **NP-hard**. Here are two well-known problems in this "intermediate" region:

- **FACTOR:** Compute a nontrivial factor of an integer.
- **DISCRETELOG:** Compute  $\log_b N$  using clock arithmetic.

It's worth noting that problems related to factoring and discrete logarithm lie at the heart of some widely used cryptographic techniques. As already discussed on page 216, an efficient algorithm solving **FACTOR** would crack the popular RSA cryptosystem. Similarly, it turns out that an efficient algorithm solving **DISCRETELOG** would break the Diffie–Hellman key exchange technique, which is commonly used for establishing encrypted network connections. Sadly, it's beyond the scope of this book to study any more details of the connections between complexity theory and cryptography. But hopefully, you are already struck by the remarkable fact that real-world cryptographic techniques depend on computational problems that seem to lie in the sparsely populated intermediate region between **Poly** and **NPComplete**.

**GRAPHISOMORPHISM** is another problem that may occupy this intermediate region and also has important applications. It is defined as follows.

- **GRAPHISOMORPHISM:** Given two graphs, determine whether they are *isomorphic* (i.e., are they actually the same graph, but with the nodes permuted?).

This problem has an interesting history. It is clearly in **NP**, and naïve methods of solution require exponential time. But decades of research have chipped away at the problem, producing surprisingly effective algorithms. In 2015, the University of Chicago mathematician László Babai announced a quasipolynomial-time algorithm for **GRAPHISOMORPHISM**. At the time of writing (late 2010s), there is a certain amount of optimism that a true polynomial-time algorithm may yet be discovered.

### Some problems that are in **P**

While we are in the business of deflating some of the rhetoric surrounding NP-completeness, let's remind ourselves that there are some quite remarkable

algorithms solving important, difficult problems in polynomial time. A short list would include the following:

- ISPRIME: Given an integer, determine whether it is prime (see page 237).
- MINCUT: Given a weighted graph, split the nodes into two subsets so that the total weight of the edges between the subsets is minimized.
- SHORTESTPATH: Find the shortest path between nodes in a weighted graph (see figure 11.6, page 235).
- LINEARPROGRAMMING: Find an optimal solution to a set of linear equations and inequalities. (This problem crops up in Richard Karp's famous paper, discussed in chapter 17—see page 367.)

And of course there are thousands of other important, useful problems in P. Consult any algorithms textbook for a selection of these.

### Some NP-hard problems can be approximated efficiently

Here's some more good news: despite the fact that we can't solve them exactly, some NP-hard problems can be *approximated* efficiently. For example, there is a polynomial-time algorithm guaranteed to find a solution to TSP at most 50% longer than the optimal solution, provided the input graph satisfies a reasonable condition known as the *triangle inequality*. In fact, if the graph satisfies the slightly stronger requirement of being *Euclidean*, TSP can be approximated arbitrarily well! That is, given any desired nonzero accuracy—say, 1%—there's a polytime algorithm that finds a solution of the desired accuracy. The TASKASSIGNMENT problem mentioned above is another example of an NP-hard problem that can be approximated arbitrarily well. There are, however, trade-offs involved in these approximations. For example, as the maximum error of the approximation approaches zero, the constant hidden in the big-O notation increases dramatically. So we still can't obtain a truly practical algorithm with arbitrarily good accuracy.

### Some NP-hard problems can be solved efficiently for real-world inputs

Our last piece of good news to combat the gloom of NP-completeness is that there are some NP-hard problems that have efficient algorithms *in practice*. This means that the worst-case running time of the algorithm is exponential, but the algorithm happens to run efficiently on large instances of the problem that are actually encountered in the real world. The most famous example of this is SAT. Because of the numerous practical applications mentioned on page 281, solving SAT problems is a huge industry in its own right. There are annual contests held for various types of real-world SAT problems, and researchers are constantly coming up with improved SAT-solvers that perform well in these contests on problem instances involving millions of clauses.

### Some NP-hard problems can be solved in pseudo-polynomial time

As we now know, complexity theorists usually insist on measuring the complexity of a program as a function of the length of the input,  $n$ . For problems whose

inputs are integers, however, it does sometimes make sense to instead measure the complexity in terms of the magnitude of the largest input integer,  $M$ . If a problem of this kind can be solved in polynomial time as a function of  $M$ , we say there is a *pseudo-polynomial-time* algorithm.

Now for some more good news: several famous NP-hard problems can be solved in pseudo-polynomial time. These include PACKING, SUBSETSUM, PARTITION, and TASKASSIGNMENT.

## EXERCISES

**14.1** Recall that the decision problem 5-SAT is defined in the same way as 3-SAT, but with clauses of up to 5 literals instead of 3. Is 5-SAT NP-hard? Give a rigorous proof of your answer.

**14.2** Let EXACTLY3-SAT be a decision problem defined in the same way as 3-SAT, except that each clause must have exactly 3 literals instead of at most 3. (Literals in a clause need not be distinct.) Prove that EXACTLY3-SAT is NP-complete.

**14.3** Let DISTINCT3-SAT be the same as EXACTLY3-SAT except that each clause must contain 3 distinct variables. Is DISTINCT3-SAT NP-complete? Give a rigorous proof of your answer.

**14.4** Define the decision problem SAT-5CLAUSES as follows. The input is a Boolean formula  $B$  in CNF. The solution is “yes” if it is possible to simultaneously satisfy at least 5 of  $B$ ’s clauses. Is SAT-5CLAUSES NP-complete? Give a rigorous proof of your answer.

**14.5** RSA is a popular public key encryption system discussed on page 216. For its security, RSA relies on the fact that there is no known efficient (or polytime) way of computing a private RSA key from the corresponding public key. Let CRACKRSA be the function problem that takes a public RSA key as input and outputs the corresponding private RSA key as a solution. It’s not hard to show that a suitable decision variant of CRACKRSA polyreduces to FACTOR. Based on these facts, discuss whether CRACKRSA is likely to be a member of Poly, NPoly, or NPHard. Explain your reasoning. (Detailed reductions are not required. Knowledge of how RSA works is not required. Just use the facts stated in the question.)

**14.6** For the purposes of this exercise, we consider only the following complexity classes: P, NP, Exp, Poly, NPoly, Expo, NPComplete, NPHard. For each of the computational problems given below, (i) list each complexity class that the problem is known to be a member of, and (ii) list each complexity class that the problem is widely believed to be outside of:

- (a) FACTOR, a search problem defined on page 215
- (b) The decision variant of FACTORINRANGE, defined on page 237
- (c) DHC, considered as a search problem whose positive solutions are directed Hamilton cycles
- (d) TSPD—a decision problem defined on page 253

- (e) ISPRIME, considered as a decision problem
- (f) 3-SAT, considered as a decision problem
- (g) 5-SAT, considered as a decision problem
- (h) 2-SAT, considered as a decision problem

**14.7** Using some brief research, find and describe an NP-complete problem that is not closely related to any computational problem discussed in this book. Informal sources are fine, but you should cite your source(s).

**14.8** Prove that any decidable problem has a polyreduction to YESONEMPTY.

**14.9** Define QUARTERUHC in the same way as HALFUHC (claim 14.4, page 308), except that at least a quarter of the nodes must be in a cycle.

- (a) Prove that QUARTERUHC is NP-complete.
- (b) Implement a polyreduction from UHC to QUARTERUHC in Python.  
(Hint: Use convertUhcToHalfUhc.py as a starting point.)

**14.10** Consider the problem MANYSAT, defined as follows. The input is a Boolean formula  $B$  in CNF. If it is possible to satisfy  $B$  in at least two different ways, the instance is positive and the solution consists of a string containing exactly two different satisfying truth assignments separated by a semicolon. Otherwise there must be exactly one or zero satisfying truth assignments, and the solution is “no”.

For example, the input

“(x1 OR NOT x2) AND (NOT x1 OR x2) AND (x1 OR NOT x1)”

is a positive instance, because there are two satisfying truth assignments ( $x_1 = 1, x_2 = 1$  and  $x_1 = 0, x_2 = 0$ ), so a solution is “ $x_1=1, x_2=1; x_1=0, x_2=0$ ”. On the other hand, the input

“(x1 OR NOT x2) AND (NOT x1 OR x2) AND (NOT x1)”

is a negative instance (there is only one satisfying truth assignment), so the solution is “no”.

Prove that MANYSAT is NP-hard.

**14.11** Consider the decision problem LEXHAMCYCLE (LHC). The input is an undirected graph in which each node is (as usual) labeled with a distinct ASCII string. An instance is positive if there exists a Hamilton cycle that visits the nodes in lexicographical order. For example, the instance “a, b a, d b, c c, d” is positive but the instance “a, c a, d b, c b, d” is negative.

Which common complexity classes is LHC a member of? Specifically, state whether or not LHC is a member of P, Poly, NP, NPoly, Exp, NPComplete, and NPHard. Rigorous proof is not required, but justify your answer with informal reasoning.

**14.12** Let MIXEDHC be as in exercise 13.16 (page 293), and let  $C$  be an NP-complete problem. Prove that there is a polyreduction from MIXEDHC to  $C$ .

**14.13** In another galaxy, there is a planet populated by strange creatures called *zorks*. The zorks have developed powerful technology for growing an

immense variety of vegetables. The vegetables are cooked and served by a central government agency, and each type of vegetable is served with either pepper or salt, but not both. Suppose there is a total of  $M$  zorks denoted  $z_1, z_2, \dots, z_M$  and  $N$  vegetables denoted  $v_1, v_2, \dots, v_N$ . Every year, the zork government carries out a survey of the entire population to find out which vegetables should be served with pepper, and which with salt. Each zork is permitted to submit a vote specifying pepper or salt for up to three vegetables of their choice. For example, zork  $z_{65}$  might submit as a vote

$$v_{4125} : \text{pepper}, \quad v_{326} : \text{salt}, \quad v_{99182} : \text{pepper}. \quad (\star)$$

A zork is *happy* if the government follows at least one of its choices; if the government implements none of a zork's choices, the zork will be *sad*. For example, given the vote  $(\star)$  above,  $z_{65}$  would be happy if the government served  $v_{326}$  with salt, regardless of the government's decision on  $v_{4125}$  and  $v_{99182}$ .

Consider the computational problem HAPPYZORKS, defined as follows: The input is a list of  $M$  votes in the form of equation  $(\star)$  above. A solution is a list specifying a particular choice of salt or pepper for each type of vegetable, such that every zork will be happy.

- (a) Prove that HAPPYZORKS is NP-hard.
- (b) In practice, do you think it will be possible for the government to make all the zorks happy if there are a few million zorks and a few million vegetables? Write a few sentences explaining your answer.

**14.14** Consider the computational problem HALFSAT defined as follows. The input is a Boolean formula  $B$  in CNF. If it is impossible to satisfy at least half of  $B$ 's clauses, the solution is “no”. Otherwise a solution is a truth assignment that does satisfy at least half of  $B$ 's clauses. For example, the input “ $(x_1 \text{ OR NOT } x_2) \text{ AND } (x_2) \text{ AND } (\text{NOT } x_1)$ ” is a positive instance, because the truth assignment “ $x_1=1, x_2=1$ ” satisfies two of the three clauses and is therefore a positive solution. Prove that HALFSAT is NP-hard.

**14.15** Consider the decision problem 2PACKING, which is the same as PACKING (figure 12.6, page 257) except that the packages must be packed into two identical trucks rather than a single truck. Prove that PACKING and 2PACKING are polyequivalent. (Hint: A direct polyreduction in one direction is easy. For the other direction, use the ideas of this chapter.)

**14.16** Consider a computational problem called MYSTERYSEARCH, or MS for short. The only three facts we are given about this problem are

- MS is a search problem, *not* a decision problem;
- $\text{PACKINGD} \leq_p \text{MS}$ , where  $\text{PACKINGD}$  is the decision variant of  $\text{PACKING}$ ;
- $\text{MS} \in \text{NPoly}$ .

Answer the following questions, giving a brief justification or explanation in each case:

- (a) Is  $\text{MS} \in \text{NP}$ ?
- (b) Does there exist a polytime verifier for MS?

- (c) Is MS NP-complete?
- (d) Is MS NP-hard?
- (e) Is  $\text{MS} \in \text{Poly}$ ?

**14.17** Let KCYCLE be a decision problem defined as follows: The input is a graph  $G$  and integer  $K$ . The solution is “yes” if  $G$  contains a cycle of exactly  $K$  nodes, and “no” otherwise. Prove that KCYCLE is NP-complete.

**14.18** Consider the decision problem DECIMATE, defined as follows: The input is the same as for PARTITION, that is, a list of numeric weights separated by whitespace. The solution is “yes” if the weights can be partitioned into 10 sets of equal weight, and “no” otherwise. Discuss whether or not DECIMATE is, or is likely to be, a member of P, NP, NPComplete, and NPHard. Give rigorous mathematical proofs where possible.

Part III  
ORIGINS AND APPLICATIONS





# 15



## THE ORIGINAL TURING MACHINE

We must not therefore thinke that Computation, that is, Ratiocination, has place onely in numbers.

— Thomas Hobbes, *Elements of Philosophy* (1656)

In 1936, before electronic computers had even been invented, Alan Turing wrote a paper that laid the foundation for a new field: the theory of computation. The 36-page paper was entitled “On computable numbers, with an application to the Entscheidungsproblem.” It was published in the *Proceedings of the London Mathematical Society*, initially in two parts (November and December 1936), then again as a complete unit in the following year, 1937. It is hard to overstate the groundbreaking insight and genius of Turing’s paper. Among other things, it contains

- (a) a definition of “computing machines” (which we now call “Turing machines”);
- (b) a convincing argument that any computation performed by a human can also be performed by a computer;
- (c) a complete example of a universal computer;
- (d) proofs that various problems are undecidable, including a variant of the halting problem;
- (e) a resolution of a famous mathematical problem (the so-called *Entscheidungsproblem*, described in more detail on page 347);
- (f) many other computational ideas, including nondeterminism, subroutines, recursion, binary numbers, Turing reductions, and proofs that an immense variety of mathematical calculations can be mechanized.

In this chapter, we will look at excerpts from “On computable numbers” in an attempt to understand some of Turing’s thinking. We will cover only a small fraction of the paper, focusing on items (a) and (b) in the list above. Entire books have been devoted to Turing’s paper, and the presentation here is directly inspired by Charles Petzold’s wonderful book *The Annotated Turing: A Guided Tour through Alan Turing’s Historic Paper on Computability and the Turing Machine*.

The excerpts presented here have been freshly typeset with a shaded background, but are otherwise identical to Turing's original publication. (The only exception is that the symbol  $\mathcal{S}$  is used instead of the weird Gothic font in the original.) My own comments and explanations are inserted between square brackets [*like this*].

## 15.1 TURING'S DEFINITION OF A "COMPUTING MACHINE"

Let's jump right in to our first excerpt. This is the introduction, which previews some of the main results in the paper:

### ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

*By A. M. TURING.*

[RECEIVED 28 MAY 1936.—READ 12 NOVEMBER, 1936.]

The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means.... According to my definition, a number is computable if its decimal can be written down by a machine.

In §§9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers  $\pi$ ,  $e$ , etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In §8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. [Here there is a footnote referencing Gödel's 1931 paper containing the first incompleteness theorem, which will be discussed in the next chapter of this book.] These results have valuable applications. In particular, it is shown (§11) that the Hilbertian Entscheidungsproblem can have no solution.

At this point, Turing has finished his preview and is ready to launch into detailed definitions. In §1, he defines “computing machines,” which we today call Turing machines.

#### 1. Computing machines.

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions  $q_1, q_2, \dots, q_R$  which will be

called "*m*-configurations" [i.e., "states"]. The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares" [or "cells"]) each capable of bearing a "symbol". At any moment there is just one square, say the *r*-th, bearing the symbol  $S(r)$  which is "in the machine" [today we would say "scanned by the read-write head"]. We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly aware". However, by altering its *m*-configuration [*state*] the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the *m*-configuration [*state*]  $q_n$  and the scanned symbol  $S(r)$ . This pair  $q_n, S(r)$  will be called the "configuration": thus the configuration determines the possible behaviour of the machine.

Some of Turing's terminology seems strange to us, and is noted using square brackets in the excerpt above. But here we must stop for a more careful discussion about terminology, because the word "configuration" has a very different meaning for Turing than it does for us. Most textbooks, including this one, define a "configuration" to include the entire tape contents, as well as the current state, head position, and scanned symbol. For Turing, however, a "configuration" is only the current state and scanned symbol. There is no widely agreed modern equivalent for Turing's "configurations." To make matters worse, Turing will soon define below a new concept called a "complete configuration," which corresponds to the modern definition of "configuration." To summarize, here are the differences in terminology we have so far:

Turing's word or phrase	Modern equivalent
square	cell
in the machine	scanned by the read-write head
<i>m</i> -configuration	state
configuration	current state and scanned symbol
complete configuration	configuration (i.e., tape contents, state, and head position)

Another minor difference is that today we have a special symbol in our alphabet called the blank symbol. As you will see in the next excerpt, Turing thinks of the blank in a more physically realistic way, as the absence of a symbol.

In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the *m*-configuration [*state*] may be changed. Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough

notes to “assist the memory”. It will only be these rough notes which will be liable to erasure.

It is my contention that these operations include all those which are used in the computation of a number. The defense of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by “machine”, “tape”, “scanned”, etc.

Overall, the operation of the machine just defined is remarkably similar to a modern Turing machine. In particular, the head moves at most one square at a time in either direction, and the transition function depends only on the current state and scanned symbol. Turing continues with a more detailed analysis of his machines.

## 2. Definitions.

### *Automatic machines.*

If at each stage the motion of a machine (in the sense of §1) is *completely* determined by the configuration, we shall call the machine an “automatic machine” (or *a-machine*) [*i.e.*, *a deterministic machine*].

For some purposes we might use machines (choice machines or *c-machines* [*i.e.*, *nondeterministic machines*]) whose motion is only partially determined by the configuration (hence the use of the word “possible” in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems. In this paper I deal only with automatic machines, and will therefore often omit the prefix *a-*.

Turing’s insight into nondeterminism here is fascinating, but he doesn’t follow up on it in this paper. His conception of nondeterminism doesn’t involve launching multiple threads simultaneously. Instead, at certain times the machine makes a nondeterministic choice of which single thread to launch, via an external (presumably human) “operator.” This is one of the three equivalent models of nondeterminism discussed on page 158. In any case, the rest of Turing’s paper deals only with deterministic machines, which he calls “automatic,” or *a-machines*.

### *Computing machines.*

If an *a-machine* prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine. If the machine is supplied with a blank tape and set in motion, starting from the correct initial *m*-configuration [*state*], the subsequence of the symbols printed by it which are of the first kind will be called the *sequence computed by the machine*. The real number whose expression as a binary decimal is obtained by prefacing this sequence by a decimal point is called the *number computed by the machine*.

Turing's division of symbols into the "first kind" and "second kind" is a little obscure. The "first kind" is 0s and 1s, and the "second kind" is everything else (e.g., a, b, c). The basic idea is that the machine will write a mixture of the two different kinds of symbols on its tape, such as "abc0def11g0h10ijk0abc...". The plan is to interpret just the 0s and 1s as the output of the computation. The other symbols are used to record extra information that's needed during the computation, but they are not part of the output. For example, the output for a tape containing "abc0def11g0h10ijk0abc..." would be "0110100....". And we interpret this as a numeric output by viewing it as the binary number 0.0110100....

For us, it's easiest to imagine a machine with two tapes: a regular tape used for both input and intermediate results, and an initially blank output tape on which only 0s and 1s are written. Let's call this model a *two-tape binary computing machine*.

In fact, the use of binary here is irrelevant. Our discussion will be simpler using decimal instead. So let's allow the output tape to contain any of the digits 0–9, and call the resulting model a *two-tape decimal computing machine*, or 2TDCM. The output tape of a 2TDCM contains a (possibly infinite) sequence of digits that can always be interpreted as a real number between 0 and 1. For instance, the string "43900023..." corresponds to the number 0.43900023.... The book materials provide a simulator, `simulate2TDCM.py`, which can be used to simulate 2TDCMs in the same way that we used `simulateTM.py` to simulate standard Turing machines in chapter 5.

As we'll soon see, Turing is primarily interested in machines that produce an *infinite* sequence of digits. In fact, for Turing, the only valid computations are those that produce infinitely many digits. So the finite output "384", for example, represents a failed computation and does *not* represent the number 0.384. Instead, the number 0.384 is represented by the infinite sequence "384000...". (And yes, "383999..." would be another valid representation of 0.384.)

At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the *m*-configuration [*state*] will be said to describe the *complete configuration* at that stage. The changes of the machine and tape between successive complete configurations will be called the *moves* of the machine.

We've already been warned about this on page 319: Turing's "complete configuration" is what we simply call a "configuration" today.

Until this point in the paper, all the differences between Turing's computing machines and the modern Turing machine have been cosmetic: they involve different terminology, but the same fundamental workings. The next excerpt, by contrast, is a more serious divergence from the modern point of view.

#### *Circular and circle-free machines.*

If a computing machine never writes down more than a finite number of symbols of the first kind, it will be called *circular*. Otherwise it is said to be *circle-free*.

A machine will be circular if it reaches a configuration from which there is no possible move, or if it goes on moving, and possibly printing symbols of the second kind, but cannot print any more symbols of the first kind. The significance of the term “circular” will be explained in §8.

There is no modern equivalent to the notions of *circular* and *circle-free*, but let’s investigate these ideas more carefully. We’ll continue with the 2TDCM model, so that symbols of the “first kind” (0–9) are written on a special output tape, and all other work is done on a regular tape. Then Turing says a machine is *circle-free* if it writes infinitely many digits on the output tape. Otherwise, the machine writes only a finite number of digits on the output tape, and we call it *circular*.

To a modern reader, it seems completely obvious that the circular machines would be “good” machines, because they produce a finite output. And circle-free machines would be “bad,” because they don’t even terminate and their output is infinite. But Turing had exactly the opposite idea: the whole objective of his machines is to compute the (infinite) representation of a real number as a sequence of digits. Therefore, to Turing, the circle-free machines are “good.” In contrast, a circular machine prints only finitely many digits, so it must have got “stuck,” and is therefore “bad” (from Turing’s point of view).

It’s also essential to note that the modern notion of a machine that “halts” is not equivalent to a machine that is circular. Turing never defines the notion of “halt.” But we, as modern readers, can recognize the concept of “halt” in Turing’s phrase “a configuration from which there is no possible move.” In other words, Turing’s machines have no explicit halting states, but we may assume they transition into a halting state whenever the transition function is undefined. So, as Turing points out, a machine that halts must be circular, but there are other behaviors that can make it circular too. For example, the machine could enter an infinite loop that prints nothing at all, or prints only on the regular tape. Or it could execute some more chaotic nonterminating behavior, as long as it stops writing on the output tape after a finite time. This is what Turing means by the phrase “it goes on moving, and possibly printing symbols of the second kind.”

So Turing’s key definition, circular versus circle-free, is not at all the same thing as halting versus non-halting. Instead, Turing is interested in a particular type of non-halting computation: these are the circle-free computations, which run forever and produce an infinite sequence of output digits representing a real number between 0 and 1. This would be a good time to experiment a little with the provided `simulate2TDCM.py` simulator: implement and test some circle-free and circular 2TDCM machines.

### *Computable sequences and numbers.*

A sequence is said to be computable if it can be computed by a circle-free machine. A number is computable if it differs by an integer from the number computed by a circle-free machine.

We shall avoid confusion by speaking more often of computable sequences than of computable numbers.

Don't be confused by the phrase "differs by an integer." The point here is that computing machines, as defined by Turing, can only compute numbers between 0 and 1. With a tiny change to his definitions (namely, allowing the decimal point to be a symbol of the "first kind"), his machines could have computed numbers outside the interval between 0 and 1. But Turing chose instead to focus only on the part of the number after the decimal point. Thus,  $\pi = 3.14159\dots$  is computable because it is exactly 3 more than the number  $0.14159\dots$ , which can be produced by a circle-free machine.

### *Functions versus numbers*

The modern approach defines computability as a concept that applies to computational problems or functions, not numbers. Psychologically, this is a big difference between Turing and the modern approach—especially because Turing's machines always compute numbers starting from a blank tape (see the excerpt on page 320), so there is no way of providing input to his machines.

Formally, however, the difference between computing functions and numbers is perhaps not so great. This is because numbers can be represented as sequences of symbols, and vice versa. Computational problems map strings to sets of strings, and these can all be encoded as numbers if desired. And although Turing's original definition requires a blank tape, some of his proofs and definitions later in the paper allow the obvious generalization of starting a machine with a non-blank tape.

Alternatively, there's a more direct connection between computable functions and computable numbers. To understand this connection, we consider functions  $f(n)$  mapping positive integers to positive integers. We say that  $f$  is a *total* function if it is defined for all positive integers; otherwise  $f$  is a *partial* function. Now suppose  $M$  is a modern Turing machine that computes a (total or partial) function  $f(n)$  mapping positive integers to positive integers. Note that if  $f$  is a partial function, then  $M$  doesn't halt on all inputs— $M$  halts and returns  $f(n)$  only on inputs  $n$  for which  $f$  is defined.

We can easily use  $M$  to create a Turing-style machine (i.e., a 2TDCM)  $T$  that starts with a blank tape and successively attempts to compute  $f(1)$ ,  $f(2)$ ,  $\dots$ , writing these numbers down on the output tape. It's clear that  $T$  is circle-free if and only if  $M$  always terminates (i.e.,  $M$  computes a total function). Thus, the circle-free problem for  $T$  is essentially the same as the halting problem for  $M$ .

### *Turing, undecidability, and the halting problem*

Let's now debunk a common misconception about Turing's paper. It's often stated that Turing proved the undecidability of the halting problem. But as we now know, Turing never even defined the concept of halting, and he certainly never proved anything about it. What he did prove is a closely related idea: in §8 of the paper, he proves that the question of whether a machine is *circle-free* is undecidable. Circle-free machines don't terminate, but some circular machines don't terminate either. So although there is an easy reduction from the circle-free problem to the halting problem (as described in the previous paragraph), these problems are not the same.

Nevertheless, it's clear that Turing already had a deep understanding of undecidability. For example, Turing does explicitly prove the undecidability of at least one other problem in his §8: this is the question of whether a given machine ever prints a “0”. And Turing's proof works almost verbatim for proving the undecidability of many other problems. So Turing probably knew that undecidability is a widespread phenomenon—just as we discovered in section 7.5 (page 123).

## 15.2 MACHINES CAN COMPUTE WHAT HUMANS CAN COMPUTE

Two of Turing's greatest achievements are the 1936 “On computable numbers” paper and another paper published in 1950, called “Computing machinery and intelligence.” The “Computing machinery” paper is really a work of philosophy rather than computer science or mathematics. In it, Turing considers the question, can machines think? He describes a thought experiment called the *imitation game*, which we now know as the *Turing test*. The game involves a judge conversing by text message with a human and a computer program, and trying to guess which one is human. These days, the Turing test is part of popular culture (especially after the successful 2014 movie, *The Imitation Game*). But what many people don't realize is that even in 1936, before any electronic computers had been designed or built and 14 years before his Turing test paper, Turing was already thinking deeply about the capability of machines to imitate humans.

Indeed, as we will see in the next excerpt, Turing devoted several pages to this topic in “On computable numbers.” We should bear in mind that a major motivation behind this paper was to create a rigorous model describing which numbers could be computed by a human mathematician. Therefore, in §9, Turing tries to justify the equivalence between human mathematicians and the computing machines he defined earlier. Once again, his terminology is foreign to us and must be interpreted carefully. In fact, as you can see from the table below, his use of the word “computer” is the opposite of our modern meaning: to Turing, a “computer” is a *human mathematician performing a calculation*.

Turing's word or phrase	Modern equivalent
computer	human mathematician
machine	computer

With this important terminology in mind, we are ready to tackle §9.

### 9. The extent of the computable numbers.

No attempt has yet been made to show that the “computable” numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue

is “What are the possible processes which can be carried out in computing a number?”...

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child’s arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent†. [A footnote here gives a mathematical argument explaining why the alphabet should be finite.] The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 99999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 99999999999999 and 99999999999999 are the same.

The behaviour of the [*human*] computer at any moment is determined by the symbols which he is observing, and his “state of mind” at that moment. We may suppose that there is a bound  $B$  to the number of symbols or squares which the [*human*] computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be “arbitrarily close” and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Let us imagine the operations performed by the [*human*] computer to be split up into “simple operations” which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the [*human*] computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the [*human*] computer (possibly with a special order), and the state of mind of the [*human*] computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always “observed” squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the [*human*] computer. I think it is reasonable to suppose that they

can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within  $L$  squares of an immediately previously observed square.

In connection with “immediate recognisability”, it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as immediately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find “... hence (applying Theorem 157767733443477) we have ...”. In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other “immediately recognisable” squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. This idea is developed in III below.

The simple operations must therefore include:

- (a) Changes of the symbol on one of the observed squares.
- (b) Changes of one of the squares observed to another square within  $L$  squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

- (A) A possible change (a) of symbol together with a possible change of state of mind.
- (B) A possible change (b) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested on p. 250 [*our page 325*], by the state of mind of the [*human*] computer and the observed symbols. In particular, they determine the state of mind of the [*human*] computer after the operation is carried out.

We may now construct a machine to do the work of this [*human*] computer. To each state of mind of the [*human*] computer corresponds an “ $m$ -configuration” [*state*] of the machine. The machine scans  $B$  squares corresponding to the  $B$  squares observed by the [*human*] computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than  $L$  squares from one of the other scanned squares. The move

which is done, and the succeeding configuration, are determined by the scanned symbol and the *m*-configuration [*state*]. The machines just described do not differ very essentially from computing machines as defined in §2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the [*human*] computer.

This passage is not, of course, a rigorous proof. But it is an extraordinarily persuasive argument that computers can perform any mathematical calculation that could be done by a human. In a sense, Turing has nailed a restricted version of the Turing test, 14 years early. If we insist that the Turing test conversation consists only of questions and answers about mathematical calculations, a computer will be able to do anything the human can do and will pass the test. By 1950, Turing was ready to dramatically expand this claim: in “Computing machinery,” he makes further convincing arguments that computers will eventually imitate humans in conversations about any topic.

### 15.3 THE CHURCH-TURING THESIS: A LAW OF NATURE?

We’ve already seen a bewildering array of claims that certain methods of computation are equivalent to each other. The next few subsections attempt to summarize these claims of equivalence and introduce some new ideas too. Specifically, we will first state a mathematical fact termed “the equivalence of digital computers.” Then we will introduce three related proposals, known as Church’s thesis, Turing’s thesis, and the Church–Turing thesis. Figure 15.1 summarizes these three theses. The Church–Turing thesis is particularly interesting because it can be regarded either as an empirical observation or as a physical law.

#### The equivalence of digital computers

In chapter 5, we saw a detailed argument showing that all past and present digital computers are Turing equivalent (provided they are given sufficient memory). Moreover, these computers are also equivalent to abstract models of computation such as the standard Turing machine, multi-tape Turing machines, and many other variants. Unlike the generalized “theses” discussed below, this equivalence of all existing digital computers and Turing machines is an undisputed mathematical fact. We’ll refer to it simply as the “equivalence of digital computers.”

It’s worth noting that in addition to Turing machines, there are two other famous abstract models of computation known as *lambda calculus* and *recursiveness*. We have not studied these models in this book, but it turns out that they too are equivalent to Turing machines. Both of the alternative models slightly pre-date Turing’s work: the Princeton mathematician Alonzo Church published his ideas on the lambda calculus in 1936 before Turing’s “On computable numbers” appeared, and Church’s student Stephen Kleene published results on recursiveness at around the same time. Turing moved from England to study

<b>Church's thesis</b>	The intuitive notion of “algorithm” corresponds to the precise notion of “computer program” (as implemented, for example, via Turing machines or the lambda calculus).
<b>Turing's thesis</b>	Given a particular human brain $B$ , there exists a computer program $P$ whose outputs are indistinguishable from $B$ . Here, the notion “indistinguishable” is based on the Turing test: when $B$ and $P$ are restricted to textual inputs and outputs, independent human judges cannot guess which is which.
<b>Church–Turing thesis</b>	All physically realizable forms of computation are Turing equivalent.

**Figure 15.1: Three theses about computation, as defined in this book.** Note that the phrases “Church’s thesis,” “Turing’s thesis,” and “Church–Turing thesis” are used interchangeably by most authors. This book suggests some more specific definitions highlighting the different motivations of Church and Turing.

under Church at Princeton in late 1936, after Turing had already written and submitted his work on computability and Turing machines.

In any case, the key point is that the “equivalence of digital computers” encompasses all three abstract models of computation and all known real computers.

### Church’s thesis: The equivalence of computer programs and algorithms

Humans have an intuitive notion of what an “algorithm” is. We often think of an algorithm as a method or process that is guaranteed to produce the answer to a problem. But as Alonzo Church pointed out in his seminal 1936 paper on the lambda calculus, this definition of algorithm is a “somewhat vague intuitive notion.” By the way, Church used the phrase “effectively calculable” to describe what we today refer to as “algorithms.” In the description below, the word “algorithm” is used interchangeably with the notion of an “effectively calculable process.”

In his 1936 paper, Church boldly proposed a formal definition of algorithm. His definition was given in terms of the lambda calculus—but as we know, this can be translated into equivalent definitions based on Turing machines or computer programs. Therefore, a modern paraphrase of Church’s paper could be stated as “an algorithm is the same thing as a computer program.” We will tentatively refer to this claim as “Church’s thesis.” But note that there is no consensus in the computer science literature on a clear definition of Church’s thesis; other authors give somewhat different but equally reasonable definitions. For us, then, Church’s thesis is the claim that the intuitive notion of an algorithm corresponds precisely to the class of computations that can be performed by digital computers (see figure 15.1).

## Turing’s thesis: The equivalence of computer programs and human brains

Church and Turing were both trying to tackle Hilbert’s Entscheidungsproblem, and both succeeded using very different techniques: Church via the lambda calculus and Turing via Turing machines. But quite apart from these technical differences, it seems that Church and Turing perceived different *philosophical* motivations behind the Entscheidungsproblem. In a nutshell, Church was interested in giving a formal definition of effective calculability (i.e., algorithms), whereas Turing was interested in formalizing what could be calculated by *humans*.

As a result, Turing wrote §9 of his paper (on pages 324–327 of this book), explaining why he believed that any mathematical calculation performed by a human could also be performed by a Turing machine. And as mentioned above, Turing expanded this claim 14 years later, arguing in his famous “Turing test” paper that computers could in principle mimic any aspect of human thinking.

Thus, we might be justified in defining “Turing’s thesis” as the claim that computer programs and human brains are equivalent in terms of what they can compute (see figure 15.1). In my experience, this formulation of Turing’s thesis is believed by a reasonable number of AI experts and by some philosophers, but it is rejected by most other people. Note that “Turing’s thesis” is not a standard term in the literature, but it is arguably a useful and accurate label.

## Church–Turing thesis: The equivalence of all computational processes

In contrast, the phrase “Church–Turing thesis” is commonly used in the fields of computer science, artificial intelligence, and philosophy. Unfortunately, there is no consensus on its precise meaning. Usually, the “Church–Turing thesis” is taken to be some combination or generalization of our previous three concepts: the equivalence of digital computers (a proven fact), the equivalence of computers and algorithms (Church’s thesis), and the equivalence of computers and human brains (Turing’s thesis). In my view, the most useful definition of Church–Turing thesis combines all three previous ideas and then adds the further claim that no other type of computation is possible. In more detail, this version of the Church–Turing thesis states that (i) all known physically realistic methods of computation are equivalent (including real computers, abstract computers like Turing machines, human brains, and all other physical systems such as collections of DNA molecules or quantum-mechanical particles), and (ii) no other form of computation is possible (see figure 15.1).

Stated this way, the Church–Turing thesis resembles a physical law such as the law of gravity: it is consistent with all known empirical observations, could in principle be falsified by experiment, and appears to describe an invariant property of the universe. Like other physical laws, the Church–Turing thesis cannot be proved correct, but we can increase our confidence in it over time by performing experiments. Indeed, some physicists have argued that the ideas and mechanisms of computation lie at the heart of physics, somehow governing

and limiting all physical processes. For example, in his book *A New Kind of Science*, Stephen Wolfram proposes a physical law called the “principle of computational equivalence,” which could be viewed as a variant of the Church–Turing thesis. On the other hand, even if we accept the truth of the broadest possible generalizations of the Church–Turing thesis, it’s not clear that this idea provides the predictive power that we usually expect from physical laws.

## EXERCISES

**15.1** Early on in “On computable numbers,” Turing gives examples of machines that compute the alternating sequence 010101... and the sequence of unary numbers 0010110111011110.... Using the .tm file format introduced in figure 5.19 (page 96), create 2TDCM machine descriptions that print each of these sequences. Test your descriptions using the provided `simulate2TDCM.py` simulator. Note that even though these are two-tape machines, the standard .tm file format can still be used for the following reasons: (i) the output tape is never read, so the scanned symbol is always read from the regular tape, and (ii) when a new symbol is written, digits are automatically appended to the output tape and any other symbol is written to the current cell on the regular tape.

**15.2** Describe the action of the following 2TDCM machine:

```

q0->q1: _;z,R
q1->q2: _;x,S
q2->q2: x;1,R
q2->q3: _;L
q3->q4: x;0,S
q4->q4: x;L
q4->q5: z;R
q5->q6: x;w,R
q5->q8: y;L
q6->q6: xy;R
q6->q7: _;y,L
q7->q7: xy;L
q7->q5: w;R
q8->q8: w;L
q8->q9: z;R
q9->q9: wy;x,R
q9->q10: _;L
q10->q10: x;L
q10->q2: z;R

```

**15.3** Give an example of (a) a circular machine that halts (according to our modern definition of “halt”), and (b) a circular machine that does not halt.

**15.4** We noted on page 322 that a 2TDCM can be circular for several different reasons. In particular, it could return to exactly the same configuration twice (and therefore infinitely often), or it could execute looping behavior that is more chaotic. If a circular machine returns to exactly the same configuration twice then let’s say that it has a *simple circularity*; otherwise we’ll call it a *chaotic circularity*. Adapt the provided `simulate2TDCM.py` simulator so that it detects simple circularities. Additionally, implement a heuristic that makes a reasonable guess as to when a chaotic circularity has been encountered. Test your new simulator on various circular and circle-free machines.

**15.5** On page 323, the text mentions a method of converting a modern Turing machine  $M$  into a Turing-style 2TDCM  $T$ . Specifically,  $M$  computes a partial or total function  $f$  from positive integers to positive integers; and  $T$  prints the

sequence  $f(1), f(2), \dots$  on its output tape. Draw a state diagram for  $T$ , using  $M$  as a building block. For convenience, assume  $M$  and  $T$  use unary notation for positive integers.

**15.6** Give an example of an uncomputable number (as defined by Turing), and prove that it is uncomputable.

**15.7** Prove that the question of whether a given 2TDCM ever prints a 0 is undecidable. (This is one of the results that Turing proves in his paper, but we did not study the relevant excerpt.)

**15.8** In Turing's model of a human "computer" (i.e., a mathematician), it is assumed the human can scan  $B$  squares at one time and that the human can move up to  $L$  squares in either direction before scanning any squares at the next step of the computation. But in an abstract Turing machine, we have  $B = L = 1$ . On our page 327, Turing states that a generalized Turing machine with  $B, L > 1$  does not "differ very essentially" from the standard machine with  $B = L = 1$ . Give a formal proof that a standard machine can simulate a generalized machine with  $B, L > 1$ .

**15.9** Do you think a chimpanzee could be trained to execute a given, relatively simple Turing machine? For concreteness, suppose the machine has about five states, five symbols, and a dozen transitions. Could a rat be trained for the same task? If the answer is yes, should we put chimpanzees and rats in the same category as Turing's human computers?

**15.10** Repeat the previous exercise, but instead of *training* a given chimpanzee or a rat, consider the possibility of *evolving* a chimpanzee or rat species to execute a given, relatively simple Turing machine. The evolution would take place over thousands or millions of generations, but the resulting brains should have similar overall size and complexity to present-day chimpanzee and rat brains respectively. Could these brains evolve to execute a given Turing machine? If the answer is yes, should we put chimpanzees and rats in the same category as Turing's human computers?

**15.11** Find three or four documents, drawn from a variety of sources, that each use the phrase "Church–Turing thesis." In each case, try to understand the author's intended definition of the Church–Turing thesis. Compare and contrast these definitions with the definitions in figure 15.1.

**15.12** The Nobel Prize-winning physicist Richard Feynman once delivered a famous series of lectures entitled *The Character of Physical Law*, which were published as a short book in 1967. Familiarize yourself with Feynman's notion of what constitutes a physical law, then answer the following: Would Feynman consider the Church–Turing thesis to be a physical law? Why or why not?

# 16



## YOU CAN'T PROVE EVERYTHING THAT'S TRUE

“But how shall we prove anything?”

“We never shall. We never can expect to prove any thing upon such a point. It is a difference of opinion which does not admit of proof.”

—Jane Austen, *Persuasion* (1818)

The main purpose of this book is to answer the question, what can be computed? In chapter 4, the notion of “compute” was formalized by introducing computational problems. We saw that the tasks done by computers can be reformulated as computational problems. This includes manipulating data (e.g., “sort these numbers into decreasing order”) and answering questions (e.g., “what is the square root of 3.57?”). But another potentially important type of computation—one that can also be reformulated as a computational problem—is to *prove* whether statements are true or false (e.g., “prove that a multiple of 5 always ends in a 0 or a 5”). In this chapter, we discover what it means for a computer to prove something. We then investigate some of the consequences for the foundations of mathematics. We’ll see that computers are powerful tools for generating proofs, but that there are also inherent limits to any proof algorithm: you can’t prove everything that’s true.

### The history of computer proofs

To fully appreciate the ideas in this chapter, we need some historical context. For centuries before the first electronic computers were built, mathematicians and philosophers had dreamed of building computational devices that not only helped with tedious calculations, but could also perform *reasoning*. One of the early proponents of this was Gottfried Leibniz, in the early 18th century. Leibniz is probably most famous as one of the inventors of calculus, but he also suggested the possibility of mechanical reasoning by computers. In the 19th century, George Boole (who gave us Boolean algebra, which is a central component of all programming languages) advanced new ideas in formal logic. And in the early 20th century, David Hilbert (the mathematician whose 10th problem we have

met twice already, on pages 6 and 134) led the charge in proposing that all of mathematics could be reduced to formal, automated reasoning.

Obviously, electronic computers would have helped implement Hilbert's ideas. But in 1931, over 10 years before the emergence of electronic computers, Hilbert's hopes were famously dashed by Kurt Gödel. Gödel proved his *first incompleteness theorem*, showing there were some true mathematical statements that could not be proved. Five years later, in 1936, Turing was working on a follow-up to Gödel's incompleteness theorems when he published his "On computable numbers" paper. As an unexpected side effect of this work, Turing discovered all the main ideas that we explored in part I of this book, and essentially founded the field of theoretical computer science. In this chapter, we reverse the historical flow of ideas, and show how Turing's results on uncomputability can be used to prove a version of Gödel's first incompleteness theorem. In doing so, we will see some links between computability theory and mathematical proof, thus gaining insight into the unplanned birth of CS theory in 1936.

## 16.1 MECHANICAL PROOFS

We have already presented many proofs in this book, but let's pause to analyze the meaning of "proof" a little more carefully. When we use the words "prove" or "proof" in this book, they have technical meanings much stronger than their meanings in everyday English. For example, most people would agree with the statement "smoking causes cancer," and—perhaps after first reading some of the relevant scientific studies—most people would also agree that "it's been proved that smoking causes cancer." But in this book, we always mean proof with absolute certainty: the kind of proofs that mathematicians use to prove a theorem such as "a multiple of 5 always ends in a 0 or a 5" or "there are infinitely many prime numbers."

It turns out that computers can be used to prove theorems like this. To do so, we need to formalize our notion of mathematical proof, by defining the notion of a *proof system*. The first step is to agree on an alphabet of symbols that will be used for making *statements* in the system. In our first example of a proof system, the alphabet will consist of the four symbols  $\{0, 1, =, +\}$ . For reasons explained later, we'll call our proof system *BinAd*. All statements in BinAd will be strings on the alphabet, like  $0=11==+$ ,  $1+1=0$ , and  $10=10$ .

In a proof system, some of the statements are *well formed*, and others are not. In BinAd, for example,  $11+10+1=0+1$  is well formed, but  $0=11==+$  is not well formed. Based on this, you have probably already guessed that the well-formed statements must look like mathematical equations, with a single equal sign surrounded by sums of binary numbers. We can use the regular expressions of section 9.4 to give a precise definition: a BinAd statement is well formed if and only if it matches the pure regex

$$N(+N)^* = N(+N)^*$$

where  $N$  is an abbreviation of the pure regex  $1(0|1)^*$ .

Later, we will need some more terminology related to this matching of regexes. A *maximal N-match* is a substring that matches  $N$  and can't be extended in either direction while still matching. For example, in the string  $S = "111+10+1"$ , the substring “11” is an  $N$ -match, but not a maximal  $N$ -match. In contrast, the strings “111”, “10”, and “1” are all maximal  $N$ -matches in  $S$ . We will also need the concept of a *repeated N-match*. A repeated match must correspond to the same substring in all occurrences within one statement. For example, the regex  $N1+N1$  is matched by  $101+111$  (because each instance of  $N$  can correspond to a different substring). But if we insist on using only repeated  $N$ -matches, then  $N1+N0$  is not matched by  $101+110$ . Instead, it is matched by strings such as  $11+10$  and  $101+100$ .

In a proof system, some of the well-formed statements can be proved, and they are called *provable statements*. In BinAd, it will turn out that  $10=10$  is a provable statement, but  $1+1=0$  is a well-formed statement that isn't provable. A precise definition of provable statements will be given later (page 336).

Notice that the statements of a proof system may or may not have a real-world meaning. Our BinAd example was deliberately crafted with symbols that suggest a certain meaning—so you probably find it quite reasonable that  $10=10$  is provable and  $1+1=0$  isn't provable. On the other hand, we could have used the alphabet {x, y, #, @} instead of {0, 1, =, +}. This would yield the corresponding provable statement  $xy\#xy$  and unprovable statement  $y@y\#x$ , which have no obvious meaning to a human reader. We are encountering here the crucial distinction between *syntax* (precise rules for manipulating strings of symbols that have no meaning) and *semantics* (the meanings that may be attached to strings of symbols). We'll be returning to this later, but for the moment try to keep in mind that both points of view are useful: the string  $10=10$  can be viewed as either (i) a meaningless sequence of symbols produced according to some fixed rules, or (ii) a statement about the real world which could be true or false. When working with computer proofs, it's helpful to think of the computer as dealing with the “easy” problem of syntax, manipulating meaningless sequences of symbols. But a human can look at the proof and interpret it as solving a “hard” problem involving interesting, meaningful semantics.

The next step in defining a proof system is to agree on some *axioms* for the system. Axioms are just statements that are defined to be provable statements, without needing any proof. How do we know the axioms are valid? We don't! This is a problem that mathematicians and philosophers have wrestled with since ancient times, but there's no way around it. To prove something, we will have to assume something else first. It's just a matter of first agreeing on axioms that are reasonable and then seeing what they lead to. In a very few parts of mathematics, there are even competing sets of axioms that lead to different conclusions. But for the vast majority of mathematics, there is wide agreement on the axioms that lead to correct outcomes. In any case, the axioms for BinAd will be particularly simple, since there is exactly one axiom:

$$\text{(axiom)} \quad 1=1.$$

The final step in establishing a proof system is to agree on some *inference rules*. These are rules that allow us to infer a new provable statement from one or more

axioms or statements that have already been proved. In BinAd, there are three inference rules:

Rule 1: given  $A = B$  infer  $1+A = 1+B$ .  
 Rule 2: given  $\dots \underline{N} + \underline{N} \dots$  infer  $\dots N0 \dots$   
 Rule 3: given  $\dots \underline{1} + \underline{N0} \dots$  infer  $\dots N1 \dots$

The underline notation will be explained shortly, and the other notation in these rules is straightforward:  $N$  is the regex  $1(0|1)^*$  given earlier,  $A$  and  $B$  are strings matching the pure regex  $N(+N)^*$ , and the “ $\dots$ ” symbols represent parts of the statement that are unchanged by the inference. A little more fine print is required to make these rules technically correct. First, every occurrence of  $N$  in a given rule must be a repeated  $N$ -match. For example, rule 2 applies to  $10+10=100$ , but not to  $10+11=101$ . Second, each underlined part of a rule must be a maximal  $N$ -match. For example, rule 3 applies to  $10+1+100=111$ , but not to  $101+100=1001$ .

The table below shows an example of applying each rule. Suppose, just for the purpose of these examples, that somehow we had already managed to prove the statement  $1+10+10=101$ . Then we can apply the rules as follows:

Rule	Input	Output	Comments
1	$1+10+10=101$	$1+1+10+10=1+101$	set $A$ to $1+10+10$ and $B$ to $101$
2	$1+10+10=101$	$1+100=101$	set $N$ to $10$
3	$1+10+10=101$	$11+10=101$	set $N$ to be the second 1 of the input

The Python program `binAd.py`, provided with the book resources, contains implementations of these rules. Feel free to experiment by loading this program and then executing Python commands such as

```
>>> applyRuleTwo('1+1+1=11')
```

Make sure there is no whitespace in the statements you experiment with. In BinAd, well-formed statements may not contain whitespace!

Now we are ready for some formal definitions:

**Definition of a proof system.** A *proof system* consists of an alphabet, a list of axioms, a list of inference rules, and a rule defining the well-formed statements. Any string on the alphabet is called a *statement*. Each axiom must be a well-formed statement. Each inference rule must be a computable function mapping a tuple of well-formed statements to a collection of one or more well-formed statements. The rule for determining whether statements are well formed must also be a computable function.

This definition is rather abstract, which is why we presented the specific example of BinAd first. The definition is also deliberately vague on issues such as whether the lists of axioms and rules must be finite, and whether the inference rules must produce finite collections of new statements. Logicians have studied all of these possibilities, but in this book it is enough for us to consider only simple examples such as BinAd. Note the restriction that the inference rules

and well-formedness rule must be computable functions: this means they can be implemented with computer programs that always terminate. Thinking back to our original objective of mechanizing mathematical proofs, this makes a lot of sense.

**Definition of a mechanical proof.** Given a proof system, a *mechanical proof* of a statement  $S$  in the system is a sequence of statements ending in  $S$ , where every statement in the sequence is either an axiom, or follows from an inference rule applied to statements earlier in the sequence.

We'll look at an example of this after the next definition:

**Definition of a provable statement.** Given a proof system, a statement  $S$  is *provable* in the system if  $S$  has a mechanical proof in the system.

Now let's look at an example of a mechanical proof. Here's a mechanical proof of the BinAd provable statement  $1+1+1=11$ :

Provable statement 1:	$1=1$	(axiom)
Provable statement 2:	$1+1=1+1$	(rule 1 applied to provable statement 1)
Provable statement 3:	$1+1=10$	(rule 2 applied to provable statement 2)
Provable statement 4:	$1+1+1=1+10$	(rule 1 applied to provable statement 3)
Provable statement 5:	$1+1+1=11$	(rule 3 applied to provable statement 4)

Note that although it's not part of the definition, mechanical proofs are usually accompanied by a description of which rules were applied to which statements at each step—this makes it easier to check the proof. The provided program `binAd.py` can automatically generate BinAd proofs. Experiment now using commands such as the following:

```
>>> isProvable('1+1+1=11')
>>> isProvable('1=0')
```

## Semantics and truth

You have probably guessed by now the intended meaning of BinAd strings: they are statements about binary addition (and this also explains the name, which is a contraction of *Binary Addition*). So a string such as  $101$  represents the number 5, and a statement such as  $11+10=101$  represents the notion that  $3 + 2 = 5$ . Notice how this gives us a way of recognizing whether a statement in BinAd is true: we just interpret the statement as standard arithmetic, and declare it to be true if it's a true statement about numbers. So  $11+10=101$  is true, but  $11+10=111$  is not. (And by the way, a statement like  $11+=+10=$  isn't true either—it's not even well formed.)

This helps to clear up another mystery: until now, our discussion of mechanical proofs has not mentioned the words “true” or “truth” at all! Why not? Because the definitions of *provable statement* and *true statement* are completely separate. Provability depends only on syntax: Can the statement be derived from the axioms by applying syntactical inference rules? Truth depends only on semantics: Is the statement true, when interpreted with the intended meaning of the symbols?

By definition, proof systems are completely defined by their syntax (i.e., their axioms and inference rules). When we add in some semantics, we get a new type of system that we'll call a *logical system*. For example, when we add the semantics of binary addition to the proof system BinAd, we get a logical system that we'll call BinAdLogic. Formally, we have the following definition:

**Definition of a logical system.** A *logical system* consists of a proof system and a truth assignment. A *truth assignment* is a function that maps well-formed statements to the values {true, false}. The truth assignment function need not be computable. The truth assignment may be defined for all well-formed statements or for some subset of these, such as closed well-formed statements (see page 341).

From now on, we'll consider the relationship between truth and provability in logical systems. To check your understanding that these two concepts are not the same thing, try the following exercise:

Consider the BinAdLogic statement  $10+100=110$ . (a) Is it true? (b) Is it provable?

The answers are given in a footnote below, but don't read it until you have gained the benefit of constructing your own solutions.<sup>1</sup>

These crucial concepts can be understood even more deeply via Python experiments. The provided program `binAd.py` has a function `isTrue()` that determines whether a given BinAdLogic statement is true. Examine the source code and compare it with `isProvable()`—note the starkly contrasting approach of the two functions. Experiment with a few commands such as

```
>>>.isTrue('1+1=10')
>>>isProvable('1+1=10')
>>>.isTrue('10000=10000')
>>>isProvable('10000=10000')
```

In BinAdLogic, it turns out that all true statements are provable and all provable statements are true. But in other logical systems, any other combination of truth and provability is possible. For example, there can be false provable statements and true unprovable statements. This is because we are free to dream up any syntax and semantics that we want, so there's no reason to expect truth and provability will agree with each other.

Let's look at specific examples of this by defining two more logical systems:

- **BrokenBinAdLogic.** The same as BinAdLogic, but with an extra inference rule:

Rule 1B: given  $A = B$  infer  $1+A = B$ .

- **RestrictedBinAdLogic.** The same as BinAdLogic, but with rule 3 removed.

<sup>1</sup> The answer to both questions is yes. But the answer to (a) is produced by interpreting the statement as  $2 + 4 = 6$ , whereas the answer to (b) is produced by starting with the only available axiom and applying inference rules to eventually obtain  $10+100=110$ .

As you'll find out in the following exercise, these new logical systems provide simple examples of the distinction between "provable" and "true":

- (a) Produce a `BrokenBinAdLogic` statement that is provable, but not true.
- (b) Produce a `RestrictedBinAdLogic` statement that is true, but not provable.

As usual, your understanding will benefit immensely from solving these puzzles on your own. But for full clarity, here are some solutions. For (a), the extra rule 1B lets us infer  $1+1=1$  directly from the axiom  $1=1$ . So  $1+1=1$  is provable, but it's obviously not true according to the interpretation of binary arithmetic. For (b), there is no way to prove true statements like  $1+10=11$ , because without rule 3 we can never replace 0s with 1s.

## Consistency and completeness

A logical system is *consistent* if all provable statements are true; otherwise it's *inconsistent*. It's not hard to see that `BinAdLogic` is consistent. Exercise 16.9 asks you to prove this rigorously, but the basic idea is to show that every inference rule maps true statements to true statements. On the other hand, as we saw above, `BrokenBinAdLogic` is inconsistent since you can prove untrue statements like  $1+1=1$ .

Another important property of logical systems is *completeness*. A logical system is *complete* if all true statements are provable; otherwise it's *incomplete*. It can be shown that `BinAdLogic` is complete—exercise 16.10 leads you through a proof of this. In contrast, `RestrictedBinAdLogic` is incomplete since you can't prove true statements like  $1+10=11$ .

It's important to realize that completeness and consistency depend on both syntax and semantics. To emphasize this, let's consider one more example of a logical system, this time with some different semantics:

- **FixedBinAdLogic.** The same as `BrokenBinAdLogic`, but with different semantics. We still interpret the statements as addition of binary numbers, but the `=` symbol is interpreted as meaning "greater than or equal to." For example,  $1+1=1$  is true in this system, but  $1=1+1$  is false. (Note that we could have switched to an alphabet that uses " $\geq$ " instead of "`=`". This would make it easier to recognize the meaning of the statements. But let's instead stick with `=`, which will reinforce the concept that the symbols are arbitrary and can be assigned any desired meaning.)

It's not hard to see that `FixedBinAdLogic` is consistent. As with `BrokenBinAdLogic`, rule 1B lets us prove statements like  $10+10=10$ . But rule 1B always adds more 1s to the left of the equal sign than to the right. Hence, for any provable statement, the left-hand side (interpreted as a sum of binary numbers) is always greater than or equal to the right-hand side—and this agrees with `FixedBinAdLogic`'s semantic definition of a true statement.

To summarize, `BrokenBinAdLogic` and `FixedBinAdLogic` have identical syntax (i.e., identical axioms and inference rules), but different semantics. `BrokenBinAdLogic` is inconsistent, whereas `FixedBinAdLogic` is consistent. So the semantics make a difference!

This book	Some other books
statement (p. 333)	formula
proof system (p. 335)	formal system
mechanical proof (p. 336)	proof or formal proof
provable statement (p. 336)	theorem
logical system (p. 337)	formal system with a model
truth assignment (p. 337)	simplified version of a “model”
true (p. 337)	valid
consistent (p. 338)	sound
closed well-formed statement (p. 341)	sentence

**Figure 16.1: Terminology for logical systems.** Formal logic books use a wide variety of terminology. Some of the most common alternatives are listed here.

By the way, logic textbooks are notoriously inconsistent (*definitely* no pun intended) in their use of terminology. Figure 16.1 summarizes a few of the competing terms, which may help if you explore this material further in other sources.

## Decidability of logical systems

In addition to the questions of completeness and consistency, it turns out there is another interesting question we can ask about our logical systems: Are they *decidable*? We already know what “decidable” means for decision problems: it means that we can solve the problem with a computer program that always terminates, correctly outputting “yes” or “no” as the answer (see page 62). We can now extend this notion by declaring that a logical system  $L$  is *decidable* if its “truth problem” is decidable:

**Definition of the truth problem of a logical system.** Let  $L$  be a logical system. The *truth problem* of  $L$  is the following decision problem. Input: a statement  $S$ , which is a string on  $L$ ’s alphabet. Solution: “yes” if  $S$  is true, and “no” otherwise.

So, a logical system  $L$  is decidable if there’s an algorithm that decides whether any statement is true in  $L$ . (An important warning is in order here. Some authors define a system to be decidable if there’s an algorithm that decides whether any statement is *provable*. And as we now know, deciding truth and provability are two different things. In this book, decidability of a logical system will always refer to deciding truth, not provability.)

Let’s examine some examples of decidability for logical systems. For BinAdLogic, it’s easy to write a program that decides truth—see the function `isTrue()` in the provided program `binAd.py`. The program first checks that the input is well formed, which in this case amounts to matching a regular expression. The program immediately outputs “no” if the statement isn’t well formed. Otherwise the program adds up the integers on the left- and right-hand sides of the equal sign. If the results are equal, the output is “yes” and otherwise “no.” Hence, BinAdLogic is decidable.

	Complete	Consistent	Decidable
<b>BinAdLogic</b> (rules 1, 2, 3)	✓	✓	✓
<b>BrokenBinAdLogic</b> (rules 1, 1B, 2, 3)	✓	✗	✓
<b>RestrictedBinAdLogic</b> (rules 1, 2)	✗	✓	✓
<b>FixedBinAdLogic</b> (rules 1, 1B, 2, 3, and different semantics)	✓	✓	✓
<b>Peano arithmetic</b>	✗	assumed	✗

**Figure 16.2:** Summary of completeness, consistency, and decidability for the logical systems described in this chapter.

Using essentially the same reasoning, it's easy to see that the three other logical systems described so far (BrokenBinAdLogic, RestrictedBinAdLogic, FixedBinAdLogic) are also decidable. Figure 16.2 summarizes the completeness, consistency, and decidability of each of these systems. The figure also includes a line for Peano arithmetic. This is the logical system underlying essentially all of mathematics. As you can see, Peano arithmetic is both incomplete and undecidable! In the next section, we will see proofs of these surprising results.

## 16.2 ARITHMETIC AS A LOGICAL SYSTEM

The logical systems we've seen so far are just mathematical curiosities. What we really want to do is define logical systems that correspond to significant chunks of the real world, and then let computers do the hard work of proving which facts are true. Mathematicians, philosophers, and computer scientists have studied many useful logical systems since the late 1800s, but the only one we will need is called *Peano arithmetic*. Peano arithmetic consists of statements about integers, addition, and multiplication. We won't be studying any of the axioms, inference rules, or semantics for Peano arithmetic—it's reasonable for us to believe that mathematicians have worked these out in detail, and they correspond to our usual intuitions about adding and multiplying integers.

If you're curious, figure 16.3 provides a small glimpse into the details of how Peano arithmetic can represent statements of arithmetic as ASCII strings. The details are unimportant for our later proofs, and a single example should help to convey the basic idea. Consider the statement “there do not exist integers  $x$  and  $y$  such that  $2x = y$  and  $y = 5$ .” Our conventions for converting Peano arithmetic to ASCII translate this statement as

NOT EXISTS  $x, y$ ,  $0''' * x = y$  AND  $y = 0''''''$ .

It turns out that from these basic elements, one can also express concepts such as functions, iteration, inequalities, and most other statements about numbers. The list below gives some examples of statements that can be represented in

Usually, the alphabet for Peano arithmetic contains typical logic symbols like  $\forall$ ,  $\exists$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ ,  $\Rightarrow$ . But for our purposes, it's easier to use the ASCII alphabet, so we represent these logic symbols with FORALL, EXISTS, AND, OR, NOT, IMPLIES. All other standard Peano arithmetic symbols can be translated into ASCII too, for example representing the multiplication  $x \times y$  as  $x * y$ .

Natural numbers are represented in the following strange way: 0 is 0, 1 is 0', 2 is 0'', 3 is 0''', and so on. More generally,  $x'$  represents  $x + 1$  for any natural number  $x$ . The axioms include statements such as “FORALL  $x$ ,  $x+0=x$ ”, “FORALL  $x, y$ ,  $x+y'=(x+y)'$ ”, and “FORALL  $x, y$ ,  $x*y'=x*y+x$ ”.

The inference rules include the famous *modus ponens* rule: given “ $A$ ” and “ $A \text{ IMPLIES } B$ ”, we can infer “ $B$ ”.

**Figure 16.3:** Some optional details about the logical system of Peano arithmetic.

Peano arithmetic. In other words, all of the following statements can be written down as strings of ASCII characters, and we have a set of mechanical rules for trying to prove them:

- $2 + 3 = 5$ .
- For all integers  $x$ ,  $x^2 + 3 > 2$ .
- 3 and 5 are both factors of 30.
- 47 is prime.
- Let  $f(x) = 2x + 1$ . Then  $f^5(0) = 31$ . (Here,  $f^5$  means  $f$  iterated 5 times.)
- Let  $f(x) = x + 3$ . Then for all integers  $x$ ,  $f(x) < f(x + 1)$ .
- Let  $f(x) = x + 3$ . Then for all positive integers  $y$ , there exists a positive integer  $x$  such that  $f(x) = y$ .

Note that Peano arithmetic statements can be true or false. In the list above, the first six statements are true, but the last one is false.

Although we don't study the details, there is of course a definition of well-formed Peano arithmetic statements. We also need one more technical definition: a well-formed Peano arithmetic statement is *closed* if all of its variables are quantified by “for all” or “there exists.” For example, the statement “for all  $x, y$ ,  $2x + 4y + 1$  is odd” is closed because all of the variables are quantified (in this case, by the “for all” operator). But the statement “ $2x = 6$ ” is not closed, because the  $x$  is not quantified. This distinction is important, because if a statement is not closed, we can't tell if it's true. For example,  $2x = 6$  is true if  $x = 3$  and false otherwise. That's why we will consider only closed statements when we discuss the truth of Peano arithmetic statements later on.

### Converting the halting problem to a statement about integers

Once we accept that Peano arithmetic can express standard mathematical statements like the ones listed above, we can move surprisingly quickly to expressing the outputs of computer programs in Peano arithmetic too. The next claim shows

how to do this for one particular class of statements about computer programs that we will need later.

**Claim 16.1.** Let  $P$  be a computer program (expressed, for example, as a Python program or a Turing machine). Then the statement  $S$  given by

“ $P$  halts on empty input”

can be translated into an equivalent statement  $S'$  about integers, and  $S'$  can be written down as an ASCII string in Peano arithmetic. (Here, “equivalent” means that  $S'$  is true if and only if  $S$  is true.)

*Proof of the claim.* Since we haven’t studied the formal axioms and inference rules of Peano arithmetic, we don’t give a rigorous proof based on the axioms. But the proof is rigorous if we assume Peano arithmetic is capable of representing certain elementary arithmetical operations, as in the list of examples on the previous page.

This argument is easiest to state for a standard Turing machine. Recall that at any instant during its execution, a Turing machine is completely described by its configuration, consisting of internal state, head position, and tape contents. We could specify a way of describing these configurations via ASCII strings, perhaps something like  $q_3 : AAG^T G$  to describe a machine in state  $q_3$  with tape contents  $AAGTG$  and the head over the  $T$ .

ASCII strings can easily be converted into equivalent binary strings, of course. So let’s now assume the Turing machine configuration can be uniquely specified by a binary string. The transition function of the Turing machine takes one of these configurations (a binary string) as input, and outputs the new configuration (also a binary string). Now, we can interpret these binary strings as integers if we want to. From this point of view, the transition function is a function from the integers to the integers. Let’s call this function  $Step$ , since evaluating this function once is equivalent to performing one step in the Turing machine’s computation. Note that  $Step$  can be written down in Peano arithmetic. It’s actually a reasonably simple function that depends only on a few bits of its input—the internal state, and the tape symbol at the head position.

Let’s now suppose that  $m_0$  is the integer representing our Turing machine in its initial state with a blank tape. After one step of computation, the integer representing the Turing machine’s configuration is  $Step(m_0)$ . After two steps, it is  $Step(Step(m_0))$ , which can also be written  $Step^2(m_0)$ . After  $N$  steps, the Turing machine’s configuration is represented by  $Step^N(m_0)$ . This, too, can be written in Peano arithmetic.

We are now close: we need to translate into Peano arithmetic the statement that “there exists an integer  $N$  such that  $Step^N(m_0)$  represents a halting configuration.” So the only remaining problem is to express “represents a halting configuration.”

But this is easy. We define another function,  $Done$ , which takes as input an integer  $m$ , and outputs either 1 or 0. If  $m$ , when written in binary, represents a configuration in a halting state of our Turing machine,  $Done$  outputs 1. Otherwise  $Done$  outputs 0. Again,  $Done$  is a simple function depending on only

**PROBLEM PROVABLEINPEANO**

- **Input:** An ASCII string  $S$  representing a statement in Peano arithmetic.
- **Solution:** “yes” if  $S$  is provable from the axioms and inference rules of Peano arithmetic, and “no” otherwise. If  $S$  is not a closed well-formed statement, the solution is also “no”.

**PROBLEM TRUEINPEANO**

- **Input:** An ASCII string  $S$  representing a statement in Peano arithmetic.
- **Solution:** “yes” if  $S$  is true according to the semantics of Peano arithmetic, and “no” otherwise. If  $S$  is not a closed well-formed statement, the solution is also “no”.

**Figure 16.4:** Description of the computational problems PROVABLEINPEANO and TRUEINPEANO.

a few bits of its input (it just examines the Turing machine’s internal state), and it can be written in Peano arithmetic. Hence, the original statement  $S$ , “ $P$  halts on empty input,” can be translated into Peano arithmetic as

$$\text{there exists an integer } N \text{ such that } \text{Done}(\text{Step}^N(m_0)) = 1.$$

When converted to ASCII, this is the desired  $S'$ , and the claim is proved.  $\square$

By examining the above proof, it becomes clear that the construction of  $S'$  from  $S$  is achieved by a definite algorithmic procedure that could, in principle, be implemented in a computer program. Let us call this program `convertHaltToPeano.py`. The book materials do not provide source code for this program, since it would be long, tedious, and messy. Nevertheless, the above proof shows that we could write this program if desired. In the remainder of this chapter, we will feel free to import the `convertHaltToPeano` function into Python programs whenever it is needed.

### Recognizing provable statements about integers

Figure 16.4 defines two important decision problems: PROVABLEINPEANO and TRUEINPEANO. Given a statement  $S$  in Peano arithmetic, these problems ask whether  $S$  is provable and/or true. It turns out that both problems are undecidable, and only one of them (PROVABLEINPEANO) is recognizable. We will be proving some, but not all, of these facts. Let’s start with the claim that PROVABLEINPEANO is recognizable:

**Claim 16.2.** PROVABLEINPEANO is recognizable.

*Proof of the claim.* It’s easy to see that the program `provableInPeano.py`, shown in figure 16.5, recognizes PROVABLEINPEANO. The basic idea is to iterate

```

# The following import is NOT an oracle—it's a computable function.
2 from isPeanoProof import isPeanoProof
3 def provableInPeano(inString):
4     proofString = ''
5     while True:
6         if isPeanoProof(proofString, inString)=='yes':
7             return 'yes'
8         proofString = utils.nextASCII(proofString)

```

**Figure 16.5:** Python program `provableInPeano.py`.

over all possible ASCII strings, testing to see whether each one is a proof of the desired statement. The ASCII strings are enumerated in *shortlex* order: first the string of length 0 (i.e., the empty string), then all strings of length 1, then all strings of length 2, and so on. This shortlex ordering is provided by the `nextASCII` function in the `utils` package.

The details of the test `isPeanoProof` at line 6 are simple in principle: it's simply a matter of checking whether each line in the proof is an axiom or follows from the earlier lines by one of the inference rules, and also checking that the last line of the proof is the same as the given `inString`. However, the implementation is messy so the book materials do not provide source code for `isPeanoProof.py`.

If `inString` is provable, its proof will eventually be discovered and the program returns “yes”. If `inString` is not provable, the program runs forever. But that's not a problem, since the definition of recognizability doesn't require termination on negative instances. □

The above claim is interesting in its own right, but it was included here mostly because of the proof technique it uses. The key trick is to enumerate all possible strings (ordered from shortest to longest) on a given alphabet, checking each string to see if it has a given property. We will use this technique again in our proof of Gödel's theorem.

## The consistency of Peano arithmetic

Is Peano arithmetic consistent? This is a complex and debatable question that is beyond the scope of this book. But there is an easy way out: we may as well assume Peano arithmetic is consistent, because otherwise the mathematics that underlies all of the technology in modern society would be inconsistent. And that would contradict the empirical observation that math seems to work in the real world. Hence, we follow the vast majority of professional mathematicians in making the following assumption:

**Assumption that Peano arithmetic is consistent.** It is reasonable to assume that Peano arithmetic is consistent. Therefore, any statement  $S$  that is provable in Peano arithmetic is also true in Peano arithmetic.

```

1   from trueInPeano import trueInPeano # oracle function
2
3   # The following import is NOT an oracle—it's a computable function.
4   # See page 343.
5   from convertHaltToPeano import convertHaltToPeano
6
7   def haltsViaPeano(inString):
8       haltInPA = convertHaltToPeano(inString)
9       return trueInPeano(haltInPA)

```

**Figure 16.6:** Python program `haltsViaPeano.py`.

We will be making use of this assumption to prove our next major result, that mathematics itself is undecidable.

## 16.3 THE UNDECIDABILITY OF MATHEMATICS

We saw at the start of this chapter that up until the start of the 20th century, mathematicians such as Leibniz, Boole, and Hilbert imagined that it might be possible to determine the truth of mathematical statements via a mechanical procedure. But it turns out that the existence of undecidable computational problems leads directly to the existence of undecidable mathematical statements. The next claim demonstrates this.

**Claim 16.3.** Peano arithmetic is undecidable.

*Proof of the claim.* We need to show that `TRUEINPEANO` (figure 16.4, page 343) is undecidable. Recall that `TRUEINPEANO` takes as input a Peano arithmetic statement  $S$ . The solution is “yes” if and only if  $S$  is true in Peano arithmetic.

We will show that `TRUEINPEANO` is undecidable by reducing `HALTSONEMPTY` to `TRUEINPEANO`. So let program  $P$  be an arbitrary instance of `HALTSONEMPTY`. Using claim 16.1 (page 342), convert the statement “ $P$  halts on empty input” into an equivalent statement  $S$  in Peano arithmetic. Clearly,  $P$  halts if and only if  $S$  is true, so the reduction is complete. A Python version of this reduction is given in figure 16.6  $\square$

So, Peano arithmetic—and therefore mathematics as a whole—is undecidable! There is no algorithm for determining whether a given mathematical statement is true or false. What about provability? Is this undecidable too? A very similar proof shows that it is:

**Claim 16.4.** `PROVABLEINPEANO` is undecidable.

*Proof of the claim.* Given a program  $P$ , we write  $H(P)$  for the corresponding statement of Peano arithmetic meaning “ $P$  halts on empty input.” First we will show that if  $H(P)$  is true, then it’s provable. This is not the case for *any* Peano statement—as we’ll see in the next section—but it is true for the special case of statements of the form  $H(P)$ . Why? Because we just have to simulate

$P$  until it halts. Or more explicitly in terms of Peano arithmetic, we calculate  $\text{Done}(\text{Step}^N(m_0))$  for increasing values of  $N$ , until the value 1 is obtained.

Now we can reduce `HALTSONEMPTY` to `PROVABLEINPEANO`, using the program `haltsViaPeano.py` (figure 16.6) except with `trueInPeano` replaced by an oracle function for `provableInPeano`. This works because truth and provability happen to coincide for  $H(P)$ .  $\square$

It's worth making a brief historical remark here. `PROVABLEINPEANO` and `TRUEINPEANO` are closely related to the famous *Entscheidungsproblem* posed by David Hilbert. (A completely precise definition of the Entscheidungsproblem would take us too far afield, but `TRUEINPEANO` is a reasonable approximation.) Alan Turing was working on the Entscheidungsproblem when he developed his ideas on computability, and he proved the Entscheidungsproblem undecidable in his famous 1936 paper. So Turing's paper contains a result similar to our previous two claims. (And, since history matters, let's also note here that Turing was not the first to prove the undecidability of the Entscheidungsproblem—that honor goes to Alonzo Church, who we have already met in the previous chapter. Church successfully attacked the problem using an approach very different to Turing's.)

## 16.4 THE INCOMPLETENESS OF MATHEMATICS

Turing's ideas also lead to surprisingly simple proofs of a result first proved by Kurt Gödel in the 1930s, known as Gödel's first incompleteness theorem. We can loosely state the theorem as "you can't prove everything," or a little more accurately as "there are true statements about the integers that can't be proved from the axioms." The moral of the theorem is that we can't use computers to prove all the true statements in mathematics—even in supposedly simple areas of mathematics such as adding and multiplying integers! The remainder of this section closely follows the exposition in Sipser's book, translating Sipser's pseudocode programs into Python.

Turing's ideas have such rich applications in this area that we will in fact give two versions of the incompleteness theorem, with two different proofs. The first version is *nonconstructive*. This means the proof does not actually construct a true, unprovable statement. It merely shows that such statements must exist.

**Claim 16.5.** (Nonconstructive version of Gödel's first incompleteness theorem). Peano arithmetic is *incomplete*: there exist true statements in Peano arithmetic that cannot be proved from the axioms and inference rules of Peano arithmetic.

*Proof of the claim.* We assume Peano arithmetic is complete and argue for a contradiction. Specifically, we will show that if Peano arithmetic is complete, then we can solve `HALTSONEMPTY`, which will contradict the undecidability of `HALTSONEMPTY`.

So, let  $P$  be an arbitrary instance of `HALTSONEMPTY`. We use  $P$  as the input to the program `haltsViaCompletePeano.py` shown in figure 16.7. Line 6 uses our now-familiar conversion to produce a Peano statement `haltInPeano` meaning " $P$  halts on empty input." Line 7 creates another statement `notHaltInPeano` meaning " $P$  doesn't halt on empty input." Exactly one of

```

# The following two imports are computable functions, NOT oracles.
2 from isPeanoProof import isPeanoProof
3 from convertHaltToPeano import convertHaltToPeano
4
5 def haltsViaCompletePeano(inString):
6     haltInPeano = convertHaltToPeano(inString)
7     notHaltInPeano = 'NOT ' + haltInPeano
8     proofString = ''
9     while True:
10        if isPeanoProof(proofString, haltInPeano)=='yes':
11            return 'yes'
12        if isPeanoProof(proofString, notHaltInPeano)=='yes':
13            return 'no'
14        proofString = utils.nextASCII(proofString)

```

**Figure 16.7:** Python program `haltsViaCompletePeano.py`.

```

# The following import is NOT an oracle function. The function
2 # provableInPeano correctly recognizes provable strings, but does not
# decide them. See claim on page 343.
4 from provableInPeano import provableInPeano

6 # The following import is a computable function—NOT an oracle.
# See page 343.
8 from convertHaltToPeano import convertHaltToPeano

10 def godel(inString):
11     godelProg = rf('godel.py')
12     haltInPeano = convertHaltToPeano(godelProg)
13     notHaltInPeano = 'NOT ' + haltInPeano
14     if provableInPeano(notHaltInPeano) == 'yes':
15         return 'halted' # any value would do
16     else: # This line will never be executed! But anyway...
17         utils.loop() # deliberate infinite loop

```

**Figure 16.8:** Python program `godel.py`.

these statements must be true. We assumed Peano arithmetic is complete, and therefore exactly one of these statements has a proof in Peano arithmetic. So, the rest of the program iterates over all possible ASCII strings, checking each one to see if it is a proof of either `haltInPeano` or `notHaltInPeano`. Because one of them must have a proof, the program is guaranteed to terminate, correctly deciding whether  $P$  halts on empty input. Thus, we have solved HALTSONEMPTY and produced the desired contradiction.  $\square$

Now we move on to proving a constructive version of this same result. Our constructive proof of Gödel's theorem will rely heavily on the Python program

`godel.py` in figure 16.8. Let's step through this program before attempting a proof of the theorem. Line 4 imports the function `provableInPeano`, which was discussed earlier (see figure 16.5). Recall that `provableInPeano(S)` is guaranteed to return “yes” if  $S$  is provable, but otherwise it doesn't terminate.

Line 8 imports the function `convertHaltToPeano`. This function would be complicated and tedious to implement in practice, but claim 16.1 (page 342) guarantees that it could be implemented if desired.

The main part of `godel.py` begins at line 11, where the program's own source code is stored into a string variable `godelProg`. At line 12, this source code is converted into an equivalent ASCII string of Peano arithmetic. Specifically, the variable `haltInPeano` contains a statement about integers equivalent to “`godel.py` halts on empty input.”

This statement is negated at line 13, so the variable `notHaltInPeano` contains a statement about integers equivalent to “`godel.py` doesn't halt on empty input.” We summarize this for later reference:

notHaltInPeano means “`godel.py` doesn't halt on empty input.”

(★)

Things get interesting at line 14. If `notHaltInPeano` is provable, the `provableInPeano` function is guaranteed to return “yes”, so the program returns “halted” at line 15. If `notHaltInPeano` is not provable, then `provableInPeano` enters an infinite loop (see figure 16.5). So the `else` clause at line 16 is in fact unnecessary, since it will never be executed. But for clarity, we show the program entering an infinite loop at line 17 anyway.

It is strongly recommended that you pause here and work out for yourself what is the actual behavior of `godel.py` on empty input. Specifically, does the program halt or not? The answer will be revealed in the next claim, but your understanding will be greatly increased if you can work out the answer on your own.

**Claim 16.6.** The program `godel.py` in figure 16.8 does not halt on empty input.

*Proof of the claim.* We assume `godel.py` *does* halt on empty input, and argue for a contradiction. The only way for the program to halt is at line 15, which is executed only if

$$\text{provableInPeano}(\text{notHaltInPeano}) == \text{'yes'} \quad (\dagger)$$

on the previous line. But  $(\dagger)$  is equivalent to saying that `notHaltInPeano` is provable. And by our assumption of consistency (page 344), provable statements are true in Peano arithmetic. Thus, `notHaltInPeano` is true.

But now look back to the meaning of `notHaltInPeano`, in the box labeled  $(\star)$  above. The fact that `notHaltInPeano` is true tells us that `godel.py` doesn't halt on empty input, contradicting our initial assumption and thus completing the proof.  $\square$

We are now ready to state and prove the constructive version of Gödel's theorem:

**Claim 16.7.** (Constructive version of Gödel's first incompleteness theorem). Peano arithmetic is *incomplete*: there exist true statements in Peano arithmetic that cannot be proved from the axioms and inference rules of Peano arithmetic. Specifically, the string of Peano arithmetic stored in the variable `notHaltInPeano` at line 13 of `godel.py` is an example of a true but unprovable statement in Peano arithmetic.

*Proof of the claim.* We need to show (a) `notHaltInPeano` is true, and (b) `notHaltInPeano` is unprovable. For (a), note that, from the previous claim, we know `godel.py` doesn't halt on empty input. Referring back to box (★), this immediately tells us that `notHaltInPeano` is true, completing part (a) of the proof.

For (b), we again use the fact that `godel.py` doesn't halt on empty input. This means that at line 14, the function call

```
provableInPeano(notHaltInPeano)
```

does not return "yes". But this immediately implies that `notHaltInPeano` is not provable, completing part (b) of the proof.  $\square$

For historical accuracy, we should note that the two versions of Gödel's theorem given above are significantly simplified, compared to the result Gödel actually proved. Gödel's original result was not restricted to Peano arithmetic; it applied to a wide range of logical systems.

## 16.5 WHAT HAVE WE LEARNED AND WHY DID WE LEARN IT?

It's important to grasp the big picture of this chapter, so let's try to summarize it very compactly:

- Gödel proved his incompleteness theorems in the early 1930s as a response to Hilbert's desire to formalize mathematical proof. Gödel's first incompleteness theorem tells us that, given any reasonable system of axioms and inference rules for mathematics, there exist true statements that cannot be proved from the axioms.
- Later on in the 1930s, Alan Turing was trying to extend Gödel's results and solve a famous problem posed by Hilbert—a problem known as the Entscheidungsproblem, which is similar to the computational problem `TRUEINPEANO` (figure 16.4, page 343). While working on this problem, Turing came up with the idea of Turing machines and proved there exist undecidable problems that these machines cannot solve. Turing's ideas not only demonstrated the impossibility of solving the Entscheidungsproblem, but also became the foundation of computational theory (in conjunction with the ideas of Church and others).
- In this chapter, we examined some connections between Turing's undecidable problems, Gödel's incompleteness theorem, and Hilbert's program to formalize mathematical proof. Specifically, we saw that Turing's ideas

led to simple proofs of Gödel's ideas. The undecidability of the halting problem leads directly to the existence of true but unprovable mathematical statements!

## EXERCISES

**16.1** Which BinAd inference rules can be applied to the BinAd statement “ $1+110+1=1000$ ”?

**16.2** How many new BinAd statements can be produced by applying exactly one rule to the statement “ $1+1+1=1+10$ ”? How about any sequence of two rules? (Suggestion: To assist your understanding, calculate answers manually first, then check the answers using Python from `binAd.py`.)

**16.3** How many well-formed BinAd statements of length 4 exist?

**16.4** Give a mechanical proof for “ $10+1=11$ ” in BinAd.

**16.5** Give examples of statements that are (a) true but unprovable, and (b) provable but false. You may use the logical systems described in this chapter, but invent your own examples of statements—don’t copy specific statements mentioned in the text.

**16.6** Repeat the previous exercise, but this time invent your own logical systems to provide the examples.

**16.7** For each of the following statements, determine whether the statement’s truth value is true, false, or undefined:

- (a) In BinAd, “ $10+1=11$ ”
- (b) In BinAdLogic, “ $101+1=10+100$ ”
- (c) In BrokenBinAdLogic, “ $100=10+10$ ”
- (d) In FixedBinAdLogic, “ $100=10+10+10$ ”
- (e) In Peano arithmetic, assuming this statement is suitably encoded into ASCII, “for all integers  $x$ , there exists an integer  $y$  such that  $x - y^2 < 0$ ”
- (f) In Peano arithmetic, assuming this statement is suitably encoded into ASCII, “ $x^2 = 16$ ”

**16.8** Determine the number of elements in the following sets. Feel free to use the functions in `binAd.py` and to write additional code where necessary.

- (a) Strings of length 6 on the BinAd alphabet
- (b) Well-formed BinAd strings of length 6
- (c) Provable BinAd strings of length 6
- (d) True BinAdLogic strings of length 6

**16.9** Prove that BinAdLogic is consistent, using the hint provided on page 338.

**16.10** Prove that BinAdLogic is complete, using the following steps:

- (a) Let  $n$  be an odd positive integer, written in binary using  $k$  bits as  $b_k b_{k-1} \dots b_2 1$ . Let  $N$  be the pure regexp  $1(0|1)^*$  defined on page 333. Let  $S$  be a well-formed BinAdLogic statement, and suppose  $S$  contains

a substring  $n$  as a maximal  $N$ -match. Show that  $S$  can be obtained by applying an inference rule to another statement  $S'$ , defined as follows:  $S'$  is the same as  $S$  except with  $n$  replaced by “ $1+b_kb_{k-1}\dots b_20$ ”. The key point here is that any odd integer  $n$  can be derived by adding 1 to the strictly smaller integer  $n - 1$ .

- (b) Prove a similar result for even  $n$ . Specifically, you need to show that the occurrence of any even positive integer in a well-formed statement  $S$  can be derived from another well-formed statement  $S'$  involving strictly smaller integers.
- (c) Combine the previous two results recursively to conclude that the occurrence of any integer  $n$  in a well-formed BinAdLogic statement can be derived from the same statement with  $n$  replaced by  $1+1+\dots+1$ . (Here, there are  $n$  copies of 1).
- (d) Show that any statement of the form  $1+1+\dots+1=1+1+\dots+1$ , where there is the same number of 1s on each side of the equal sign, is provable.
- (e) Conclude that any true BinAdLogic statement  $S$  can be proved by first proving a statement of the form  $1+1+\dots+1=1+1+\dots+1$ , and then coalescing groups of 1s to produce each of the integers appearing in  $S$ .

**16.11** Using `binAd.py` as a starting point, implement one or more of the systems `BrokenBinAdLogic`, `RestrictedBinAdLogic`, and `FixedBinAdLogic`. For the systems that are incomplete or inconsistent, provide concrete Python commands that demonstrate the incompleteness or inconsistency.

**16.12** Write a program `provableInBinAd.py`, using the same approach as `provableInPeano.py` in figure 16.5. (Hint: The function `isProof()`, from `binAd.py` in the book materials, will be useful.) Describe the main differences between this approach and the approach used in the function `isProvable()` in `binAd.py`. Both approaches require exponential time, but which is likely to be more efficient, and why?

**16.13** Using arguments similar to claim 16.1 (page 342), explain how the following statement can be translated into Peano arithmetic: “program  $P$  has polynomial running time.”

**16.14** Using arguments similar to claim 16.1 (page 342), explain how the following statement can be translated into Peano arithmetic: “program  $P$  outputs yes on input  $I$ .” (Thus, you have shown that `YESONSTRING` can be reformulated as a statement about integers and arithmetic.)

**16.15** Write a new proof of our nonconstructive version of Gödel’s first incompleteness theorem (claim 16.5, page 346), using `YESONSTRING` instead of `HALTSONEMPTY`. (Hint: Use your solution to the previous exercise.)

**16.16** Our proofs of Gödel’s theorem relied heavily on explicit Python programs. This exercise and the following one will help you to reformulate the proof in terms of Turing machines described in English prose. First, we define a Turing machine  $M$  analogous to `godel.py` in figure 16.8. Machine  $M$  executes the following algorithm: “if you can prove  $M$  doesn’t halt, then halt; else loop.” Next, we define a Peano arithmetic sentence  $S$  analogous to `notHaltInPeano` in `godel.py`. Let  $S = “M \text{ doesn’t halt}.”$  Show that  $S$  is a true but unprovable sentence.

**16.17** This exercise asks you to provide yet another proof of Gödel's theorem. Essentially, you will be repeating exercise 16.15 using Turing machines instead of Python programs. Let  $M$  be the Turing machine that executes the following algorithm: “if you can prove  $M$  doesn't accept, then accept; else reject.” Let  $S$  be the Peano arithmetic sentence “ $M$  doesn't accept.” Show that  $S$  is true but unprovable.

**16.18** Gödel's original proof of the first incompleteness theorem was based on the deceptively simple sentence  $S = “S$  is not provable.” The difficult part of the proof is showing that  $S$  can be converted into an equivalent sentence of Peano arithmetic, but we omit that here. Instead, *assume* that  $S$  has been expressed in Peano arithmetic. Show that  $S$  is true but unprovable.

**16.19** Gödel's first incompleteness theorem is sometimes used as evidence for the claim that human minds are intrinsically more powerful than computers. The essence of this argument is that there exist certain statements in Peano arithmetic that humans can prove are true, but that computers cannot prove are true. (a) Without doing any background reading, form your own opinion about this claim and state your reasons. (b) Consult Alan Turing's rebuttal of the claim his 1950 paper “Computing machinery and intelligence.” The discussion is in section 6, under the heading “(3) The Mathematical Objection.” Summarize Turing's argument, state whether or not you agree with it, and explain your reasoning.

# 17



## KARP'S 21 PROBLEMS

In 1972, Richard Karp published a paper that transformed the landscape of theoretical computer science. At that time, the idea of NP-completeness was already known to Karp: Stephen Cook had proved that SAT is NP-complete a year earlier. (Leonid Levin had also proved and circulated this result in Russia; but Karp, based in the USA, did not learn of Levin's work until later.) So SAT was known to be a "hardest" problem in NP. Based on this, one might think that SAT is an unusually rare type of hard problem, sitting on top of many other easier NP problems. But Karp's paper, entitled "Reducibility among combinatorial problems," showed that in fact NP-completeness isn't rare. Karp explicitly proved that 21 different problems, drawn from a surprising variety of applications in math and computer science, are NP-complete. Thus, he demonstrated that many of the hard problems studied by computer scientists are in some sense equally hard—and that none of them has an efficient method of solution unless P=NP.

The bulk of this chapter is devoted to excerpts from Karp's paper, presented in shaded boxes. All of Karp's original words, symbols, and notation have been retained, but some of the fonts have been altered to resemble modern mathematical typesetting. Occasional explanatory comments are inserted in square brackets [*like this*].

### 17.1 KARP'S OVERVIEW

We begin with Karp's abstract and introduction, which provide an excellent overview of the whole paper.

#### REDUCIBILITY AMONG COMBINATORIAL PROBLEMS<sup>†</sup>

RICHARD M. KARP

UNIVERSITY OF CALIFORNIA AT BERKELEY

*Abstract:* A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple

encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bound algorithm or none of them does.

<sup>†</sup>This research was partially supported by the National Science Foundation Grant GJ-474.

## 1. INTRODUCTION

All the general methods presently known for computing the chromatic number of a graph, deciding whether a graph has a Hamilton circuit, or solving a system of linear inequalities in which the variables are constrained to be 0 or 1, require a combinatorial search for which the worst case time requirement grows exponentially with the length of the input. In this paper we give theorems which strongly suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually.

We are specifically interested in the existence of algorithms that are guaranteed to terminate in a number of steps bounded by a polynomial in the length of the input. We exhibit a class of well-known combinatorial problems, including those mentioned above, which are equivalent, in the sense that a polynomial-bounded algorithm for any one of them would effectively yield a polynomial-bounded algorithm for all. We also show that, if these problems do possess polynomial-bounded algorithms then all the problems in an unexpectedly wide class (roughly speaking, the class of problems solvable by polynomial-depth backtrack search) possess polynomial-bounded algorithms.

Following is a brief summary of the contents of the paper. For the sake of definiteness our technical development is carried out in terms of the recognition of languages by one-tape Turing machines, but any of a wide variety of other abstract models of computation would yield the same theory. Let  $\Sigma^*$  be the set of all finite strings of 0's and 1's. A subset of  $\Sigma^*$  is called a *language*. Let  $P$  be the class of languages recognizable in polynomial time by one-tape deterministic Turing machines, and let  $NP$  be the class of languages recognizable in polynomial time by one-tape nondeterministic Turing machines. Let  $\Pi$  be the class of functions from  $\Sigma^*$  into  $\Sigma^*$  computable in polynomial time by one-tape Turing machines. Let  $L$  and  $M$  be languages. We say that  $L \propto M$  ( $L$  is *reducible* to  $M$ ) if there is a function  $f \in \Pi$  such that  $f(x) \in M \Leftrightarrow x \in L$ . If  $M \in P$  and  $L \propto M$  then  $L \in P$ . We call  $L$  and  $M$  equivalent if  $L \propto M$  and  $M \propto L$ . Call  $L$  (*polynomial*) *complete* if  $L \in NP$  and every language in  $NP$  is reducible to  $L$ . Either all complete languages are in  $P$ , or none of them are. The former alternative holds if and only if  $P=NP$ .

The main contribution of this paper is the demonstration that a large number of classic difficult computational problems, arising in fields such as mathematical programming, graph theory, combinatorics, computational logic and switching

Karp's word or phrase	Equivalent in this book
reduction	polyreduction (or “polynomial-time mapping reduction”)
complete (or “polynomial complete”)	NP-complete
$\propto$	$\leq_P$
digraph	directed graph
arc	edge (in a graph)
KNAPSACK	SUBSETSUM
$\bar{x}$	$\neg x$
$x_1 \cup x_2$	$x_1 \vee x_2$
$(x_1 \cup x_2)(x_3)$	$(x_1 \vee x_2) \wedge (x_3)$

Figure 17.1: Karp's terminology and notation compared to this book.

theory, are complete (and hence equivalent) when expressed in a natural way as language recognition problems.

This paper was stimulated by the work of Stephen Cook (1971), and rests on an important theorem which appears in his paper. The author also wishes to acknowledge the substantial contributions of Eugene Lawler and Robert Tarjan.

This elegant and succinct overview needs little additional explanation. The main thing to watch out for is differences in terminology and notation, summarized in figure 17.1. Some of these differences have already emerged in the excerpt above, but others won't arise until later. Perhaps the most important is the use of the symbol  $\propto$  for polyreductions. The symbols used in Boolean formulas are also rather different, and should be noted carefully. For example, Karp's  $(\bar{x}_1 \cup x_2)(x_3)$  becomes our  $(\neg x_1 \vee x_2) \wedge (x_3)$ .

Karp is careful to credit Stephen Cook with the discovery of NP-completeness in the final paragraph of the excerpt above. But as already mentioned, the Russian mathematician Leonid Levin independently developed many of Cook's and Karp's ideas in the early 1970s.

## 17.2 KARP'S DEFINITION OF NP-COMPLETENESS

Now we skip over some technical definitions and rejoin the paper for the last portion of section 3, which discusses the previously known relationship between NP and SAT, and makes some important remarks on the P versus NP question.

This excerpt also introduces us to Karp's notation for defining computational problems. In an earlier section that we skipped, Karp explains that you can use any reasonable encoding of mathematical objects such graphs and Boolean formulas. So his problem definitions use the mathematical objects themselves.

Karp considers only decision problems. The standard format is to present the “input” to the problem, followed by a “property.” By definition, positive instances possess the given property and negative instances do not. For example, in defining SAT below, the input is a list of clauses, and the property is “the conjunction of the given clauses is satisfiable.”

### 3. NONDETERMINISTIC ALGORITHMS AND COOK'S THEOREM

...

The class NP is very extensive. Loosely, a recognition problem is in NP if and only if it can be solved by a backtrack search of polynomial bounded depth. A wide range of important computational problems which are not known to be in P are obviously in NP. For example, consider the problem of determining whether the nodes of a graph  $G$  can be colored with  $k$  colors so that no two adjacent nodes have the same color. The nondeterministic algorithm can simply guess an assignment of colors to the nodes and then check (in polynomial time) whether all pairs of adjacent nodes have distinct colors.

In view of the wide extent of NP, the following theorem due to Cook is remarkable. We define the satisfiability problem as follows:

SATISFIABILITY

**Input:** Clauses  $C_1, C_2, \dots, C_p$

**Property:** The conjunction of the given clauses is satisfiable; i.e., there is a set  $S \subseteq \{x_1, x_2, \dots, x_n; \bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$  such that

- (a)  $S$  does not contain a complementary pair of literals and
- (b)  $S \cap C_k \neq \emptyset, k = 1, 2, \dots, p.$

It's worth pausing here to verify that Karp's definition of SAT is the same as ours, despite the differences in notation. For Karp, each clause  $C_k$  is a *set* of literals which are implicitly OR'ed together. For example, we might have  $C_3 = \{x_2, \bar{x}_5, x_8\}$ . In our own definitions, we write out clauses explicitly, as in  $C_3 = x_2 \vee \neg x_5 \vee x_8$ . Karp's set  $S$  is the set of literals that are true in a satisfying assignment. Condition (a) states that a literal and its negation cannot both be true; condition (b) states that at least 1 literal in each clause must be true. A little thought shows that this is the same as our notion of a satisfying assignment.

In the next excerpt, Karp states Cook's theorem (Theorem 2, these days usually known as the Cook–Levin theorem) and its obvious consequence (Corollary 1) for the P versus NP question.

*Theorem 2 (Cook).* If  $L \in \text{NP}$  then  $L \propto \text{SATISFIABILITY}$ .

The theorem stated by Cook (1971) uses a weaker notion of reducibility than the one used here, but Cook's proof supports the present statement.

*Corollary 1.*  $\text{P} = \text{NP} \Leftrightarrow \text{SATISFIABILITY} \in \text{P}$ .

...

Theorem 2 shows that, if there were a polynomial-time algorithm to decide membership in SATISFIABILITY then every problem solvable by a polynomial-depth

backtrack search would also be solvable by a polynomial-time algorithm. This is strong circumstantial evidence that SATISFIABILITY  $\notin P$ .

Section 4 continues with Karp's definition of NP-completeness.

#### 4. COMPLETE PROBLEMS

The main object of this paper is to establish that a large number of important computational problems can play the role of SATISFIABILITY in Cook's theorem. Such problems will be called complete.

*Definition 5.* The language  $L$  is (*polynomial*) *complete* if

- (a)  $L \in NP$  and
- (b)  $SATISFIABILITY \propto L$ .

This concept of “polynomial complete” or just “complete” is what we today call “NP-complete.” Karp immediately states the most important property of NP-complete problems (or, equivalently, NP-complete languages).

*Theorem 3.* Either all complete languages are in  $P$ , or none of them are. The former alternative holds if and only if  $P=NP$ .

This theorem follows directly from the definition of NP-completeness, and Karp doesn't give a proof.

### 17.3 THE LIST OF 21 NP-COMPLETE PROBLEMS

After a technical elaboration that we skip, Karp proceeds with the heart of the paper: a proof that 21 problems drawn from diverse areas of mathematics and computer science are in fact NP-complete.

The rest of the paper is mainly devoted to the proof of the following theorem.

*Main Theorem.* All the problems on the following list are complete.

1. SATISFIABILITY [*This has already been defined above (on our page 356), so Karp does not repeat the definition.*] ....
2. 0–1 INTEGER PROGRAMMING.  
**Input:** integer matrix  $C$  and integer vector  $d$   
**Property:** There exists a 0–1 vector  $x$  such that  $Cx = d$ .
3. CLIQUE  
**Input:** graph  $G$ , positive integer  $k$   
**Property:**  $G$  has a set of  $k$  mutually adjacent nodes.
4. SET PACKING  
**Input:** Family of sets  $\{S_j\}$ , positive integer  $l$   
**Property:**  $\{S_j\}$  contains  $l$  mutually disjoint sets.

## 5. NODE COVER

**Input:** graph  $G'$ , positive integer  $l$

**Property:** There is a set  $R \subseteq N'$  such that  $|R| \leq l$  and every arc is incident with some node in  $R$ . [ $N'$  is the set of nodes in the graph  $G'$ .]

## 6. SET COVERING

**Input:** finite family of sets  $\{S_j\}$ , positive integer  $k$

**Property:** There is a subfamily  $\{T_b\} \subseteq \{S_j\}$  containing  $\leq k$  sets such that  $\bigcup T_b = \bigcup S_j$ . [Karp's appendix clarifies that  $\{S_j\}$  and  $\{T_b\}$  are finite families of finite sets.]

## 7. FEEDBACK NODE SET

**Input:** digraph  $H$ , positive integer  $k$

**Property:** There is a set  $R \subseteq V$  such that every (directed) cycle of  $H$  contains a node in  $R$ . [ $V$  is the set of nodes in the directed graph  $H$ .]

## 8. FEEDBACK ARC SET

**Input:** digraph  $H$ , positive integer  $k$

**Property:** There is a set  $S \subseteq E$  such that every (directed) cycle of  $H$  contains an arc in  $S$ . [ $E$  is the set of edges in the directed graph  $H$ .]

## 9. DIRECTED HAMILTON CIRCUIT

**Input:** digraph  $H$

**Property:**  $H$  has a directed cycle which includes each node exactly once.

## 10. UNDIRECTED HAMILTON CIRCUIT

**Input:** graph  $G$

**Property:**  $G$  has a cycle which includes each node exactly once.

## 11. SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE

**Input:** Clauses  $D_1, D_2, \dots, D_r$ , each consisting of at most 3 literals from the set  $\{u_1, u_2, \dots, u_m\} \cup \{\bar{u}_1, \bar{u}_2, \dots, \bar{u}_m\}$

**Property:** The set  $\{D_1, D_2, \dots, D_r\}$  is satisfiable.

## 12. CHROMATIC NUMBER

**Input:** graph  $G$ , positive integer  $k$

**Property:** There is a function  $\phi : N \rightarrow Z_k$  such that, if  $u$  and  $v$  are adjacent, then  $\phi(u) \neq \phi(v)$ . [ $N$  is the set of nodes.  $Z_k$  is the set of  $k$  integers from 0 to  $k - 1$ .]

## 13. CLIQUE COVER

**Input:** graph  $G'$ , positive integer  $l$

**Property:**  $N'$  is the union of  $l$  or fewer cliques.

## 14. EXACT COVER

**Input:** family  $\{S_j\}$  of subsets of a set  $\{u_i, i = 1, 2, \dots, t\}$

**Property:** There is a subfamily  $\{T_b\} \subseteq \{S_j\}$  such that the sets  $T_b$  are disjoint and  $\bigcup T_b = \bigcup S_j = \{u_i, i = 1, 2, \dots, t\}$ .

## 15. HITTING SET

**Input:** family  $\{U_j\}$  of subsets of a set  $\{s_j, j = 1, 2, \dots, r\}$

**Property:** There is a set  $W$  such that, for each  $i$ ,  $|W \cap U_i| = 1$ .

## 16. STEINER TREE

**Input:** graph  $G$ ,  $R \subseteq N$ , weighting function  $w : A \rightarrow Z$ , positive integer  $k$  [Here,  $A$  is the set of arcs (i.e., edges) and  $Z$  is the set of integers.]

**Property:**  $G$  has a subtree of weight  $\leq k$  containing the set of nodes in  $R$ .

## 17. 3-DIMENSIONAL MATCHING

**Input:** set  $U \subseteq T \times T \times T$ , where  $T$  is a finite set

**Property:** There is a set  $W \subseteq U$  such that  $|W| = |T|$  and no two elements of  $W$  agree in any coordinate.

## 18. KNAPSACK

**Input:**  $(a_1, a_2, \dots, a_r, b) \in Z^{r+1}$  [This is a typo; should be  $Z^{r+1}$ .]

**Property:**  $\sum a_j x_j = b$  has a 0–1 solution.

## 19. JOB SEQUENCING

**Input:** “execution time vector”  $(T_1, \dots, T_p) \in Z^p$ ,

“deadline vector”  $(D_1, \dots, D_p) \in Z^p$

“penalty vector”  $(P_1, \dots, P_p) \in Z^p$

positive integer  $k$

**Property:** There is a permutation  $\pi$  of  $\{1, 2, \dots, p\}$  such that

$$\left( \sum_{j=1}^p \left[ \text{if } T_{\pi(1)} + \dots + T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0 \right] \right) \leq k.$$

## 20. PARTITION

**Input:**  $(c_1, c_2, \dots, c_s) \in Z^s$

**Property:** There is a set  $I \subseteq \{1, 2, \dots, s\}$  such that  $\sum_{b \in I} c_b = \sum_{b \notin I} c_b$ .

## 21. MAX CUT

**Input:** graph  $G$ , weighting function  $w : A \rightarrow Z$ , positive integer  $W$  [Again,  $A$  is the set of arcs (i.e., edges) and  $Z$  is the set of integers. The  $N$  below is  $G$ 's set of nodes.]

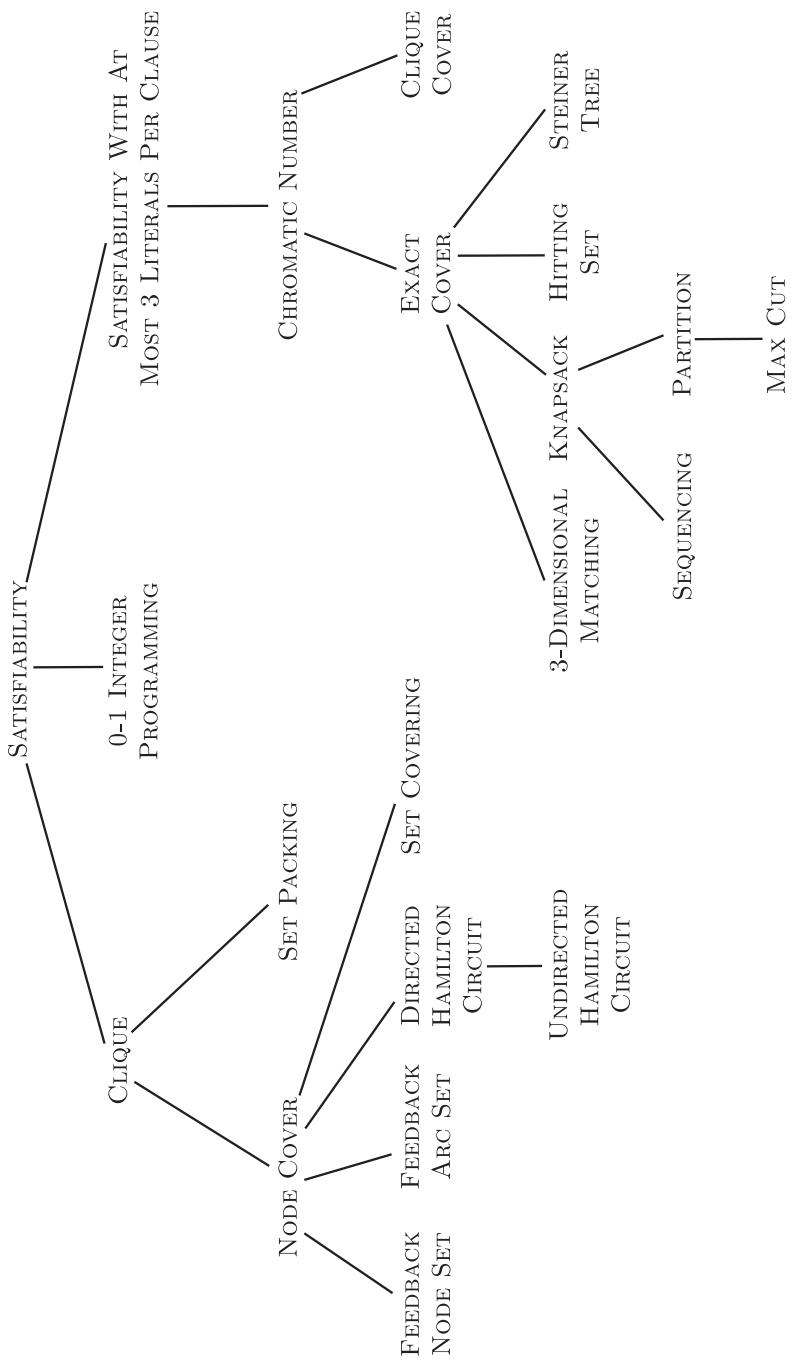
**Property:** There is a set  $S \subseteq N$  such that

$$\sum_{\substack{\{u, v\} \in A \\ u \in S \\ v \notin S}} w(\{u, v\}) \geq W.$$

## 17.4 REDUCTIONS BETWEEN THE 21 NP-COMPLETE PROBLEMS

The previous few pages were just the *statement* of the main theorem. Now Karp plows into the proof of the theorem. To do this, he provides 20 polyreductions between his 21 problems—one reduction for each of the lines in figure 17.2.

It is clear that these problems (or, more precisely, their encodings into  $\Sigma^*$ ), are all in NP. We proceed to give a series of explicit reductions, showing that SATISFIABILITY is reducible to each of the problems listed. Figure 1 [our figure 17.2, page 360] shows the structure of the set of reductions. Each line in the figure indicates a reduction of the upper problem to the lower one.



**Figure 17.2:** Karp's figure 1, captioned "Complete Problems."

To exhibit a reduction of a set  $T \subseteq D$  to a set  $T' \subseteq D'$  we specify a function  $F: D \rightarrow D'$  which satisfies the conditions of Lemma 2. In each case, the reader should have little difficulty in verifying that  $F$  does satisfy these conditions.

We skipped the “Lemma 2” referred to here, but the conditions on the reduction function  $F$  are easy enough to understand:  $F$  must map positive instances to positive instances, negative instances to negative instances, and it must be computable in polynomial time.

As we will see, Karp’s statement that “the reader should have little difficulty in verifying” the reductions is true only if the reader has a lot of expertise in algorithms. For us, there will be a substantial amount of work in understanding some of the reductions. And of the 20 reductions presented, we will examine only 5 of the easier ones. Specifically, we look at  $\text{SAT} \leq_p \text{CLIQUE}$ ,  $\text{CLIQUE} \leq_p \text{NODE COVER}$ ,  $\text{DHC} \leq_p \text{UHC}$ ,  $\text{SAT} \leq_p \text{3-SAT}$ , and  $\text{KNAPSACK} \leq_p \text{PARTITION}$ .

## Polyreducing SAT to CLIQUE

Karp’s brutally concise description of the polyreduction  $\text{SAT} \leq_p \text{CLIQUE}$  is completely described in our next excerpt.

SATISFIABILITY  $\propto$  CLIQUE

$$N = \{\langle \sigma, i \rangle \mid \sigma \text{ is a literal and occurs in } C_i\}$$

$$A = \{\{\langle \sigma, i \rangle, \langle \delta, j \rangle\} \mid i \neq j \text{ and } \sigma \neq \bar{\delta}\}$$

$$k = p, \text{ the number of clauses.}$$

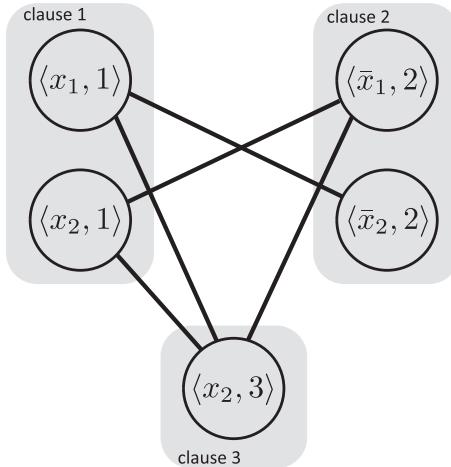
Let’s first remind ourselves of the basic objective here. We are given an instance of SAT, specified by  $p$  clauses  $C_1, C_2, \dots, C_p$  containing various combinations of the literals  $x_1, x_2, \dots, x_n, \bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ . We need to transform this instance into (i) a graph  $G$ , consisting of a set of nodes  $N$  and a set of edges (“arcs”)  $A$ , and (ii) an integer  $k$ . We need to construct  $G$  so that it has a clique of size  $k$  if and only if the clauses  $C_i$  are satisfiable. (A *clique* of size  $k$  is a set of  $k$  nodes in which every node is connected directly to every other node by an edge.)

As a running example to aid our understanding, let’s consider the instance

$$C_1 = \{x_1, x_2\},$$

$$C_2 = \{\bar{x}_1, \bar{x}_2\},$$

$$C_3 = \{x_2\}.$$



**Figure 17.3: An illustration of Karp's poly reduction from SAT to CLIQUE.** This particular example is for the SAT instance  $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2)$ .

In our usual notation, this would be written

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2).$$

Karp's approach creates a node of  $G$  for every occurrence of every literal in every clause (that's the line starting “ $N =$ ” in the excerpt above). In our running example, that gives us 5 nodes, written in Karp's notation as  $\langle x_1, 1 \rangle$ ,  $\langle x_2, 1 \rangle$ ,  $\langle \bar{x}_1, 2 \rangle$ ,  $\langle \bar{x}_2, 2 \rangle$ ,  $\langle x_2, 3 \rangle$ . You can visualize these nodes in figure 17.3.

Next (in the line starting “ $A =$ ”), Karp tells us to draw an edge between every pair of nodes, except for (i) pairs of nodes in the same clause (“ $i \neq j$ ”) and (ii) pairs whose literals are negations of each other (“ $\sigma \neq \bar{\delta}$ ”). These negated pairs, such as  $x_1$  and  $\bar{x}_1$ , are known as *complementary literals*. We see an example in figure 17.3, where the nodes  $\langle x_1, 1 \rangle$  and  $\langle \bar{x}_1, 2 \rangle$  are not joined by an edge, because they contain complementary literals.

Finally, in the line “ $k = p$ ,” Karp tells us to look for a clique whose size  $k$  is the same as the number of clauses  $p$ . Our running example has 3 clauses, so we would look for a clique of size 3 in figure 17.3.

The transformation does run in polynomial time, as we can see from the following informal reasoning. The number of nodes in  $G$  is the same as the number of literals in the SAT instance, which is in  $O(n)$ . The number of edges is bounded by the number of pairs of nodes, which is in  $O(n^2)$ . And deciding exactly which edges to include can be done by efficient lookups—certainly no more than  $O(n)$  for each potential edge.

Finally, we need to check that (a) a satisfiable instance produces a graph with a  $k$ -clique, and (b) an unsatisfiable instance produces a graph with no  $k$ -clique. The example in figure 17.3 will help to understand this. The basic idea is that a satisfying assignment leads to a clique where we pick one node from each gray rectangle (the “clauses”). In each rectangle, we choose the node that is true in the satisfying assignment. For example, our SAT instance is satisfied by  $x_1 = 0, x_2 = 1$ ,

so we pick the nodes  $\langle x_2, 1 \rangle$ ,  $\langle \bar{x}_1, 2 \rangle$ , and  $\langle x_2, 3 \rangle$  for clauses 1, 2, and 3 respectively. Note that these nodes do indeed form a 3-clique in figure 17.3.

That was a specific example. Let's now give a formal proof. For (a), we observe that every chosen node has an edge to every other chosen node because they are all taken from different clauses, and they can't contain complementary literals—because a satisfying assignment cannot contain both  $x_i$  and  $\bar{x}_i$ . So the  $k$  selected nodes do indeed form a  $k$ -clique.

For (b), assume we have an unsatisfiable instance that nevertheless yields a graph with a  $k$ -clique; we'll argue for a contradiction. Note that the  $k$ -clique would need to have exactly one node taken from each clause (i.e., gray rectangle), because nodes in the same clause never have edges between them. And a  $k$ -clique with one node from each clause cannot contain any complementary pairs (because complementary pairs are not joined by edges either). But this would immediately give us a satisfying assignment, by setting each of the chosen positive literals to 1 and negative literals to 0. This contradicts the assumption that the SAT instance is unsatisfiable.

You can get a more intuitive feeling for this polyreduction by experimenting with the provided Python implementation, `convertSatToClique.py`. For example, to see the concrete implementation of figure 17.3, use the command

```
>>> convertSatToClique('(x1 OR x2) AND (NOT x1 OR NOT x2) AND (x2)')
```

## Polyreducing CLIQUE to NODE COVER

Karp dispenses with this polyreduction in two lines:

**CLIQUE  $\propto$  NODE COVER**

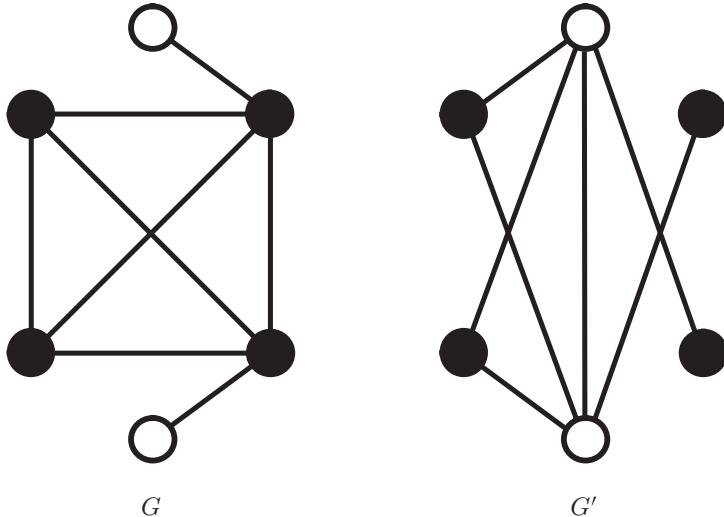
$G'$  is the complement of  $G$ .

$$l = |N| - k$$

This polyreduction starts with a graph  $G$  and integer  $k$ , producing a new graph  $G'$  and new integer  $l$ . The requirement is that  $G$  has a  $k$ -clique if and only if the edges of  $G'$  can be covered by  $l$  nodes. Here, “covered” means every edge is attached to at least one of the chosen  $l$  nodes. Karp tells us  $G'$  should be the “complement” of  $G$ , which means that  $G'$  has an edge everywhere that  $G$  doesn't, and vice versa. And the size  $l$  of the covering set is  $|N| - k$ , where  $|N|$  is the number of nodes in  $G$ .

Figure 17.4 gives an example where we take  $k = 4$ . The  $G$  shown here on the left does in fact have a 4-clique consisting of the filled circles, so we have a positive instance of CLIQUE. Note that  $G$  has 6 nodes. The complementary graph  $G'$ , shown on the right, has the same 6 nodes. As Karp suggests, we take  $l = 6 - 4 = 2$ , and observe that  $G'$  can indeed be covered by 2 nodes (the unfilled circles). So, as expected, the positive instance CLIQUE has been converted to a positive instance of NODE COVER.

To prove that this works in general, we need to show that (a) if  $G$  has a  $k$ -clique, then  $G'$  can be covered by  $l$  nodes, and (b) if  $G'$  can be covered by



**Figure 17.4:** Polyreduction from CLIQUE instance  $G$  with  $k = 4$  (left) to NODE COVER instance  $G'$  with  $l = 6 - 4 = 2$  (right).

$l$  nodes, then  $G$  has a  $k$ -clique. For (a), we choose to cover  $G'$  with the  $l$  nodes that were not in the clique (e.g., the unfilled circles in figure 17.4). Any edge  $e$  not covered by this set must join two elements of the original clique in  $G$ , but that means  $e$  was in  $G$  and therefore cannot be in the complementary graph  $G'$ . For (b), we run this argument in reverse. Choose the set of  $k$  nodes that were not used to cover  $G'$  (e.g., the filled circles in figure 17.4). These  $k$  nodes must be a clique in  $G$ , since any missing edge between two chosen nodes would become an uncovered edge in  $G'$ .

Finally, note that it's easy to complement a graph in polynomial time (at worst, in fact, quadratic time)—so this is a polyreduction.

### Polyreducing DHC to UHC

DIRECTED HAMILTON CIRCUIT  $\propto$  UNDIRECTED HAMILTON CIRCUIT

$$\begin{aligned} N &= V \times \{0, 1, 2\} \\ A &= \{\{\langle u, 0 \rangle, \langle u, 1 \rangle\}, \{\langle u, 1 \rangle, \langle u, 2 \rangle\} \mid u \in V\} \\ &\quad \cup \{\{\langle u, 2 \rangle, \langle v, 0 \rangle\} \mid \langle u, v \rangle \in E\} \end{aligned}$$

This is exactly the polyreduction explained in chapter 13 (see figure 13.7, page 280)—and implemented in the provided program `convertDHCtoUHC.py`.

But Karp's notation is a little different. Recall that each node in the directed graph becomes a triplet of three child nodes in the undirected version. In chapter 13, a node  $u$  became the triplet  $ua, ub, uc$ ; in Karp's notation,  $u$  is instead converted to the triplet  $\{ \langle u, 0 \rangle, \langle u, 1 \rangle, \langle u, 2 \rangle \}$ . If you puzzle out the formula for the undirected edge set (the line beginning “ $A =$ ” in the excerpt above), you will find that this too is identical to the chapter 13 construction. Specifically, Karp tells us to draw an edge between the 0-child and 1-child, and between the 1-child and 2-child, for each set of triplets. And for every edge in the directed graph, we draw an edge between corresponding triplets in the undirected graph, from the 2-child to the 0-child. A formal proof that the polyreduction works correctly was given in claim 13.2 (page 279).

## Polyreducing SAT to 3-SAT

### SATISFIABILITY $\propto$ SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE

Replace a clause  $\sigma_1 \cup \sigma_2 \cup \dots \cup \sigma_m$ , where the  $\sigma_i$  are literals and  $m > 3$ , by

$$(\sigma_1 \cup \sigma_2 \cup u_1)(\sigma_3 \cup \dots \cup \sigma_m \cup \bar{u}_1)(\bar{\sigma}_3 \cup u_1) \dots (\bar{\sigma}_m \cup u_1),$$

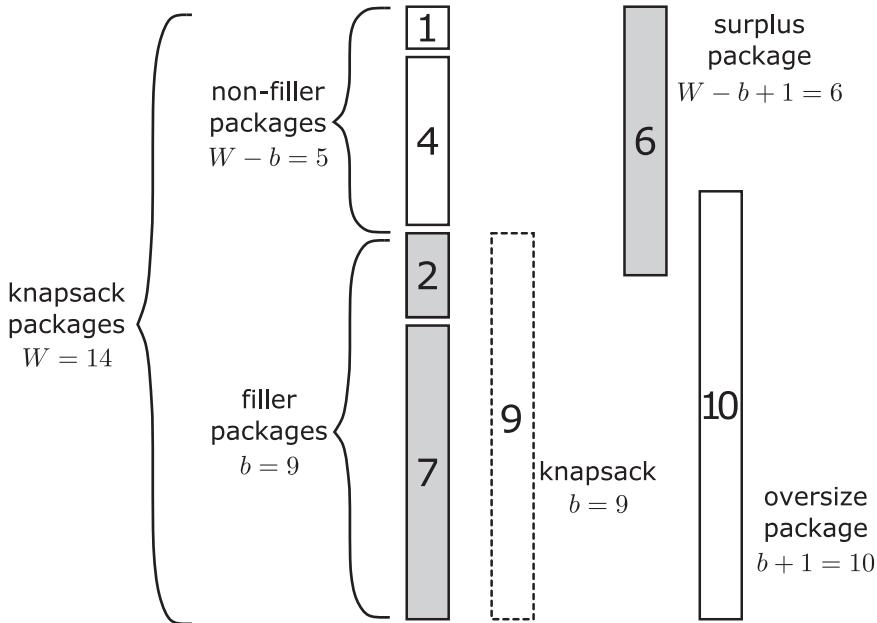
where  $u_1$  is a new variable. Repeat this transformation until no clause has more than three literals.

This polyreduction relies on the same basic trick as our own proof in claim 13.7 on page 289. Once you translate Karp's notation into our own, you'll see that the main idea is to break long clauses down into shorter clauses by introducing dummy variables. The construction Karp uses for doing this is a little more elaborate than necessary, since only the first two clauses above are really needed. But the proof that it works correctly is essentially identical to ours, so we don't discuss it further here.

## Polyreducing KNAPSACK to PARTITION

### KNAPSACK $\propto$ PARTITION

$$\begin{aligned} s &= r + 2 \\ c_i &= a_i, \quad i = 1, 2, \dots, r \\ c_{r+1} &= b + 1 \\ c_{r+2} &= \left( \sum_{i=1}^r a_i \right) + 1 - b \end{aligned}$$



**Figure 17.5: An example of Karp's reduction from KNAPSACK to PARTITION.** The knapsack instance has weights 1, 2, 4, 7 with knapsack size 9. This reduces to a partition instance with weights 1, 2, 4, 7, 6, 10. Because the knapsack instance is positive, we obtain a positive partition instance—the three shaded weights 2, 7, 6 can form one of the partitions.

Let's first understand the two problems. Karp's KNAPSACK is what we today call SUBSETSUM and was defined on page 257; in Karp's notation, KNAPSACK asks whether some subset of the  $a_i$  sums to  $b$ . (Metaphorically, can we exactly fill the “knapsack” of capacity  $b$  with some of the  $r$  “packages” whose sizes are given by the  $a_i$ ?) Karp's version of PARTITION (see also our own definition on page 257) asks whether the  $s$  “packages” with weights  $c_i$  can be split into two sets of equal weight.

In the first line of the polyreduction, we see  $s = r + 2$ , meaning that the PARTITION instance will have 2 more packages than the KNAPSACK instance. The next line says that the first  $r$  packages will be the same as for KNAPSACK. Then we add 2 new packages. First, we have  $c_{r+1}$ , whose weight is 1 more than the KNAPSACK capacity, i.e.,  $b + 1$ . Call this the *oversize package*. Second, we have  $c_{r+2}$ , whose weight is 1 more than the knapsack “surplus.” Here, the *surplus* is the total weight  $W$  of the knapsack packages, minus the knapsack capacity. Call this the *surplus package*, with weight  $W - b + 1$ . Figure 17.5 gives a specific example.

We need to show that (a) a positive KNAPSACK instance yields a positive PARTITION instance; and (b) a positive PARTITION instance must result from a positive KNAPSACK instance. For (a), we can take our positive KNAPSACK instance and relabel the packages so that the first  $k$  packages fill the knapsack, i.e.,  $\sum_{i=1}^k a_i = b$ . These are the *filler packages*. That means the remaining  $r - k$

packages have the rest of the weight, i.e.,  $\sum_{i=k+1}^r a_i = W - b$ . These are the *non-filler* packages. Let's now split our PARTITION packages into two equal sets, each weighing  $W + 1$ . In the first set we put the filler packages and the surplus package, for a total weight of  $b + (W + 1 - b) = W + 1$  as claimed (see the shaded packages in figure 17.5 for a specific example). In the other set we put the non-filler packages and the oversize package (i.e., the unshaded packages in the figure), for a total weight of  $(W - b) + (b + 1) = W + 1$ , as claimed. This completes part (a).

For (b), note that the weight of all PARTITION packages is  $W + (b + 1) + (W - b + 1) = 2W + 2$ . So in a positive instance, the weight of each partition must be  $(2W + 2)/2 = W + 1$ . Observe that the oversize and surplus packages can't be in the same partition: if they were, the weight of that partition would be at least  $(b + 1) + (W - b + 1) = W + 2$ , which is bigger than the target  $W + 1$ . Therefore, one of the partitions contains only the surplus package and some knapsack packages. The weight of the knapsack packages in this partition is  $(W + 1) - (W - b + 1) = b$ . So selecting just these packages produces a positive solution to the original KNAPSACK instance. The shaded filler packages in figure 17.5 again provide a specific example of this.

## 17.5 THE REST OF THE PAPER: NP-HARDNESS AND MORE

We won't examine any more excerpts of Karp's "Reducibility" paper, but it's worth briefly noting the remaining content. First, Karp did not explicitly define NP-hardness, but he was clearly aware of this important concept. He lists three problems that we would today call NP-hard, and he proves they are at least as hard as the NP-complete problems.

Finally, Karp is careful to admit that there may be "hard" problems in NP that are not NP-complete. He gives three examples of problems that, in 1972, were thought likely to be in this category. In modern terminology, the three problems are graph isomorphism, primality, and linear programming. It's interesting to note that these three problems have been substantially slain in the intervening years. A polynomial-time algorithm for linear programming was discovered in 1979. And as discussed on page 237, the polynomial-time AKS algorithm for primality was discovered in 2002. Graph isomorphism has also been attacked rather successfully—see page 309 for details.

## EXERCISES

### 17.1 Consider the following SAT instance:

“(x<sub>1</sub> OR x<sub>2</sub>) AND (x<sub>2</sub> OR NOT x<sub>3</sub> OR x<sub>4</sub>) AND (NOT x<sub>2</sub> OR NOT x<sub>4</sub>) AND  
(NOT x<sub>1</sub> OR NOT x<sub>2</sub> OR x<sub>4</sub>)”

- (a) Without using any assistance from a computer, apply Karp's polyreduction from SAT to CLIQUE to this instance.
- (b) Use the provided `convertSatToClique.py` to check your answer to part (a).

**17.2** Consider the following SAT instance:

```
"(x1 OR x2 OR NOT x3 OR NOT x4 OR x5) AND (NOT x1 OR NOT x2 OR x3  
OR x4) AND (x4 OR NOT x5)"
```

- (a) Without using any assistance from a computer, apply Karp's polyreduction from SAT to 3-SAT to this instance.
- (b) Again without using a computer, apply our own version of the SAT→3-SAT polyreduction to this instance. (Claim 13.7 on page 289 describes this reduction.)
- (c) Use the provided `convertSatTo3Sat.py` to check your answer to part (b).

**17.3** Implement Karp's polyreduction from CLIQUE to NODE COVER in a Python program.

**17.4** Write a Python program that implements a polyreduction from SAT to NODE COVER. (Hint: This requires very little effort if you combine `convertSatToCliques.py` with your solution to exercise 17.3.)

**17.5** Of the six versions of the P versus NP question described in chapter 14, which is most similar to Karp's Corollary 1 (on our page 356)?

**17.6** Consider Karp's definition of NP-completeness, which is Definition 5 on our page 357. Now consider the four equivalent definitions of NP-completeness given in chapter 14. Which of our definitions is most similar to Karp's definition?

**17.7** Most of Karp's reductions are not invertible, but the reduction from CLIQUE to NODE COVER is an exception. Formally define the inverse reduction from NODE COVER to CLIQUE. Apply your new reduction to the following instance of NODE COVER:  $G' = \{a, d\} \cup \{b, d\} \cup \{b, e\} \cup \{c, e\} \cup \{e, f\}$ ,  $l = 2$ .

**17.8** Suppose we have a collection of packages with the following weights: 3, 5, 5, 8, 9, 9, 12; now suppose we have a knapsack (or delivery truck) that we would like to fill with packages of total weight exactly 29.

- (a) Write this scenario down as an instance of Karp's KNAPSACK problem, using the notation  $a_i, b$ .
- (b) Apply Karp's reduction to convert your answer to part (a) into an instance of PARTITION, using Karp's  $c_i$  notation.
- (c) Write down the KNAPSACK and PARTITION instances from parts (a) and (b) using this book's notation for SUBSETSUM and PARTITION (see page 257). Give positive solutions for both instances.

**17.9** Describe a polyreduction from PACKING to PARTITION, and give a formal proof that your polyreduction has the desired properties.

**17.10** Suppose that instead of Karp's suggested reduction from KNAPSACK to PARTITION (page 365), we use

$$\begin{aligned}s &= r + 2, \\c_i &= a_i, \quad i = 1, 2, \dots, r, \\c_{r+1} &= b + 500, \\c_{r+2} &=?\end{aligned}$$

Fill in the missing definition of  $c_{r+2}$  in order for this proposed reduction to work correctly, and explain your reasoning.

**17.11** Let 5-PARTITION be a decision problem defined in the same way as PARTITION, except that we need to divide the given weights into 5 sets of equal weight. Give a polyreduction from KNAPSACK to 5-PARTITION.

# 18



## CONCLUSION: WHAT WILL BE COMPUTED?

UNIVAC at present has a well grounded mathematical education fully equivalent to that of a college sophomore, and it does not forget and does not make mistakes. It is hoped that its undergraduate course will be completed shortly and it will be accepted as a candidate for a graduate degree.

— Grace Murray Hopper, *The Education of a Computer* (1952)

In the quotation above, the formidable computer pioneer Grace Hopper suggests that 1950s computers could be given a metaphorical “education” equivalent to a graduate degree in mathematics. Her dream has been realized and surpassed. Six decades later, at a 2012 event celebrating the centenary of Alan Turing’s birth, the Cambridge mathematician and Fields medalist Timothy Gowers gave a talk entitled “Will Computers Ever Become Mathematical Researchers?” Gowers believed that the answer, broadly speaking, is yes. In an age where artificial intelligence pervades many aspects of our lives, this is just one example of the extraordinary and expanding capabilities of real computers to solve real problems in the real world. The march of progress in AI also illuminates the importance of the fundamental question addressed by this book: What can be computed? Formal answers to this question, in terms of computability and complexity theory, lead to a deeper understanding of the fundamental capabilities *and* limitations of computation.

### 18.1 THE BIG IDEAS ABOUT WHAT CAN BE COMPUTED

To achieve this deeper understanding, what are the most important ideas that we should take away from this book? Here’s my personal list of the “big ideas” about what can be computed:

1. **Uncomputable problems exist** (i.e., some programs are impossible).
2. **Universal computers and programs exist.**

3. **All reasonable computational models can solve the same set of problems.** The set of “reasonable” models includes Turing machines, Python programs, Java programs, quantum computers and algorithms, multicore computers, nondeterministic algorithms, and all physical systems described by current theories of physics. This statement is one version of the Church–Turing thesis, which can also be regarded as a physical law. More expansive versions of the Church–Turing thesis place human brains and all other forms of consciousness into the same class of computational models as Turing machines.
4. **Reductions let us prove that many uncomputable problems exist.** And in fact, Rice’s theorem yields infinite families of uncomputable problems.
5. **Finite automata cannot solve the same set of problems as Turing machines.**
6. **To determine whether an algorithm is practical, we need to understand its absolute running time and its asymptotic running time.**
7. **The single most important property of a computational problem is, can the problem be solved in polynomial time?**
8. **If a problem’s solutions can be efficiently verified, they can also be efficiently computed using nondeterminism, and vice versa.** In other words, PolyCheck=NPoly. But we must remember that the “nondeterminism” here is the physically unrealistic model of unlimited simultaneously executing processors.
9. **The famous P versus NP question is equivalent to asking the following question:** If solutions can be verified efficiently, can they be computed efficiently? And when we say “computed efficiently,” we mean that the computation must employ a physically realistic model. Unlimited nondeterminism is ruled out.
10. **It’s widely believed that  $P \neq NP$ —so computing solutions is strictly harder than verifying them, in general.**
11. **The NP-hard problems form a large and important class of computational problems.** Every NP-hard problem is at least as hard as any other problem whose solutions can be efficiently verified.
12. **Most naturally arising computational problems are either NP-hard or have a polynomial-time solution.** But the rare exceptions to this rule of thumb are important. Modern cryptography is based on a handful of “intermediate” problems—problems for which no polytime algorithm is known, but which also don’t seem to be NP-hard.
13. **Thousands of real-world computational problems—drawn from areas throughout mathematics, physics, engineering, and the other sciences—are NP-hard.** Therefore, it is not possible to solve all large instances of these problems in a reasonable amount of time, unless  $P=NP$ .

Computation is a subtle but fundamental aspect of our universe. So it is my hope that the ideas above can serve as a guide, pointing the way towards a deeper understanding of the nature of computation.



## BIBLIOGRAPHY



- Agrawal, M., Kayal, N., and Saxena, N. (2004). PRIMES is in P. *Annals of Mathematics*, 160(2):781–793.
- Aiken, H. (1956). The future of automatic computing machinery. In Walther, A., Wosnik, J., and Hoffmann, W., editors, *Elektronische Rechenmaschinen und Informationsverarbeitung [Electronic Mechanical Calculators and Information Processing]*, volume 4 of *Nachrichtentechnische Fachberichte*, pages 32–34. Vieweg. Year of publication is reported differently in various sources, but the text is a transcript of a lecture delivered in 1955.
- Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press.
- Austen, J. (1818). *Persuasion*. John Murray.
- Austrin, P., Benabbas, S., and Georgiou, K. (2013). Better balance by being biased: a 0.8776-approximation for max bisection. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 277–294. SIAM.
- Babai, L. (2015). Graph isomorphism in quasipolynomial time. arXiv:1512.03547.
- Babbage, C. (1864). *Passages from the Life of a Philosopher*. Longman. Reprinted 1994 by Rutgers University Press, edited by Martin Campbell-Kelly.
- Child, J., Beck, S., and Bertholle, L. (1961). *Mastering the Art of French Cooking*. Knopf.
- Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363.
- Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the ACM Symposium on Theory of Computing*, STOC ’71, pages 151–158.
- Descartes, R. (1998). *Discourse on Method and Meditations on First Philosophy*. Hackett, 4th edition. Translated by Donald Cress.
- Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3): 449–467.
- Eliot, T. S. (1922). *The Waste Land*. Horace Liveright.
- Feynman, R. (1967). *The Character of Physical Law*. MIT Press.
- Fortnow, L. (2013). *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press.
- Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman.
- Glover, D. (1953). *Arawata Bill: A Sequence of Poems*. Pegasus Press.
- Goldreich, O. (2010). *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press.
- Hartmanis, J. and Stearns, R. E. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306.
- Hilbert, D. (2000). Mathematical problems. *Bulletin of the American Mathematical Society*, 37(4):407–436. Originally published 1900, translated by M. Newson.
- Hobbes, T. (1656). *Elements of Philosophy*. R. and W. Leybourn, London.

- Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York.
- Hopper, G. M. (1952). The education of a computer. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*, ACM '52, pages 243–249. ACM.
- Jobs, S. and Beahm, G. W. (2011). *I, Steve: Steve Jobs in His Own Words*. Agate.
- Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Proceedings of the Symposium on Complexity of Computer Computations*, pages 85–103. Plenum.
- Kleene, S. (1936). General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742.
- Knuth, D. E. (1974). Computer programming as an art. *Communications of the ACM*, 17(12):667–673.
- Ladner, R. E., Lynch, N. A., and Selman, A. L. (1975). A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–123.
- Levin, L. (1973). Universal search problems. *Problemy Peredachi Informatsii*, 9(3):115–116. English translation available in *Annals of the History of Computing* 6(4):399–400, 1984.
- Linz, P. (2011). *An Introduction to Formal Languages and Automata*. Jones & Bartlett, 5th edition.
- Lovelace, A. (1843). Sketch of the analytical engine invented by Charles Babbage. *Taylor's Scientific Memoirs*, 3:666–731. Article XXIX. Translation and notes by Lovelace. Original article by L. F. Menabrea, Bibliothèque Universelle de Genève (41), October 1842.
- Minsky, M. (1962). Size and structure of universal Turing machines using tag systems. In *Recursive Function Theory (Proceedings of Symposia in Pure Mathematics)*, volume 5, pages 229–238. AMS.
- Moore, C. and Mertens, S. (2011). *The Nature of Computation*. Oxford University Press.
- Nisan, N. and Schocken, S. (2005). *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press.
- Papadimitriou, C. H. (1994). *Computational Complexity*. Addison Wesley, Massachusetts.
- Petzold, C. (2008). *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley.
- Priestley, J. (1782). *Disquisitions Relating to Matter and Spirit: To which is Added the History of the Philosophical Doctrine Concerning the Origin of the Soul, and the Nature of Matter; With Its Influence on Christianity, Especially with Respect to the Doctrine of the Pre-existence of Christ*. Printed by Pearson and Rollason, for J. Johnson, 2nd edition. First published 1777.
- Russell, B. (1912–13). On the notion of cause. *Proceedings of the Aristotelian Society, New Series*, 13:1–26.
- Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, 3(3): 417–424.
- Sipser, M. (2013). *Introduction to the Theory of Computation*. Cengage, 3rd edition.
- Tseytin, G. S. (1970). On the complexity of derivation in propositional calculus. In Slisenko, A. O., editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, volume 8 of *Seminars in Mathematics, Steklov Mathematical Institute*, pages 115–25. Plenum. Translated from Russian, originally published 1968.
- Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265. Initially published Nov.–Dec. 1936 in two parts, republished 1937.
- Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.
- Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.

# 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
- BOTHCONTAININGAGA, 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<sub>F</sub>**, 132–134  
 uncomputability of, 132  
**COMPUTESISEVEN**, 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  
**containsGAGAandCACAndTATA.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
  - $\epsilon$ -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
- Expo, 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  
 HALTSONEEMPTY, 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  
 IS EVEN, 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, Christopher, 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
- NOTHASPATH, 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
- NUMCHARONSTRING, 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  
    polytyme, 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

