LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable

Edna E. Reiter
Clayton Matthew Johnson

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable

LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable

Edna E. Reiter
Clayton Matthew Johnson

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2013 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20120808

International Standard Book Number-13: 978-1-4398-8207-8 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface, xi

Acknowledgments, xiii

About the Authors, xv

Introduction, xvii

CHAPTER 1 ▪ Set Theory 1

1.1 SETS—BASIC TERMS 1
1.2 FUNCTIONS 6
1.3 CARDINALITIES 7
1.4 COUNTING ARGUMENTS AND DIAGONALIZATION 8
EXERCISES 12

CHAPTER 2 ▪ Languages: Alphabets, Strings, and Languages 13

2.1 ALPHABETS AND STRINGS 13
2.2 OPERATIONS ON STRINGS 17
2.3 OPERATIONS ON LANGUAGES 22
EXERCISES 30

CHAPTER 3 ▪ Algorithms 33

3.1 COMPUTATIONAL PROBLEMS 33
3.2 DECISION PROBLEMS 36
3.3 TRAVELING SALESMAN PROBLEM 38
3.4 ALGORITHMS: A FIRST LOOK 39
3.5 HISTORY 41

3.6 EFFICIENCY IN ALGORITHMS 42
3.6.1 Why Polynomial? 43
3.6.2 Why Polynomial? Measuring Time 44
3.6.3 Size of the Input 44
3.7 COUNTING STEPS IN AN ALGORITHM 44
3.8 DEFINITIONS 45
3.9 USEFUL THEOREMS 46
3.10 PROPERTIES OF *O* NOTATION 48
3.11 FINDING *O*: ANALYZING AN ALGORITHM 48
3.12 BEST AND AVERAGE CASE ANALYSIS 53
3.13 TRACTABLE AND INTRACTABLE 54
COMMENTS 55
EXERCISES 56

CHAPTER 4 ■ Turing Machines 61
4.1 OVERVIEW 61
4.2 THE TURING MACHINE MODEL 61
4.3 FORMAL DEFINITION OF TURING MACHINE 62
4.3.1 Input Alphabet 63
4.3.2 Tape Alphabet 63
4.3.3 Set of States 64
4.3.4 Accept State 64
4.3.5 Reject State 64
4.3.6 Start State 64
4.3.7 Transition Function 65
4.4 CONFIGURATIONS OF TURING MACHINES 66
4.5 TERMINOLOGY 67
4.6 SOME SAMPLE TURING MACHINES 69
4.7 TURING MACHINES: WHAT SHOULD I BE ABLE TO DO? 74
4.7.1 Decide If a Given Diagram Describes a Turing Machine 74

4.7.2 Given a Turing Machine, Trace Its Operation on a Given String 75
4.7.3 Given a Turing Machine, Describe the Language It Accepts 75
4.7.4 Given a Language, Write a Turing Machine That Decides (or Accepts) This Language 75
EXERCISES 76

CHAPTER 5 ■ Turing-Completeness 81
5.1 OTHER VERSIONS OF TURING MACHINES 81
5.1.1 Semi-Infinite Tape 82
5.1.2 Stay Option 84
5.1.3 Multiple Tapes 85
5.1.4 Nondeterministic Turing Machines (NDTMs) 90
5.1.5 Other Extensions and Limitations of Turing Machines 97
5.2 TURING MACHINES TO EVALUATE A FUNCTION 98
5.3 ENUMERATING TURING MACHINES 98
5.4 THE CHURCH–TURING THESIS 99
5.5 A SIMPLE COMPUTER (OPTIONAL) 101
5.6 ENCODINGS OF TURING MACHINES 104
5.7 UNIVERSAL TURING MACHINE 108
EXERCISES 110

CHAPTER 6 ■ Undecidability 113
6.1 INTRODUCTION AND OVERVIEW 113
6.1.1 Problems That Refer to Themselves (Self-Reference) 113
6.1.1.1 Problem 1: The Barber 114
6.1.1.2 Problem 2: Grelling's Paradox 115
6.1.1.3 Problem 3: Russell's Paradox 115
6.2 SELF-REFERENCE AND SELF-CONTRADICTION IN COMPUTER PROGRAMS 115

6.2.1 Problem 1: The "Hello World" Writing Detector Program 115
6.2.2 Problem 2: The Infinite Loop Detector Program 118
6.3 CARDINALITY OF THE SET OF ALL LANGUAGES OVER AN ALPHABET 121
6.4 CARDINALITY OF THE SET OF ALL TURING MACHINES 122
6.5 CONSTRUCTION OF THE UNDECIDABLE LANGUAGE $ACCEPT_{TM}$ 124
EXERCISES 126

CHAPTER 7 ■ Undecidability and Reducibility 129
7.1 UNDECIDABLE PROBLEMS: OTHER EXAMPLES 129
7.2 REDUCIBILITY 132
7.3 REDUCIBILITY AND LANGUAGE PROPERTIES 134
7.4 REDUCIBILITY TO SHOW UNDECIDABILITY 135
7.5 RICE'S THEOREM (A SUPER-THEOREM) 139
7.6 UNDECIDABILITY: WHAT DOES IT MEAN? 141
7.7 POST CORRESPONDENCE PROBLEM (OPTIONAL) 141
7.8 CONTEXT-FREE GRAMMARS (OPTIONAL: REQUIRES SECTION 7.7) 153
EXERCISES 157

CHAPTER 8 ■ Classes **NP** and **NP**-Complete 161
8.1 THE CLASS **NP** (NONDETERMINISTIC POLYNOMIAL) 161
8.2 DEFINITION OF **P** AND **NP** 161
8.3 POLYNOMIAL REDUCIBILITY 165
8.4 PROPERTIES 167
8.5 COMPLETENESS 168
8.6 INTRACTABLE AND TRACTABLE—ONCE AGAIN 169
8.7 A FIRST **NP**-COMPLETE PROBLEM: BOOLEAN SATISFIABILITY 171
8.8 COOK–LEVIN THEOREM: PROOF 174

8.8.1 Proof Part I: Construction of the Clauses 175
8.8.2 Proof Part II: Construction in Polynomial Time and Correctness 179
8.9 CONCLUSION 180
EXERCISES 180

CHAPTER 9 ■ More **NP**-Complete Problems 185
9.1 ADDING OTHER PROBLEMS TO THE LIST OF KNOWN **NP**-COMPLETE PROBLEMS 185
9.2 REDUCTIONS TO PROVE **NP**-COMPLETENESS 185
9.2.1 Restriction (Also Called Generalization) 186
9.2.2 Local Replacement 187
9.2.3 Component Design 189
9.3 GRAPH PROBLEMS 190
9.4 VERTEX COVER: THE FIRST GRAPH PROBLEM 194
9.5 OTHER GRAPH PROBLEMS 199
9.6 HAMILTONIAN CIRCUIT (HC) 201
9.7 EULERIAN CIRCUITS (AN INTERESTING PROBLEM IN **P**) 208
9.8 THREE-DIMENSIONAL MATCHING (3DM) 209
9.9 SUBSET SUM 215
9.10 SUMMARY AND REPRISE 219
EXERCISES 220

CHAPTER 10 ■ Other Interesting Questions and Classes 225
10.1 INTRODUCTION 225
10.2 NUMBER PROBLEMS 225
10.3 COMPLEMENT CLASSES 230
10.4 OPEN QUESTIONS 231
10.5 ARE THERE ANY PROBLEMS IN **NP**-**P** BUT NOT **NP**-COMPLETE? 232
10.5.1 Linear Programming 236
10.5.2 PRIME 237

10.5.3 Graph Isomorphism 237
10.5.4 Other Examples? 237
10.6 **PSPACE** 237
10.7 REACHABLE CONFIGURATIONS 240
10.8 **NP**SPACE = **PSPACE** 241
10.9 A **PSPACE** COMPLETE PROBLEM 243
10.9.1 Quantified Boolean Formulas (QBFs) 245
10.10 OTHER **PSPACE**-COMPLETE PROBLEMS 247
10.11 THE CLASS EXP 248
10.12 SPACE RESTRICTIONS 249
10.13 APPROACHES TO HARD PROBLEMS IN PRACTICE 250
10.14 SUMMARY 251
EXERCISES 252

BIBLIOGRAPHY, 253

Preface

To the Student: We think that the theory dealing with what is hard about computation (and what is impossible!) is challenging but fun. This book grows out of these ideas and our approach to teaching a course in computational complexity.

There is no doubt that some of the material in these chapters is what might be called "wrap your brain around it" material, where a first reaction might be that the authors are pulling off a trick like a magician pulling a rabbit out of a hat. For instance, consider the proof—using contradiction—that there can be no algorithm to tell whether a program written in C++ will go into an infinite loop. One reaction upon reaching the contradiction might be that there must be a misstep somewhere in the proof; another might be that there cannot really be a contradiction. Only after reading, rereading, and carefully considering each step can the student buy into the proof. There are no shortcuts here; this is not reading to be done with the television playing in the background.

There are also diversions here such as the bridges of Königsberg problem—interesting but easy, and useful to point out that there can be vast differences in the difficulty of problems that sound very much alike.

To the Instructor: We hope you have as much fun explaining the difficulties and complexities of computation as we do.

At California State University, East Bay, there is a required course for Computer Science Master of Science students in complexity theory.

This book grew out of the problems we had in choosing a text for this course. Sipser's *Introduction to the Theory of Computation* (Gale/Cengage Learning, 2006) has many good points, and we recommend it to all students as a reference—but it covers automata theory and formal languages (Part 1) as well as Computability Theory (Part 2) and Complexity (Part 3), which leads to less depth than we might like. And for a course that does not cover Part 1, the references back to it from Parts 2 and 3 are a bit problematic.

The Hopcroft, Motwani, and Ullman (2007) text is exemplary in many ways, and is another excellent reference; it is, however, more suitable to students in a doctoral program or those in an advanced course in complexity.

And of course, there is the wonderful Garey and Johnson (1979) text on NP-completeness. We love to point out to our computer science students that this book is still important and that every computer scientist should own a copy. Can you say that about any other book on technology of that vintage? There are also good, older books available, although not at a level appropriate for an M.S. student without a strong background in automata. (See Brainerd and Landweber, 1974; Harrison, 1978; Lewis and Papadimitriou, 1997; and Papadimitriou, 1994.)

We also think that the more popular books that mention Turing machines (such as those by Douglas Hofstader, 1979 and Penrose, 1989) are nice supplemental extracurricular material, although students must become adept at reading different models of Turing machines.

This book, then, is intended for advanced undergraduates or beginning graduate students who may not have a strong background in theoretical computer science and who do not plan to become experts in the area.

The book is designed so that essentially all of it can be covered in a one-quarter (4 hour/week) or one-semester (3 hour/week) course, with roughly half the course devoted to what is undecidable, and half to what is intractable. To do this, it may be necessary to omit some proofs (such as the proof that if $\mathbf{P} \neq \mathbf{NP}$, then there are problems in **NP** that are not **NP**-complete), but we feel that such proofs must be included in the text for the interested reader.

The authors have included a wide range of exercises. There is no way to learn this material without doing it. We feel strongly that the student should read this book with pencil in hand, filling in any missing details ("it is easily shown that …"). The student then needs to test his or her understanding by doing a variety of exercises, and he or she needs feedback that his or her approach to solving the exercises was (or was not) correct. We give weekly graded written assignments, and either one or two midterm exams in a 10-week quarter, plus a final exam. We have read at least one study on pedagogy stating that something must be learned three times in order to be mastered and for the learning to last: we hope that in doing a homework set, in studying for a midterm, and then again in studying for a final exam, we have forced our students into rethinking the material at least three times.

Acknowledgments

The authors wish to thank the many people who have helped in the path toward publication. First and foremost, there is the California State University, East Bay (CSUEB) office staff—Richard Uhler, who helped with many technical problems, and Susan Foye, Kimberly Cherry, and Erendira McDunn, who helped with the administrative details and kept the department running. We thank our colleagues Shirley Veomett, who helped with the illustration on function growth, and Steve Simon, who also teaches complexity. We greatly appreciate the efforts of Stan Wakefield, literary agent, and Amy Blalock, project coordinator and Linda Leggio, project editor at Taylor & Francis/CRC Press. We also thank our families and friends, particularly Jim Reiter and Grant Petersen, who supported us during the writing (and rewriting) of the book.

About the Authors

Drs. Edna Reiter and Clayton Matthew Johnson are faculty in the Department of Mathematics and Computer Science at California State University, East Bay (CSUEB) in the San Francisco Bay area. Together, they developed the subject matter of the CSUEB course Computation and Complexity, required for all students in the Master of Science program. The course covers the hard problems of computer science—those that are intractable or undecidable. The text and the exercises in the text have been tested on multiple sections of CSUEB students.

Edna E. Reiter, Ph.D., received M.S. degrees from the University of Michigan and the University of California at Davis, and a Ph.D. from the University of Cincinnati. Her initial research interests were in noncommutative ring theory, but she then became interested in the theoretical aspects of computer science. Dr. Reiter has discovered that she enjoys introducing students to these subjects. She is currently Chair of the Department of Mathematics and Computer Science at CSUEB..

Clayton Matthew Johnson, Ph.D., received an M.S. degree from Michigan State University, and holds a Ph.D. from the College of William and Mary in Virginia. His current research interests are genetic algorithms and machine learning. He teaches many of the core courses for both graduate and undergraduate students including data structures, automata theory, analysis of algorithms, and complexity. Dr. Johnson is the graduate coordinator for all M.S. students at CSUEB; he is also the incoming Chair of the Department of Mathematics and Computer Science.

Introduction

ALGORITHM: WHAT IS IT, WHEN DOES ONE EXIST?

Most of the computer science courses in the first several years of a student's experience are concerned with teaching algorithms, and getting students to implement them. The student of computer science spends many hours in many courses learning how to write programs to solve problems. She learns about sorting data, saving data in various data structures, maintaining databases, managing networks, creating graphics, and more. She learns algorithms for supporting an operating system, maintaining database purity, parsing source files, and so on. The question of unsolvable problems may not even appear—the instructor, the text, and the student are too busy learning the efficient algorithms that are in use.

Even the word *algorithm* is often used without a good definition—and expressing that definition will be a major theme of this text, one that cannot be answered in just a few paragraphs.

IMPORTANT QUESTIONS

Only in a few places does the computer science student study such questions as: What is computation? What is an algorithm? How do I know that this problem has a solution? If there is a solution, will it answer the problem fast enough? An answer in the next century is no better than no answer at all.

Any "educated" computer scientist needs to know something about the answers to these questions. There are many easy-to-state questions in computer science that either have no algorithm at all, or have no practical algorithm. These include:

- Given a context-free grammar G, is it ambiguous?
- What is the shortest route for a salesman to take, starting at his home, visiting all the cities on his route?
- Given a program written by another computer science student, will this program terminate? Or will it go into an infinite loop?
- Given a set of processes running on a system, will they end in deadlock?

These questions have intrigued computer scientists and others. Some popular books address some of the same questions as this text—see Hofstadter (1979) and Penrose (1991). Some questions have monetary prizes for anyone who can solve them (see Devlin, 2002).

DOES MY PROBLEM HAVE A SOLUTION? A GOOD SOLUTION?

The computer scientist should be aware of questions like these, and be suspicious of new assignments—if told to write code to solve problem X, it would be nice to know that X has a solution and that this solution will not require centuries or millennia to execute.

Questions to consider before beginning are: Is there a known good algorithm for this problem? Or, are there algorithms that work, but take too long? A problem that used to be difficult, but now has become easier, is the forecasting of weather. Weather (like many physical systems) obeys a complicated set of differential equations, and it can be forecast by getting initial data points—the current weather—and solving these equations. However, if it takes longer to solve the equations than for weather to happen, these forecasts are not useful. It does not do much good to have the forecast for Tuesday on the following Wednesday. Problems like this one generated research in methods for solving differential equations, and considerable progress has been made.

But what problems are like this—solvable with good methods? Are there any problems that are difficult now and that we cannot expect ever to have good solutions?

Of course, if one needs a solution to a problem that has no solution—or no reasonable solution—then what? Here too, some knowledge is useful. It is not necessary just to give up and say, "Can't be done." Knowing the difficulties (and the options around them) is the first step.

- What is an algorithm?
- What is a computation?
- What is a computer?
- When does an algorithm exist?
- When do reasonable algorithms exist?
- What is meant by *reasonable*?

FIGURE I.1 Questions for this course.

In short, an educated computer scientist needs more than tools of programming. He or she needs to understand what is possible to program—which is the topic of this book.

THE "BIG" IDEAS

The two big questions that this book deals with are:

Which problems have no algorithm at all (and what does that mean)?

Which problems cannot be solved efficiently (and what does that mean)?

Answering these questions—and even having the machinery to properly pose them as questions—will take some time and effort.

This can be summarized in Figure I.1.

CHAPTER 1

Set Theory

STUDENTS WILL HAVE SEEN set theory before and thus, the following is a brief review. Some important ideas, though, may be new and will be covered in more detail.

1.1 SETS—BASIC TERMS

Definition 1.1

A **set** is a collection of objects. ■

Definition 1.2

A **member** or **element** is an object in a set. A set is said to **contain** its elements. ■

Elements in a set are listed in braces.

Examples

$$S_1 = \{1, 2, 3, 4\}$$

$$S_2 = \{a, b, c\}$$

$$S_3 = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$$

Repetition does not matter in a set and ordering means nothing, so {a, b, c} = {b, a, c, b}.

Sets can be finite or infinite. Ellipses can be used in set notation once a pattern of membership has been established.

Examples

$$S_4 = \{1, 2, 3, \ldots, 98, 99, 100\}$$

$$S_5 = \{1, 2, 3, \ldots\}$$

$$S_6 = \{\ldots, -3, -2, -1\}$$

$$S_7 = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

Sets can also be described using Peano's notation.

$$S = \{x \mid x \text{ satisfies some condition}\}$$

Examples

$\{x \mid x = y^2 \text{ and } y \text{ is an integer}\}$ {squares}

$\{x \mid x = 2y \text{ and } y \text{ is an integer}\}$ {even numbers}

There must always be an underlying **universal set** *U*, either specifically stated or implicit. Some common universal sets include:

$\mathbf{N} = \{0, 1, 2, 3, \ldots\}$	(natural or counting numbers)
$\mathbf{Z} = \{\ldots, -2, --1, 0, 1, 2, \ldots\}$	(integers)
$\mathbf{Z}^+ = \{1, 2, 3, \ldots\}$	(positive integers)
$\mathbf{Z}^- = \{\ldots, -3, -2, -1\}$	(negative integers)
$\mathbf{Q} = \{x : x = m/n, m, n$ are integers, $n \neq 0\}$	(rational numbers)
$\mathbf{R}$ = real numbers	

Set membership is indicated by the $\in$ symbol, and **set exclusion** (is not a member) by $\notin$.

Examples

$$a \in \{a, b, c\}$$

$$d \notin \{a, b, c\}$$

Definition 1.3

The set A is a **subset** of set B, denoted $A \subseteq B$, if and only if (iff) every member of A is also a member of B. ■

Example

{a}, {b, c}, and {c, b, a} are some of the subsets of {a, b, c}.

Definition 1.4

The **empty set**, denoted $\varnothing$, is the set {}. It contains no elements. ■

Definition 1.5

The set A is a **proper subset** of set B iff every member of A is also a member of B and $A \neq B$, denoted $A \subset B$. ■

Example

{a}, {b, c} are some proper subsets of {a, b, c}.

The empty set is a subset of every set, and a proper subset of every set except itself.

Definition 1.6

The standard set operations are union, intersection, difference, and complement. They are defined as:

The **union** of two sets A and B, denoted $A \cup B$, is the set $\{x \mid x \in A \text{ or } x \in B\}$.

The **intersection** of two sets A and B, denoted $A \cap B$, is the set $\{x \mid x \in A \text{ and } x \in B\}$.

The **difference** of two sets A and B, denoted $A - B$, is the set $\{x \mid x \in A \text{ and } x \notin B\}$.

The **complement** of a set A, denoted $\bar{A}$ or A^c, is the set $\{x \mid x \notin A \text{ and } x \in \mathbf{U}\}$. ■

Definition 1.7

A **multiset** is a set in which the repetition of elements is important. Order is still irrelevant in a multiset. ■

Example

$\{4, 1, 2, 4, 1\} \neq \{4, 1, 2\}$ (for multisets)

$\{4, 1, 2, 4, 1\} = \{4, 1, 2\}$ (for sets)

$\{4, 1, 2, 4, 1\} = \{1, 1, 2, 4, 4\}$ (for multisets and sets)

Definition 1.8

A **well-ordered set** is a set in which there is a natural ordering of the elements such that for any two distinct elements e_1 and e_2 in the set, either $e_1 < e_2$ or $e_1 > e_2$. For example, the English language alphabet {a, b, c, …, x, y, z} is a well-ordered set. We rely on this fact when we alphabetize. ■

Definition 1.9

A **sequence** is a list of objects in an order. Elements in a sequence are listed in parentheses. ■

Example

(a, b, r, a, c, a, d, a, b, r, a)

(3, 1, 4, 1, 5, 9, 2)

Repetition and order both matter in a sequence, so (1, 2, 3) ≠ (1, 1, 2, 3) ≠ (2, 1, 3).

Definition 1.10

An **empty sequence** is the sequence (). ■

As with sets, a sequence can be finite or infinite. The set of natural numbers can be viewed as a sequence (0, 1, 2, 3, …).

Finite sequences have particular names.

Definition 1.11

A **tuple** is a finite sequence.

An ***n*-tuple** is a sequence containing exactly n elements. The sequence (a, b, c) is therefore a 3-tuple, and the sequence (1, 2, 3, 4) is a 4-tuple.

An **ordered pair** is a 2-tuple. ■

Definition 1.12

The **power set** of A, denoted P(A), is the set of all subsets of A. ■

Examples

$$P(\{a, b, c\}) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

$$P(\{1, 2\}) = \{\varnothing, \{1\}, \{2\}, \{1, 2\}\}$$

$$P(\varnothing) = \{\varnothing\}$$

Definition 1.13

The **Cartesian product** or **cross-product** of two sets A and B, denoted A × B, is the set {(x, y): x ∈ A and y ∈ B}. ■

Example

$$\{a, b\} \times \{c, d\} = \{(a, c), (a, d), (b, c), (b, d)\}$$

$$\{1, 2, 3\} \times \{1, 2\} = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)\}$$

$$\{a, b, c\} \times \varnothing = \varnothing$$

1.2 FUNCTIONS

Again, functions are a concept quite familiar to computer science.

Definition 1.14

A **function** or **mapping** from set A to set B (written f: A $\rightarrow$ B) is a subset of A $\times$ B such that each x $\in$ A is associated with a *unique* y $\in$ B.

For f: A $\rightarrow$ B:

- A is called the **domain** of f.
- B is called the **codomain** of f.

If f(x) = y:

- y is called the **image** of x under f.
- x is the **preimage** of y under f. ■

Thus, the mapping from a person to his or her mother is a function (assuming exactly one mother per person), but the mapping from a person to his or her child is not. The mapping (person x, mother of X) has a domain of all people—since every person has a mother, and a codomain of the set of women who have children.

Definition 1.15

A function f from a set A to a set B is an **injection** if no two values from A are mapped to the same element of B (f(x) = f(y) implies that x = y). It is a **surjection** if it is onto B (for every b $\in$ B, there is an x $\in$ A such that f(x) = b). It is a **bijection** or **one-to-one correspondence** if it is both an injection and a surjection (one-to-one and onto). ■

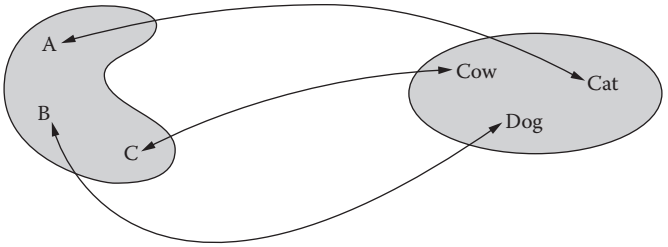


FIGURE 1.1 Example of a bijection.

Example

There are six different bijections from {A, B, C} to {cow, cat, dog}. Any one of these mappings would show that the two sets are in one-to-one correspondence. An example of one such mapping is shown in Figure 1.1.

1.3 CARDINALITIES

Before we can discuss cardinality, we must review what we mean when we say that two sets are the "same size." For example, consider the two sets below:

{A, B, C} {cow, dog, cat}

The "wrong" answer is that the two sets both have three elements—or perhaps more correctly, counting elements to three is all right, but does not go back to first principles. **Why** do they each have three elements? What does it mean to have three elements? The "right" answer, the one that goes back to the foundations of arithmetic, is that there is a one-to-one correspondence between them—for example, A ↔ cat, B ↔ cow, and C ↔ dog. The reason they both have three elements is that each is in one-to-one correspondence with some set that exhibits the basics of "threeness."

Definition 1.16

Let $\boldsymbol{N_k}$ be the first k natural numbers $\{0, 1, 2, \ldots, k - 1\}$. A set S is **finite** iff a bijection exists f: S $\rightarrow \boldsymbol{N_k}$ for some natural number k. ■

Definition 1.17

A set S is **countable** iff there is a surjective function f: $\mathbf{N} \rightarrow$ S. A set S is **infinite** iff there is a surjective function f: S $\rightarrow \mathbf{N}$. ■

Georg Cantor (German mathematician, 1845–1918) proved that there are different sizes of infinities. For example, there are obviously an infinite number of natural numbers, since the mapping $f(i) = i$ is a surjective function f: $\mathbf{N} \rightarrow \mathbf{N}$.

0, 1, 2, 3, 4, 5, … (base 10)

0, 1, 10, 11, 100, … (base 2)

There are also an infinite number of real numbers. Even in the half-open interval on the real numbers, $\mathbf{R}[0, 1) = \{x : 0 \leq x < 1\}$, there are an infinite number of numbers (you should be able to find a surjective function from f: $\mathbf{R}[0, 1) \rightarrow \mathbf{N}$).

Recall that each real number in this (or any) interval has a decimal expansion such as 0.1210….

These expansions are infinite. Some, like the decimal for 1/5 or the binary expansion of 1/4, are all zeros after a certain point. Some, like 1/7, have a repeating pattern, and some, like $\pi/10$, have no pattern at all.

But, what kind of infinities are these? More on that shortly.

1.4 COUNTING ARGUMENTS AND DIAGONALIZATION

Many infinite sets have the same cardinality; there are one-to-one correspondences between the two. Three simple examples would be:

$\mathbf{Z}^+$ = positive integers = 1, 2, 3, …

$\mathbf{Z}_{n\geq 10}$ = positive integers starting at 10 = 10, 11, 12…

Correspondence: $f(n) = n + 9$

$\mathbf{Z}^+$ = positive integers = 1, 2, 3, …

$\mathbf{Z}_{even}$ = even positive integers = 2, 4, 6, 8, …

Correspondence: $f(n) = 2n$

$\mathbf{Z}^+$ = positive integers 1, 2, 3, …

$\mathbf{Z}^{+3} = \mathbf{Z}^+ \times \mathbf{Z}^+ \times \mathbf{Z}^+ = \{(x, y, z) \mid x, y, z \in \mathbf{Z}^+\}$

Correspondence: List the triples in some order, for example, start with n = 3.

List all that add to n (n initialized to 3)	$(1, 1, 1) \leftrightarrow 1$
Increment n, list all that add to n: n = 4	$(1, 1, 2) \leftrightarrow 2$
	$(1, 2, 1) \leftrightarrow 3$
	$(2, 1, 1) \leftrightarrow 4$
n = 5	$(1, 1, 3) \leftrightarrow 5$
	$(1, 2, 2) \leftrightarrow 6$
	$(1, 3, 1) \leftrightarrow 7$
	$(2, 1, 2) \leftrightarrow 8$
	$(2, 2, 1) \leftrightarrow 9$
	$(3, 1, 1) \leftrightarrow 10$
n = 6	$(1, 1, 4) \leftrightarrow 11$
	etc.

Note the well-defined order. All numbers that add to n are listed before those that add to (n + 1). In the triples that sum to n, all those with 1 in the first position are listed before those that begin with 2, and so on; likewise in the second position. Thus, with the sum n = 12, the first triples are (1, 1, 10), (1, 2, 9) … (1, 10, 1), (2, 1,, 9), … (10, 1, 1).

Any well-defined order will work, such as the one above—where every triple (r, s, t) will be enumerated at some point when enumerating the triples that sum to u = r + s + t. So the mapping is onto. No two triples match the same integer, so it is also one-to-one and a bijection.

Set 1 = $\mathbf{R}(0, 1)$ = reals from 0 to 1, not including 0 or 1.

Set 2 = $\mathbf{R}$ = all reals.

Correspondence: Any bijective function, for example, $f(x) = \tan \pi(x - ½)$ or $g(x) = (x - 1/2)/x(x - 1)$.

Theorem 1.18 (Cantor)

There is no one-to-one correspondence between the set of all natural numbers and the set of reals **R**[0,1).

The proof is not difficult. First, let's represent all numbers in binary. Second, to make the representation for a binary fraction unique, we need to disallow any infinite sequence of 1's. For example, these two reals denote exactly the same real number:

$$0.001111111\ldots = 0.010000\ldots$$

so we use the second representation for the number 1/4.

(You may ask why the two numbers above are the **same** number. Two numbers are only different if there is some distance between them, or if, when you subtract one from the other, you do not get zero. Try the above subtraction.)

Now, to prove something does **not** exist, assume that it does and arrive at a contradiction. Thus, assume that there is a one-to-one correspondence between the natural numbers and **R**[0, 1).

$r_1 = 0.\mathbf{0}\,1\,0\,1\,1\,1\,0\ldots \leftrightarrow 1$
$r_2 = 0.\,0\,\mathbf{1}\,0\,1\,0\,1\,1\ldots \leftrightarrow 10$
$r_3 = 0.\,0\,1\,\mathbf{1}\,1\,0\,0\,1\ldots \leftrightarrow 11$
$r_4 = 0.\,1\,0\,1\,\mathbf{0}\,1\,1\,0\ldots \leftrightarrow 100$ Diagonal = .0110110 …
$r_5 = 0.\,1\,1\,0\,1\,\mathbf{1}\,0\,0\ldots \leftrightarrow 101$
$r_6 = 0.\,0\,1\,0\,1\,1\,\mathbf{1}\,1\ldots \leftrightarrow 110$
$r_7 = 0.\,0\,1\,0\,0\,1\,1\,\mathbf{0\ldots} \leftrightarrow 111$
…

To be a one-to-one correspondence, every real number must appear once, and every natural number must appear once. However, we can construct a real that does not correspond to any natural number, that is, it is not r_k for any k.

Let $x = x_1\, x_2\, x_3 \ldots$ be defined by

$x_i = 0$ if the ith digit of r_i is 1
1 if the ith digit of r_i is 0

For the example above this will result in the following decimal expansion:

$$x = 0.\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ \ldots$$

This number cannot be r_k for any k, since it differs from r_k in the kth digit. This contradicts our one-to-one correspondence, and completes the proof. ■

The technique used in this proof is called **diagonalization**. We found the diagonal digits ($r_{i,i}$) and constructed an element x that differed from each r_i since its kth digit $x_k \neq r_{k,k}$.

Corollary 1.19

There is no one-to-one correspondence between the natural numbers and the set of infinite sequences over {0, 1}.

Proof

Exactly the same, only (1) take r_i as infinite sequences of {0, 1} instead of real numbers, and (2) there is no need to disallow the sequence (1, 1, 1, 1 …). ■

Cantor gave these two cardinalities the name $\aleph_0$ (the cardinality of the natural numbers, often called countable) and $\aleph_1$ (the cardinality of R[0.1), which is also the cardinality of the set of all real numbers, and the cardinality of the set of all infinite sequences over {0, 1}, and termed uncountable). Or, we can say that $\aleph_0 = |\mathbf{N}|$ and $\aleph_1 = |\mathbf{R}|$.

Note that in the $\aleph_1$ sequence $x = x_1\ x_2\ x_3 \ldots$, there are two choices of each of the x_i (0 and 1). Thus, we can say, in some sense, that

$$2 * 2 * 2 * \ldots * 2 = 2^{\aleph_0} = \aleph_1$$

with $\aleph_0$ the number of 2's.

EXERCISES

1. Prove that the following sets have the same cardinalities.
   a. The natural numbers (1, 2, 3 …) and the odd natural numbers (1, 3, 5 …)
   b. The natural numbers and the rational numbers (n/m, where n and m are natural numbers, m ≠ 0)

2. Definition: An algebraic number is any number that satisfies an equation of the form

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0$$

   where $n > 0$, every a_i is an integer, and a_n is nonzero.
   a. Show that the number $\sqrt{2}$ and the complex number i are both algebraic.
   b. Show that the cardinality of the algebraic numbers is countable.
   c. What does this say about the existence of nonalgebraic numbers? Can you produce a nonalgebraic number?

3. Let F = {f: f is a function from the reals (**R**) to the reals}. Show that the cardinality of F is greater than $\aleph_1$.

CHAPTER 2

Languages

Alphabets, Strings, and Languages

2.1 ALPHABETS AND STRINGS

The study of computation can be made more formal by the use of alphabets, strings, and languages. The basic goal of this text is to get the student of computer science to understand what it means to "solve a problem," and similarly, what it means that "a problem has no solution." Thus, this chapter is devoted to presenting the terms and definitions that will be needed to discuss solving problems in precise terms.

Definition 2.1

An **alphabet** is a finite set of symbols (or characters). (The Greek letter Σ is often used to denote an alphabet; occasionally, the letter Γ is used.) ■

Examples

$\Sigma = \{a, b, c\}$

$\Sigma = \{0, 1\}$

$\Sigma = \{$, $\}$

$\Sigma = \{a, b, c, d, \ldots, z, A, B, C, D, E, \ldots, Z\}$

There is no particular reason that a symbol has to be a connected line—consider the letter j, or the symbol $\bar{x}$. There is also no particular reason that a symbol has to have a well-defined name, except that then we would be reduced to making up names for each one (squiggle-1, squiggle-2, etc.). Exotic, artistic alphabets are hard to name, and thus hard to discuss and to write. The standard alphabets in formal language theory consist of some set of standard characters, perhaps with embellishments such as $\bar{x}$ or x'. In most class and text examples, alphabets are small. The alphabets {0, 1} or {a, b, c} are used in many examples.

Definition 2.2

A **string** over the alphabet Σ is a finite sequence of symbols from Σ. ■

Letters near the end of the alphabet are often used as names of strings (u, v, w, etc.). To stand for the symbols in the string, σ_i (or sometimes τ_i) will be used.

Definition 2.3

The **length** of a string w, denoted |w|, is the number of nonunique characters w contains. As strings are finite, every string has a defined lengths. ■

Examples

$w = abb = \sigma_1\sigma_2\sigma_3$ over $\Sigma = \{a, b, c\}$, $\sigma_1 = a$, $\sigma_2 = \sigma_3 = b$, $|w| = |abb| = 3$

$u = bbbb$ over $\Sigma = \{a, b, c\}$, $|u| = |bbbb| = 4$

$v = 1001111$ over $\Sigma = \{0, 1\}$, $|v| = 7$

Note: Sequences have different notations—often the notation used depends on whether the sequence is used in a course in mathematics, formal languages, or algorithms analysis.

w = ababa	(formal languages)
w = (a, b, a, b, a)	(algorithms analysis)

The shortest string over any alphabet is the *empty string*—a string with no characters at all. We denote the empty string as ε—the Greek letter *epsilon*—although some authors may use e or λ instead. Its length is zero, $|\varepsilon| = 0$.

There is no longest string over an alphabet. A string may be of any length, so long as it is finite.

Examples

Consider the two alphabets $\Sigma_1 = \{a, b, c\}$ and $\Sigma_2 = \{b, c, d\}$. The following are strings over both Σ_1 and Σ_2:

ε
bbbbbbbbbbbbbbbbbbbbbbbbbb
c
bcbcb

The following are *not* strings over Σ_1 and Σ_2:

abacabad	($d \notin \Sigma_1$, $a \notin \Sigma_2$)
b c c	(" "—that is, blank—$\notin \Sigma_1$ or Σ_2)

An exponent can be used to note the **repetition** or number of times a character is repeated within a string.

Example

$a^4b^5a^2$ = aaaabbbbbaa

b^{28} = bbbbbbbbbbbbbbbbbbbbbbbbbb

0^5 = 00000

$a^0 = \varepsilon$

1^3 = 111 (This is a string operation, not arithmetic!)

Definition 2.4

A **language** over Σ is a set of strings over Σ. ■

Examples

The following are languages over the alphabet $\Sigma = \{0, 1\}$:

$$L_1 = \{0, 01, 001, 0001, 00001, 000001\}$$

$$L_2 = \{\varepsilon, 111111, 000000000\}$$

$$L_3 = \{\varepsilon\}$$

$$L_4 = \{\}$$

An infinite language consists of an unbounded number of strings. It would be impossible in this case to list all of the elements of the language, so we instead simply describe its contents using Peano's notation.

Examples

The following are languages over the alphabet $\Sigma = \{a, b, c\}$—all but L_5 are infinite.

$$L_1 = \{w \mid w \text{ begins with b}\}$$

$$L_2 = \{w \mid w \text{ begins or ends with c}\}$$

$$L_3 = \{w \mid w \text{ contains an even number of a's}\}$$

$$L_4 = \{w \mid w \text{ does not contain c}\}$$

$$L_5 = \{w \mid w \text{ is a word in the Scrabble® dictionary}\}$$
$$= \{a, aa, aba, ba, baa, baba, bacca, cab \ldots\}$$

In addition to the above notation, we can also use ellipses when a pattern can be clearly established.

Example

The following are languages over the alphabet $\Sigma = \{0, 1\}$:

$$L_1 = \{1, 01, 001, 0001, 00001, \ldots\}$$

$$L_2 = \{1, 010, 00100, 0001000, \ldots\}$$

$$L_3 = \{\varepsilon, 00, 0000, 000000, \ldots\}$$

We can also use either of the above notations for finite languages if we wish to stress the structure of its component strings.

The smallest language over any alphabet Σ is $L_\varnothing = \varnothing = \{\}$, with no strings whatsoever (the language has zero strings, or $|L_\varnothing| = 0$). This is distinct from another small language, $L_\varepsilon = \{\varepsilon\}$, a language with exactly one string in the language ($|L_\varepsilon| = 1$). Remember this distinction! The string ε is not "nothing"; it is indeed a real string, just one with no symbols in it.

2.2 OPERATIONS ON STRINGS

There are many operations that can be done on strings—some simple, others more complicated.

The **reversal** of a string $w = \sigma_1\sigma_2 \ldots \sigma_n$ (written w^R) is obtained by placing its characters in the opposite order, $w^R = \sigma_n\sigma_{n-1} \ldots \sigma_1$. Note that the lengths are the same, $|w| = |w^R|$.

Examples

$$(abbc)^R = cbba$$

$$(00111)^R = 11100$$

$$(abba)^R = abba$$

$$\varepsilon^R = \varepsilon$$

The **concatenation** of two strings $u = \sigma_1\sigma_2 \ldots \sigma_n$ and $v = \tau_1\tau_2 \ldots \tau_m$ (written uv, u·v, or sometimes as u × v) is the string obtained by appending v onto the end of u, as $\sigma_1\sigma_2 \ldots \sigma_n\tau_1\tau_2 \ldots \tau_m$.

The length $|uv| = |u| + |v|$.

Examples

Consider the following strings over $\Sigma = \{0, 1\}$:

$$w = 000$$

$$x = 1$$

$$y = 01010$$

$$z = \varepsilon$$

Then:

$$wx = 0001$$

$$xy = 101010$$

$$yz = 01010$$

$$zw = 000$$

Note that $\varepsilon w = w\varepsilon = w$ for any string w. So in particular, $\varepsilon\varepsilon = \varepsilon\varepsilon\varepsilon = \varepsilon\varepsilon\varepsilon\varepsilon = \varepsilon\varepsilon \ldots \varepsilon = \varepsilon$.

One can take multiples of a string as well as symbols. The **repetition** of a string, also denoted by the use of an exponent, is the repeated concatenation of a string with itself.

Examples

Consider the following strings over $\Sigma = \{a, b, c\}$:

$$w = abcba$$

$$x = ca$$

$$y = bbb$$

$$z = \varepsilon$$

Then:

$$w^2 = abcbaabcba$$

$$w^3 = abcbaabcbaabcba$$

x^7 = cacacacacacaca

x^1 = ca

y^2 = bbbbbb

$y^0 = \varepsilon$ (no occurrences of y)

$z^6 = \varepsilon$

$z^0 = \varepsilon$

Repetition has a higher precedence than concatenation; thus:

ab^4 = abbbb

$(ab)^4$ = abababab

The string x is a **substring** of the string y if for some other strings u and v, y = uxv.

We will use w_i to denote a substring of the string w. For instance, if w = abb, then $w = w_1w_2$ with w_1 = ab and w_2 = b (as one of several possible ways to divide abb into two substrings).

Examples

String y = abb contains the following substrings:

Length 3: abb
Length 2: ab, bb
Length 1: a, b, b
Length 0: ε

The six unique substrings are therefore:

ε, a, b, ab, bb, aab

String z = bbbac contains the following substrings:

Length 5: bbbac
Length 4: bbac, bbba
Length 3: bac, bba, bbb
Length 2: ac, ba, bb, bb
Length 1: a, b, b, b, c
Length 0: ε

The 13 unique substrings are therefore:

ε, a, b, c, ac, ba, bb, bac, bba, bbb, bbac, bbba, bbbac

Note from the above examples that a string w of length n has 1 + (n(n + 1)/2) possible substrings as follows:

One substring of length n
Two substrings of length (n – 1)—can remove first or last symbol
Three substrings of length (n – 2)—remove first two, last two, or one from each end
Four substrings of length (n – 3)
…
n substrings of length 1
One substring of length 0

Recall that 1 + 2 + 3 + … + n = n(n + 1)/2; adding in ε yields the formula. These 1 + (n(n + 1)/2) substrings, however, may contain duplicates and not be unique (how many unique substrings are there of w = a^6?).

Definition 2.5

The **length-first lexicographical ordering** ***L***(v, w), or simply the **lexicographical order**, on strings over an alphabet Σ is defined so that v < w if:

1. $|v| < |w|$
2. $|v| = |w|$, $v = \sigma_1\sigma_2 \ldots \sigma_n$, $w = \tau_1\tau_2 \ldots \tau_n$ for some $i > 0$ such that $\sigma_j = \tau_j$ for $j = 1, 2, \ldots, i - 1$, and $\sigma_i < \tau_i$ in the ordering of symbols of the alphabet Σ. ■

And, as usual, $v \leq w$ if $v < w$ or $v = w$.

Note that the empty string ε is less than any nonempty string w.

This definition can be restated as: if one string is shorter than another, it is smaller; if two strings have the same length, the one that would appear first in a dictionary is smaller.

Example

$aab \leq aba$	$aab < aba$
$c < aaaa$	$\varepsilon < aaa$
$\varepsilon < c < aaa < aab < aba < aaaa$	$\varepsilon \leq c \leq aaa \leq aab \leq aba \leq aaaa$

The **homomorphic image** of a string is the string with substitutions made for each character. If there are two alphabets Σ_1 and Σ_2, and a mapping h that takes each character in Σ_1 to some string of symbols from Σ_2, then this is easily extended to strings over Σ_1 by replacing each character in the string by its image. (The name *homomorphism* comes from mathematics; it is a function in which $f(xy) = f(x)f(y)$—here, you are given what to do with each symbol; you extend that naturally to strings.)

Example

Let $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c\}$ with $0 \rightarrow abc$, $1 \rightarrow b$ (mapping defined on the two elements of Σ_1). That is, $h(0) = abc$, $h(1) = b$. Or, informally, what do you do with 0? Replace it with abc. What do you do with 1? Replace it with just b. Now you know what to do with the only two symbols in the alphabet, so you should know what to do with every string over {0, 1}.

$001 \rightarrow abcabcb$	$h(001) = abcabcb$
$110 \rightarrow bbabc$	$h(110) = bbabc$
$0 \rightarrow abc$	$h(0) = abc$
$\varepsilon \rightarrow \varepsilon$	$h(\varepsilon) = \varepsilon$

Example

Let $\Sigma_1 = \{0, 1, 2\}$, $\Sigma_2 = \{a, b\}$, $0 \rightarrow ab$, $1 \rightarrow \varepsilon$, $2 \rightarrow ba$, so $h(0) = ab$, $h(1) = \varepsilon$, $h(2) = ba$:

$00122 \rightarrow ababbaba$	$h(00122) = ababbaba$
$110 \rightarrow ab$	$h(110) = ab$
$202 \rightarrow baabba$	$h(202) = baabba$
$\varepsilon \rightarrow \varepsilon$	$h(\varepsilon) = \varepsilon$

Definition 2.6

The **shuffle** of strings v and w is a set of strings of the form $v_1w_1v_2w_2 \ldots v_nw_n$, where $n \geq 1$ and $v = v_1, \ldots, v_n$ and $w = w_1, \ldots, w_n$. The v_i and w_j are substrings of v and w. ■

Because some of the substrings can be ε, this does not mean the two strings are the same length or that characters must alternate. It does mean that the symbols of v and w both appear in order in any string in their shuffle.

For example, if v = 000 and w = 11111, then 01110110 is in the shuffle of the strings v and w. And if v = 01 and w = 110, then 11010 is in the shuffle of v and w. (Exercise: List other strings in the shuffle.) Note that for v = 01 and w = 110, all strings in the shuffle of v and w have two 0's and three 1's—but not all strings with two 0's and three 1's are in shuffle(v, w); for example, 00111 is not (why?).

Shuffles are not the only interesting operations on strings and languages. One can take first halves (or last halves, or middle thirds) of strings, cycle the characters in a string, and more. However, none of these operations have much general use. And if you think that shuffle is weird and of not much general use, you may be correct—generally, operations such as concatenation and reversal are of much more use (and simpler to understand).

2.3 OPERATIONS ON LANGUAGES

Languages are sets, so of course all the usual set operations are available on two languages L_1 and L_2: **union** ($L_1 \cup L_2$,), **intersection** ($L_1 \cap L_2$), **difference** ($L_1 - L_2$), and **complement** ($\overline{L_1}$ or L_1^c).

Examples

Consider the following languages over {a, b, c}:

$$L_1 = \{a, b, aabc, bbaabcc\}$$

$$L_2 = \{\varepsilon, aa, cba\}$$

$$L_3 = \{w \mid w = w^R\}$$

$$L_4 = \{w \mid w \text{ contains exactly one b}\}$$

$$L_5 = \{w \mid w \text{ starts with a}\}$$

$$L_6 = \varnothing$$

Union (all strings that are in either of the languages):

$$L_1 \cup L_2 = \{\varepsilon, a, b, aa, cba, aabc, bbaabcc\}$$

$$L_3 \cup L_4 = \{w \mid w = w^R \text{ or w contains exactly one b}\}$$

$$L_5 \cup L_6 = L_5$$

Intersection (strings that are in both of the languages):

$$L_1 \cap L_4 = \{b, aabc\}$$

$$L_2 \cap L_5 = \{aa\}$$

$$L_3 \cap L_6 = L_6$$

Difference (a string that is a member of the first language but not a member of the second):

$$L_2 - L_3 = \{cba\}$$

$$L_1 - L_5 = \{b, bbaabcc\}$$

$$L_4 - L_6 = L_4$$

Complement (strings not in the language):

$$L_2 = \{w \mid w \neq \varepsilon \text{ and } w \neq aa \text{ and } w \neq cba\}$$

$$L_4 = \{w \mid w \text{ does not contain exactly one b}\}$$

$$L_6 = \Sigma^*$$

But, because the elements of a language are strings, there are also other operations available.

Definition 2.7

The **concatenation** of two languages L_1 and L_2 is the set of all strings that can be formed by concatenating a string from L_1 with a string from L_2.

$$L_1L_2 = \{w \mid w = xy, x \in L_1 \text{ and } y \in L_2\}$$ ■

Examples

Consider the following languages over {0, 1}:

$L_1 = \{\varepsilon, 1, 00\}$

$L_2 = \{1, 11, 111\}$

$L_3 = \{w \mid w$ contains any number of 0's (no 1's); that is, $w = 0^k$ for some $k\}$

$L_4 = \{w \mid w$ contains any number of 1's; i.e., $w = 1^j$ for some $j\}$

$L_5 = \{\varepsilon\}$

$L_6 = \varnothing$

Then:

$L_1L_2 = \{1, 11, 001, 111, 0011, 1111, 00111\}$

L_3L_4 = {w | w contains any number of 0's followed by any number of 1's)

$L_2L_5 = L_2$

$L_4L_6 = L_6$

The concatenation of a language L with itself multiple times can also be written with the **repetition** operator.

Examples

Consider the following languages over {0, 1}:

$L_1 = \{0, 11\}$

$L_2 = \{w \mid \text{the length } |w| = 2\}$

$L_3 = \{\varepsilon\}$

$L_4 = \varnothing$

Now:

$L_1^2 = \{00, 011, 110, 1111\}$

$L_1^3 = \{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$

$L_2^2 = \{w \mid \text{the length } |w| = 4\}$

$L_2^k = \{w \mid \text{the length } |w| = 2k\}$

$L_3^k = L_3$

$L_4^k = L_4$ for $k > 0$

The **Kleene star** (or **Kleene closure**) of a language is $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$. Note that ε is always in L^* for any language L.

$$
\begin{aligned}
L^* &= \{w \mid w = x_1x_2 \ldots x_n, \text{ where } n \geq 0 \text{ and } x_i \in L\} \\
&= \{\varepsilon\} \cup L \cup LL \cup LLL \cup \ldots \\
&= \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots \cup L^k \cup \ldots
\end{aligned}
$$

Example

Consider the following languages over {a, b, c}:

$$L_1 = \{a, b\}$$

$$L_2 = \{aaa\}$$

$$L_3 = \{\varepsilon\}$$

$$L_4 = \varnothing$$

Then:

$$L_1^* = \{w \mid w \text{ does not contain } c\}$$

$$L_2^* = \{w \mid w \text{ contains only a and } |w| = 3k\}$$

$$L_3^* = L_3$$

$$L_4^* = \{\varepsilon\} \qquad \text{(since } \varepsilon \in L^* \text{ for all languages)}$$

When we take the * operator on a string, we are considering the string w as the language {w}; thus w^* is language consisting of all strings that can be formed through any number (zero or more) of repetitions of w.

Examples

Consider the following strings over $\Sigma = \{a, b\}$:

$$w = aaa$$

$$x = b$$

$$y = aba$$

$$z = \varepsilon$$

So:

$$w^* = \{\varepsilon, aaa, aaaaaa, aaaaaaaaa, \ldots\}$$

$$x^* = \{\varepsilon, b, bb, bbb, bbbb, bbbbb, \ldots\}$$

$$y^* = \{\varepsilon, aba, abaaba, abaabaaba, \ldots\}$$

$$z^* = \{\varepsilon\}$$

The **homomorphic image of a language** L under some mapping *h* from Σ_1 to $\Sigma_2{}^*$ is the homomorphic mapping of all of the strings in L.

Examples

I. Let $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c\}$, h: $0 \rightarrow abc$, $1 \rightarrow b$.

L_1 = strings that contain the substring 00

$L_2 = 0^n 1^m$

$h(L_1)$ = combinations of abc and b that contain the substring abcabc

$h(L_2) = (abc)^n (b)^m$

II. Let $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c\}$, h: $0 \rightarrow aa$, $1 \rightarrow \varepsilon$.

L_1 = strings that contain the substring 00

$L_2 = 0^n 1^m$

$h(L_1) = (aa)^{k+2} = \{a^4, a^6, \ldots\}, k \geq 0$

$h(L_2) = (aa)^k = a^{2k}, k \geq 0$

III. Let $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b\}$, h: $0 \rightarrow a$, $1 \rightarrow b$.
Then $h(L)$ is simply L with 0 becoming a, 1 becoming b, for any L.

One can also form such sets (or languages) from L_1 as

$$\textbf{Prefix}\ (L_1) = \{w \mid w \text{ is the prefix of some } x \text{ in } L_1\}$$

$$\textbf{Suffix}\ (L_1) = \{w \mid w \text{ is the suffix of some } x \text{ in } L_1\}$$

The prefixes of the string $x = \sigma_1\sigma_2 \ldots \sigma_n$ are $\varepsilon, \sigma_1, \sigma_1\sigma_2, \ldots, \sigma_1\sigma_2 \ldots \sigma_n$. Similarly, the suffixes are $\sigma_1\sigma_2 \ldots \sigma_n, \ldots, \sigma_{n-1}\sigma_n, \sigma_n, \varepsilon$. Note that a string of length n has n + 1 prefixes and n + 1 suffixes.

Example

Consider the following language over {0, 1}:

$$L_1 = \{10, 110\}$$

Now:

$$\textbf{Prefix}\ (L_1) = \{\varepsilon, 1, 10, 11, 100\}$$

$$\textbf{Suffix}\ (L_1) = \{\varepsilon, 0, 10, 110\}$$

One can also form **left and right quotients** of two languages:

$$L_1/L_2 = \{y \text{ in } \Sigma^* \mid yx \in L_1 \text{ for some } x \in L_2\} \quad \text{(right quotient)}$$

$$L_1\backslash L_2 = \{y \text{ in } \Sigma^* \mid \text{for some } x \in L_1, xy \in L_2\} \quad \text{(left quotient)}$$

This can be stated as: L_1/L_2 contains those strings such that it is possible to find a string in L_2 to add after this string and obtain a string in L_1. In other words, to see if w is in L_1/L_2 you have to see if there is any string that you can find from L_2 that, when added after w, puts the combined string in L_1. Think of L_1/L_2 as the strings you get if you start with a string in L_1 and remove a string from L_2 from the *right*.

$L_1\backslash L_2$, however, has those strings such that it is possible to find a string from L_1 that when added in the front (left) gives a string in L_2. Or again, think that the L_2 is on the top (numerator), and think of $L_1\backslash L_2$ as the strings obtained by beginning with a string from L_2 and removing a string from L_1 from the *left*.

These can be hard to understand. If we denote the special language $L_\varepsilon = \{\varepsilon\}$, then note that

$$L_1/L_\varepsilon = L_1$$

$$L_\varepsilon \backslash L_2 = L_2$$

$$L_\varepsilon/L_1 = L_\varepsilon \text{ or } \varnothing \quad \text{(depending on whether } \varepsilon \text{ is in } L_1\text{)}$$

L_1/Σ^* = Prefix (L_1) = {begin with something in L_1; remove anything from the right}

$\Sigma^*\backslash L_2$ = Suffix (L_2) = {begin with something in L_2; remove anything from the left}

Example

$$(01)^*/1 = (01)^*0$$

$$(01)^*/11 = \varnothing$$

$$0^*\backslash 0^*1^* = 0^*1^*$$

$$0^*1\backslash 0^*1^* = 1^*$$

A word on notation: 01 is a string, but $(01)^*$ and 0^*1^* denote sets of strings, or languages in our notation. (If this were a course in formal languages, we would say that $(01)^*$ and 0^*1^* are regular expressions that have an associated language, but here we are avoiding discussions of regular expressions.)

Thus, one could object to $(01)^*/1$ since $(01)^*$ is a language, and 1 is a *string*, but the / operation works on two *languages*. We will say that 1 can stand for both the string 1 and the language with only one string 1 (or $L_1 = \{1\}$), and that it is clear from the context which is being used.

One can also shuffle languages. The **shuffle** of L_1 and L_2 consists of all strings that are formed by shuffling any string v from L_1 and any string w from L_2. This is the set of strings that can be written in the form $v_1w_1v_2w_2 \ldots v_nw_n$, where $n \geq 1$ and $v_1, \ldots, v_n$ are strings such that the concatenation $v_1 \ldots v_n$ is in L_1 and $w_1, \ldots, w_n$ are strings such that $w_1 \ldots w_n$ is in L_2.

EXERCISES

1. Construct reasonable definitions for these new string operations. The result might be one string, or a set of strings (a language):

   a. Permute(w) = w with its characters rearranged.

   b. Half(w) = w with half its characters removed.

   c. First_half(w) = ?

   Can these definitions be extended to languages? If so, how? And if L = strings over $\Sigma = \{a, b\}$ that begin with aa and end with bb:

   d. What is Permute (L)?

   e. What is Half (L)?

   f. What is First_half (L)?

2. Consider these three languages, and these three homomorphisms ($\Sigma = \{a, b, c\}$).

   L_1 = strings that begin with aa and end with bb

   L_2 = strings with an even number of a's

   L_3 = strings that do not contain the substring bbb

   h_1: $a \rightarrow bb$, $b \rightarrow ab$, $c \rightarrow cc$

   h_2: $a \rightarrow \varepsilon$, $b \rightarrow abc$, $c \rightarrow \varepsilon$

   h_3: $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow aa$

   What is $h_i(L_j)$ for i = 1, 2, 3, and j = 1, 2, 3?

3.

   a. Define a language $L_{smaller\text{-}than\text{-}w}$. If w = aabbc, under your definition, what is $L_{smaller\text{-}than\text{-}w}$?

   b. For strings r, s, v, w with r < s and v < w, prove or disprove:

      i. rv < sw

      ii. $v^R < w^R$

iii. $w^R < v^R$

iv. For any string x in Prefix(v), x < v

4. Suppose $\Sigma = \{a, b, c\}$ and $v = v_1, \ldots, v_n$, n = 3, v = abbc. Give all possible divisions of v into three substrings, $v = v_1 v_2 v_3$.

5. Let $\Sigma = \{a, b, c\}$ with languages $L\varepsilon = \{\varepsilon\}$, $L_\varnothing = \varnothing$, $L_{3k} = \{w$ so that $|w| = 3k$ for some $k \geq 0\}$, $L_a = \{w \mid w$ contains an a$\}$. Find:

   a. $L_\varnothing L_{3k}$

   b. $L_\varepsilon L_{3k}$

   c. L_{3k}/L_a

   d. $L_{3k} \backslash L_a$

   e. Prefix (L_{3k})

   f. Suffix (L_a)

CHAPTER 3

Algorithms

3.1 COMPUTATIONAL PROBLEMS

A text with the word *computation* in the title will clearly need to describe what is meant by a computation. This is both easy (in one sense) and hard (in others).

Definition 3.1

A **computational problem** specifies a relation between two sets of finite sequences (or two sets of strings). ■

Recall that a finite sequence is a finite ordered list of elements from a set. For example, an integer is a finite sequence of digits (so that 921,233,456,181 is a sequence of length 15 as written, but the commas could be omitted, which would make it a sequence of length 12). An English word is a finite sequence of letters. An array of integers can be considered a finite sequence of integers—this is the usual interpretation in an algorithms analysis class—or as a finite sequence of digits with separators.

(23, 512, 312, 987, 45, 641) is a sequence of six integers

or

23,512,312,987,45,641 is a sequence of 21 characters from 0 to 9 and commas.

The domain of a computational problem is a set of data (the *questions*) called the input. The range is a set of *answers* called the output. A computational problem, then, describes how questions are related to answers.

For example, students in computer science are frequently asked to arrange data in ascending order. This is known as the **Sorting Problem** and is defined as follows.

Example: Sorting Problem

Input: $(i_1, i_2, \ldots, i_n)$
Output: $(o_1, o_2, \ldots, o_n)$, where $(o_1, o_2, \ldots, o_n)$ is a permutation of $(i_1, i_2, \ldots, i_n)$ such that $o_1 \leq o_2 \leq \ldots \leq o_n$

One solution to the sorting problem is the bubble sort algorithm, as it correctly maps inputs (questions) to outputs (answers).

Definition 3.2

An **instance** of a computational problem is one particular element from the input set. A **solution of an instance** is the corresponding element from the output set as specified by the computational problem. ■

For example, the instance (5, 4, 2, 3, 1) of the sorting problem has the solution (1, 2, 3, 4, 5), whereas the instance (20, 40, 20, 60, 20, 60, 80, 40, 20) has the solution (20, 20, 20, 20, 40, 40, 60, 60, 80).

Note that there are several possible solutions to some instances! It does not matter, for example, which of the four 20 elements from (20, 40, 20, 60, 20, 60, 80, 40, 20) we permute into the first position of the output sequence just as long as we place one of them at that location. There are in fact many (4! 2! 2!) different solutions to this instance!

There is nothing in the statement of the sorting problem that places a limit on the number of elements that an instance may have. In other words, the sorting problem can take any number of arguments. This means, for example, that the empty sequence () is a valid instance. It also means that an instance could be phenomenally, staggeringly, large. An instance cannot, however, be infinite. There must be some final element i_n of an instance, as—by definition—a computational problem only describes a relation between *finite* sequences.

There are, however, an infinite number of both instances and solutions to the sorting problem. This distinction is important. Although each instance and each solution to *every* computational problem must be finite, there may be infinitely many instances.

Consider another familiar problem from introductory computer science—searching.

Example: Searching Problem

Input: (k, i_1, i_2, …, i_n)
Output: (o), where $o = TRUE$ if $\exists j\ \{i_j = k\}$; $o = FALSE$ otherwise
Alternate output: (n), where $n = j$ if $\exists j\ \{i_j = k$ and $m < j$, $i_m \neq k\}$; $n = -1$ otherwise

The input to this computational problem might seem a bit confusing at first. In this case, the input sequence contains a key k followed by a list of elements i_1, i_2, …, i_n. The output of the searching problem is *TRUE* if the key exists in this list, or *FALSE* if it does not.

Just as with the sorting problem, there is no upper limit to the number of elements i that might appear in the input sequence—just as long as this number is finite. The empty sequence (), however, is not a valid instance of the searching problem, as a key k must always be the first element of every input sequence.

Note that even though there are only two possible solutions to the searching problem, there are still an infinite number of instances, and therefore an infinite number of question–answer pairings.

Example: Integer Addition Problem

Input: ($d_1 d_2 \ldots d_n$, $c_1 c_2 \ldots c_m$), where d_i and c_i are digits with base 10
Output: ($b_1 b_2 \ldots b_r$), where this is the proper sum of the two integers

One valid input instance would be (10,341, 79,855); its solution would be (90,196).

The problem of real addition is not a computational problem—there is no way to specify a pair of arbitrary real numbers (or even a single real number) as a finite sequence.

3.2 DECISION PROBLEMS

A computational problem is a mapping between sets of sequences. As a language is a set of strings (which are sequences), a computational problem can also be seen as a statement about strings. This connection is an important one, and is the basis of computation theory.

Most of the computer science algorithms in standard texts find something in a set of data ("find the shortest path from A to B in the given graph") or manipulate a set of input ("sort this array into ascending order" or "output the amounts for current paychecks for employees in this database"). A simplification—but one which we will be able to show can be made without really losing anything—is to restrict the problems considered to those with only yes/no answers.

Definition 3.3

A **decision problem** is a computational problem with output from the set {yes, no}. ■

Thus, a decision problem can be restated as a set membership question (if S is the set of yes instances, is this instance in S or not?).

The input is encoded as a string; this can be done in various ways (it is easy to encode an integer or an English sentence; a little less easy to encode a graph—and how would you encode a map?). Encodings of graphs will be discussed later.

Example

"Find the shortest path from A to B in graph G" becomes the decision problem: "Given a number k and a graph G with vertices A and B and the path $A = v_{i1}, v_{i2}, v_{i3}, \ldots, v_{in} = B$, is this a valid path in G with length $\leq k$?" Here, if we do not care about the actual vertices that make up the path, we can translate it to: "Given a number k and a graph G with vertices A, B, does G contain a path of length $\leq k$ from A to B?" This becomes the language problem (or decision problem):

Language problem $L_{path_of_length_k} = \{k, G, A, B : k \text{ integer}, G \text{ graph with vertices A and B, and G has a path from A to B of length} \leq k\}$

Example

Find an English language palindrome (phrase made up of legal words, that, ignoring spaces, punctuation, and upper/lowercase, is the same forwards as backwards).

Language problem $L_{palindrome}$ = {string s | s is a palindrome according to the above definition}
Yes instances: Level; madam, I'm Adam; A man, a plan, a canal, panama
No instances: This is a palindrome; this is not a palindrome

Example

"Find the nth prime number P" can be restated as the decision problem: "Given n and P, is P the nth prime?" (Assume that 2 is the first prime.)

Language problem: $L_{nth\ prime}$ = {n, P | n, P are positive integer numbers and P is the nth prime}
Yes instances: 1, 2 (two is the first prime); 4, 7; 7, 17
No instances: 2, 8; 7, 19

Example

"Add the integers n, m in binary to find their sum" is restated as "Given binary integers n, m, r, is r = n + m?"

Language problem: $L_{binary\ addition}$ = {n, m, r | n, m, r are integers in binary and r = n + m}
Yes instances: 10, 10, 100; 1, 1, 10
No instances: 10, 10, 101; 1, 1000, 1100

Example

"Given a legal chess board configuration and that it is White's move, which move should he make?" becomes "Given this chess board and a suggested move, can White force a win by making this move?"

In all cases that we will use, there are an infinite number of possible inputs, which are divided into the yes instances and the no instances. If

there were only a finite number of possible instances, the question could be answered by a simple table lookup, and would not be interesting.

Chess is something of an aberration—there are only a finite number of board configurations, and for each board configuration, only a finite number of possible moves. However, except for some near-end-game configurations with only a few pieces left on the board, the number of possible moves and countermoves is so large that no solution has been found. But, because of the finiteness, standard 8 by 8 chess does not fit into the problems considered in this course.

Computational problems can always—with varying degrees of difficulty—be restated as decision problems. Certainly, a solution to the general problem solves the decision problem, but the reverse is also true. A machine powerful enough to answer the decision problem can—perhaps with some effort—do the path finding, prime generation, or addition (by restatements such as those above). Thus, in formal language and computation theory, we will not lose any generality by restricting ourselves to decision problems, and we will state almost all problems as language membership questions.

3.3 TRAVELING SALESMAN PROBLEM

A particularly important computational problem is the traveling salesman problem (TSP). It gets its name from a salesman who lives in a given city but must visit other cities and return home—in what order should he make his calls so that his travel time is minimized? For an entire book that discusses the history and significance of TSP, see Cook (2011).

Version 1 (TSP): Given a weighted graph G with vertices $v_1\ v_2 \ldots v_n$ and edges e_{ij} with a positive integer weight w_{ij} between each pair of vertices v_iv_j, what is the length of the shortest path that starts at v_1, visits every vertex of G, and returns to v_1?

Note that G is assumed to be a complete graph—there is an edge and a weight between any two vertices. It would certainly be possible to allow consideration of other graphs. However, negative weights are not allowed.

Version 2 (Path TSP): Given a weighted graph G with vertices $v_1\ v_2 \ldots v_n$ and edges e_{ij} with a positive integer weight w_{ij} between each pair of vertices v_iv_j, what is the best order of vertices to visit $(v_1\ v_{i2,}\ v_{i3} \ldots v_{i(n-1)}$ that yields the shortest path that starts at v_1, visits every vertex of G,

and returns to v_1? In other words, what is the length of the shortest path, and what is the ordering of vertices that gives this length?

Version 3 (K-TSP or decision problem TSP): Given a weighted graph G with vertices $v_1\ v_2 \ldots v_n$ and edges e_{ij} with positive integer weight w_{ij} on each edge e_{ij}, and a bound K, does G have a path that starts at v_1, visits every vertex of G, and returns to v_1 such that the length of the path is less than K? Or, as a language problem: L_K = {graph G = (V, E, W), integer K: G has an appropriate circuit of weight K, where $|V| \leq |V| \times \min(w_{ij}) \leq K \leq |V| \times \max(w_{ij})$}. (These are the only possible values for K. Why?) This is clearly related to the Hamiltonian circuit decision problem. (Given an unweighted graph G, does it have any path that begins at v_1, visits every vertex of G, and returns to v_1? More on this in Chapter 9.)

Notice that this is a set of decision problems, one for each K. The smallest K such that L_K is a yes instance will yield the shortest solution to TSP. It is also easy to answer TSP given an answer to decision problem TSP, and vice versa. See the Exercises.

This relationship between search (find smallest/largest n such that …) and decision (does n satisfy…?) holds for many problems. But neither the search nor the decision problem has an easy transformation to find whatever makes the question into a yes instance—in this case, the actual shortest TSP path through the graph.

3.4 ALGORITHMS: A FIRST LOOK

One of the main goals of computer science theory is the statement of a good definition of an algorithm. Almost any computer scientist has an intuitive feel for what is meant by the word *algorithm*, but formalizing the definition is not easy. None of the definitions below are really adequate, but they will be good enough for a discussion of the number of steps in a computational process.

Trial Definition I

An **algorithm** is a solution to a computational problem. ■

Thus, BubbleSort, HeapSort, MergeSort, and so on, are all solutions to the Sorting Problem.

Each of these is a finite sequence of steps that provides a solution to every instance of the computational problem, that is, it is a mapping from the set of all inputs to the set of correct solutions (outputs).

The above definition is quite helpful: it tells us what an algorithm is in functional terms. It does not, however, tell us how we arrive at this mapping between input and output. As programmers, we realize that there are many issues surrounding the design, implementation, and execution of a computer algorithm. What logic do we use? What data structures do we need? How do we show that our algorithm is working correctly? There is clearly more attached to the notion of algorithm than this abstract statement.

The word *algorithm* itself is fairly recent in the English language; older dictionaries do not contain it, though they do contain the related word *algorism*.

Trial Definition II (*Merriam-Webster's Collegiate Dictionary*, 11th edition)

An **algorithm** is a "procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; *broadly*, a step-by-step procedure for solving a problem or accomplishing some end, esp. by a computer." ■

Donald Knuth may have had the most influence on the use of the word, and other computer science terms, with his three-volume book on algorithms, *The Art of Computer Programming*, which he began in 1962 (when he was a graduate student); it is most recently available in a third edition (1997).

Trial Definition III (after Knuth, 1968, p. 4)

An **algorithm** is a finite set of rules that gives a sequence of operations for solving a specific type of problem, with five important features:

1. Finite: It must always terminate after a finite number of steps.
2. Definite: Each step is precisely defined; actions to be carried out must be rigorously and unambiguously specified for each case.

3. Input: Zero or more inputs (quantities given to it before the algorithm begins).
4. Output: One or more outputs (quantities with specified relation to inputs).
5. Effective: Operations are sufficiently basic that they can *in principle* be done exactly and in a finite length of time by someone using pencil and paper. ■

Trial Definition IV

An **algorithm** is a well-defined finite sequence of steps, guaranteed to terminate. ■

Definition IV is a reasonable working definition for most programmers. The phrase "well-defined" can be interpreted as "expressible in some programming language"—though after we have the definition of a Turing machine, that will be the preferred definition. "Termination" is the absence of any infinite loop. An algorithm to find an entry in a database will return that entry or announce that it is not there; an algorithm to sort a file will end with the file in proper order.

Some definitions for the word *algorithm* omit the restriction that it must terminate—and there are certainly computer programs, such as the one managing a patient monitoring system in a hospital, that should never terminate. Such an almost algorithm, that may not or does not terminate, will here be called a computation or computational method.

3.5 HISTORY

Although the word *algorithm* only recently began appearing in dictionaries, it has a long history. Algorism/algorithm is derived from the name of Abu Abd Allah Muhammad al-Khwarizmi, a Persian mathematician (from or native of Khwarizm; so the word actually derives from a place name) who, about 810 C.E., wrote an influential treatise on procedures for solving problems in algebra—also a word from the Arabic, *al-jabr*. The confusion between the word *algorism*, which morphed into the word *algorithm*, is likely linked to similar words in English (such as arithmetic and logarithm).

The first standard examples of algorithms are much older than the word itself. The following dates back to Book 7 of Euclid's *Elements*, c. 300 B.C.E. It is a standard example of an algorithm, the Euclidean method for

finding the greatest common divisor of two positive integers (the largest integer that divides both of them).

Euclid's algorithm for GCD (n, m), with n, m positive integers:

```
while m != 0 do
      { r := n mod m
            n := m
            m := r
      }
return n
```

This is fairly obviously a sequence of four simple steps: one test, one step to find a remainder, and two assignment statements. The loop must terminate (for positive inputs) since m decreases during each iteration.

```
n = 60 m = 18  r = 6, n = 18, m = 6
               r = 0, n = 6, m = 0    return 6

n = 55, m = 89 r = 55, n = 89, m = 55
               r = 34, n = 55, m = 34
               r = 21, n = 34, m = 21
               r = 13, n = 21, m = 13
               r = 8, n = 13, m = 8
               r = 5, n = 8, m = 5
               r = 3, n = 5, m = 3
               r = 2, n =3, m = 2
               r = 1, n = 2, m = 1
               r = 0, n = 1, m = 0    return 1
```

These are examples of computations, or the sequence of steps followed when an algorithm is applied to a set of data, that is, to an instance of a computational problem.

3.6 EFFICIENCY IN ALGORITHMS

Definition 3.4

An **efficient** algorithm requires at most polynomially many steps as a function of input size. That is, if the computation returns the value $f(x)$ on input x, where x is a string, then it is efficient if there exists a power K such that the number of computational steps is bounded by $|x|^K$, where $|x|$ is the length (number of characters) of x. ■

Two parts of this definition require immediate comment: Why polynomial? and How to count the size of the input? Also note that some standard functions, such as f(n) = n ln n, are not themselves polynomial, but are bounded by a polynomial (ln n < n, so n ln n < n^2).

3.6.1 Why Polynomial?

It is not truly reasonable to say that all polynomials are small enough to say that an action that takes that many steps should be called efficient. For example, an algorithm that would take $10{,}000n^{100}$ steps to do something to an array of size n would not be generally useful. However, this is the standard definition of tractable/reasonable/efficient for mathematical (and other) justifications that we will go into later.

The main reason for the definition is that problems **not** bounded by a polynomial, whose growth is faster than any polynomial, grow so fast that they are absolutely not efficient. Most of these "superpolynomial" algorithms are exponential, although other functions (such as factorials and functions like $n^{\log_2 n}$) are superpolynomial.

The definitions of *efficient* and *tractable* for algorithms will be revisited in later chapters, with further justifications. For the moment, it will be enough to see that any superpolynomial algorithm is not efficient. Consider how long it would take a computer with a 1-nanosecond instruction cycle to compute the number of instructions given by the first column in Table 3.1.

TABLE 3.1 Graphical Comparison of the Growth of Functions

	n =10	**n = 20**	**n = 50**	**n = 100**	**n = 1000**
n	10^{-8} s	2.0×10^{-8} s	5.0×10^{-8} s	1.0×10^{-7} s	1.0×10^{-6} s
$n \log_2 n$	3.3×10^{-8} s	8.6×10^{-8} s	2.8×10^{-7} s	6.6×10^{-7} s	9.97×10^{-6} s
n^2	10^{-7} s	4.0×10^{-7} s	2.5×10^{-6} s	1.0×10^{-5} s	0.001 s
n^3	10^{-6} s	8.0×10^{-6} s	1.2×10^{-4} s	1.0×10^{-3} s	1 s
n^{10}	10 s	2.84 h	3.1 years	31.7 centuries	3.2×10^{11} centuries
$n^{n\log_2 n}$	2.1×10^{-6} s	4.2×10^{-4} s	3.88 s	5.39 h	2.5×10^{11} centuries
2^n	1.0×10^{-6} s	1.0×10^{-3} s	1.86 weeks	4.0×10^{11} centuries	4.0×10^{282} centuries
3^n	5.9×10^{-5} s	3.49 s	2.28×10^{5} centuries	1.6×10^{29} centuries	4.2×10^{458} centuries
n!	3.6×10^{-3} s	77 years	9.6×10^{47} centuries	3.0×10^{139} centuries	1.3×10^{2549} centuries

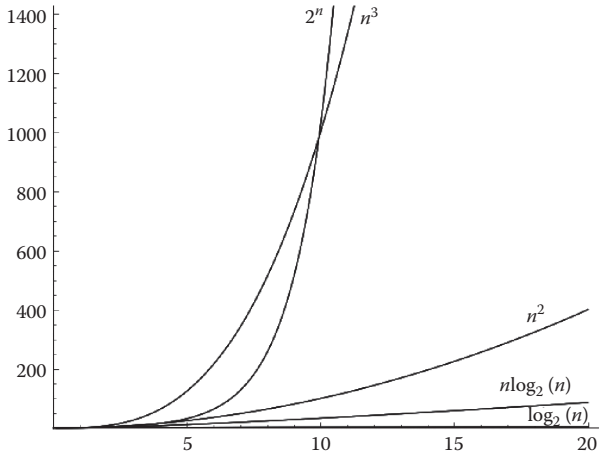


FIGURE 3.1 Growth rates of some functions.

3.6.2 Why Polynomial? Measuring Time

Assume a constant time for each instruction of 1 nanosecond (10^{-9} s); then times required for inputs of various sizes are shown in Figure 3.1 and Table 3.1.

Note that even much faster computers—say, 1000 times faster, with instruction times of 10^{-12} s—do almost nothing to make the unreasonable problems any better. And, this presupposes that such an algorithm even exists. What about those problems with no algorithms? What are they? Are they any worse than one with a running time of centuries?

3.6.3 Size of the Input

Counting the size of the input string x deserves a moment of discussion. This size of the input will be important, and will get further discussion later in the text. In these examples, the input consists of two integers, each with two digits. As input, one cannot write them together (where does one start and the other stop if one writes 6018?), so the length of the input is the number of digits in the first number + the number of digits in the second number + 1 for a separating character. Thus, if the input x is the pair of numbers 60 and 18, it can be written as $x = 60\#18$, and $|x| = 5$. For the pair of numbers 600,132,157 and 18,924, the size of the input would be 18 (or 15 if we drop the three commas).

3.7 COUNTING STEPS IN AN ALGORITHM

Algorithms are analyzed since, if there is a choice between various algorithms, the most efficient would be the best choice. But what does it mean

to analyze an algorithm? And what do we mean by an efficient algorithm? Analysis could mean many things. Efficiency could mean that it runs quickly, or that it does not use much memory, or even that it is easily modified and updated by a programming maintenance team. The standard measure of efficiency is running time, but even here, do we mean best case running time? Average case? Worst case? And on what computer or model?

The standard approach is to count the number of steps (operations) necessary to solve an input I with size $n = |I|$ in the worst case input with that size. Note that in counting the number of steps, it is automatically assumed that all steps require the same amount of time—although division by 7113191 is almost certainly more expensive than multiplication by 2—however, only by a constant factor. And constant factors do not affect rates of growth—it is more important that an (n ln n) algorithm is better than an n^5 algorithm than that a $3n^2$ algorithm might be better than a $5311n^2$ algorithm. In fact, because the $3n^2$ and $6311n^2$ algorithms have the same polynomial order, they will be in some sense considered the same size. Although $6311n^2$ is more than 2000 times as large as $3n^2$, they both depend on the n^2 term for most of their size when n is large. One grows faster than the other, but constantly faster. On the other hand, n^3 grows much faster than n^2. And the function 2^n grows much faster than either.

Thus, we will say that n^3 and $10{,}000n^3$ have the same rate of growth since they differ only by a constant.

3.8 DEFINITIONS

There are many ways to compare the sizes of two functions or two algorithms. We will use only *O* (capital letter omicron from the Greek alphabet, pronounced "big oh"). This is the measure you probably met first in a data structures or algorithms analysis class. But the main reason for using it here is that it is an upper bound—which in this course is what we care about, rather than an exact measure—and it is usually fairly convenient to calculate.

Definition 3.5

For functions f and g defined on the positive integers, f is $O(g)$ if there exist positive integers C and N such that $|f(n)| \leq C\,|g(n)|$ for all $n \geq N$. ■

All functions that are orders of algorithms are positive valued (and nondecreasing), so the absolute value signs are not necessary. For this chapter

and this book, we will always assume that f and g are positive-valued, non-decreasing functions defined on the positive integers.

Definition 3.5 Restated

For positive functions f and g defined on the positive integers, f is O(g) if there exists a positive integer C such that f(n) ≤ C g(n) for all positive integers n. ■

Sometimes it is convenient to consider O(g) to be a set, and then we say f ∈ O(g), instead of f is O(g). Some authors will say O(f) = g, but this is best avoided.

The distinction between f ∈ O(g) and f is O(g) is rarely a problem.

3.9 USEFUL THEOREMS

The following two theorems are often useful for proving that one function is f is O of another. Direct limits are often not useful, since any function that describes the order of an algorithm will be increasing, positive, and tend to infinity.

This is a standard case where applying l'Hospital's rule works well.

Theorem 3.6

If f and g are positive functions, and

$$\lim_{x\to\infty}\frac{f(x)}{g(x)}=C$$

for some positive number C, then f is O(g).

Proof

This is a simple application of the definition of limit. If

$$\lim_{x\to\infty}\frac{f(x)}{g(x)}=C$$

then there is a number N such that

$$n \geq N, \frac{f(n)}{g(n)} \leq C + 1$$

■

Theorem 3.7 (l'Hospital's Rule)

If

$$\lim_{x \to \infty} \frac{f(x)}{g(x)}$$

is an indeterminate limit of the form

$$\frac{\infty}{\infty} \left(\text{or} \quad \frac{0}{0} \right)$$

then

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$$

where f' is the derivative of f with respect to x.

Proof

Any standard calculus text. The name l'Hospital is also written Lhospital or L'Hôpital. In any event, the *s* is not pronounced. ■

Simple application: n ln n is $O(n^2)$. Both (n ln n) and n^2 get very large (tend to infinity) as n gets large—as do essentially all functions that count steps in an algorithm. By using L'Hospital's rule, we have:

$$\lim_{x \to \infty} \frac{n \ln n}{n^2} = \lim_{x \to \infty} \frac{\ln n}{n} = \lim_{x \to \infty} \frac{1/n}{1} = \lim_{x \to \infty} \frac{1}{n} = 0$$

3.10 PROPERTIES OF *O* NOTATION

The following properties are useful; all are easy to prove.

- f is O(f) for all f.
- f is O(1) if and only if it is a bounded function.
- Constant factors can be ignored in any O calculations; that is, Kf(n) is O(f(n)) for any positive constant K.
- $\log_a$ n is $O(\log_b)$ n for any bases a and b, a, b > 1.
- $\log_b$ n is O(n) for any base b, b > 1.
- n^k is $O(b^n)$ for all b > 1, k ≥ 0. Thus n^5 and n^{51} are both $O(2^n)$.
- If r ≤ s, then n^r is $O(n^s)$.
- Any polynomial p(n) is O(the highest-order term in p). (Thus $7n^5 + 11n^2 + 5n + 1111$ is $O(n^5)$.)
- For all k > 0, kf is O(f).
- If f is O(g) and h is O(g), then f + h is O(g).
- If f is O(g) and g is O(h), then f is O(h).
- If f is $O(h_1)$ and g is $O(h_2)$, then f + g is $O(\max(h_1, h_2))$.
- If f is $O(h_1)$ and g is $O(h_2)$, then f·g is $O(h_1 \cdot h_2)$.

3.11 FINDING *O*: ANALYZING AN ALGORITHM

Analyzing O for an algorithm involves making a number of assumptions and simplifications. It can be quite difficult. In this course, we often do not care for the best analysis—an upper bound may be sufficient. Thus, often we do not care if we overcount steps. It is often easy to find an acceptable upper bound.

Two notes: (1) As usual, unless otherwise indicated, the analysis is for the worst case, the largest number of steps on input of size n, and (2) algorithms may be written in C++, Java, any language, or any reasonable pseudocode, with each instruction counted as taking one unit of time.

Each example is stated as a decision problem, that is, a problem with a yes/no solution.

Example: Does a List Contain a Certain Element, the Key?

Algorithm to solve: Find an element key in a list of n elements, if present.

```
Algorithm:  (1) read(key)
            (2) element = first element
            (3) while (element != key) && (element
                <= last)
                   (4) element := next element
```

Analysis: initialization: 2 steps, one read, one assignment moving through the list: 2 tests and 1 move, done 1…n times

$$O = 2+3n = O(n)$$

Note the common simplifications:

All operations assumed to take one step or 1 unit of time.
Worst case counted (best case would be first, average case in center, or …).

Example: Is d the Greatest Common Divisor of n, m Positive Integers?

Algorithm: Find GCD using Euclid's algorithm for GCD (n, m), n, m positive integers.

```
(1) while m != 0 do {
    (2) r := n mod m
    (3) n := m
    (4) m := r
        } // at end of loop, n is the GCD
(5) if (n==d) return true else return false
```

For the length of the input, we need the number of digits required to express an integer n in base 10. This is calculated by letting $x = \log_{10}(n + 1)$, and then taking the ceiling of that number (the next integer above x, or x if x is an integer (so, ceiling 5.21 = 6, ceiling 7.00 = 7).

For example, $n = 1000 \rightarrow \log_{10} 1001 = 3.013 \rightarrow x = 4 \rightarrow 1000$ requires four digits; $n = 563{,}219 \rightarrow x = 6 \rightarrow 563{,}219$ requires six digits.

We generally add the extra character comma for readability, but it is not necessary.

This formula for the length of input of n and m in base 10 becomes $1 + \lceil \log_{10}(n + 1) \rceil + \lceil \log_{10}(m + 1) \rceil$ (where $\lceil\ \rceil$ denotes the ceiling function), or essentially $2\max(\lceil \log_{10} n \rceil, \lceil \log_{10} m \rceil)$. Since the first step reverses n and m if $n < m$ and only adds one extra iteration, we can assume $n \geq m$, and this is $\log_{10} n$; that is, the size of the input is essentially the number of digits in *n*.

If one used binary, octal, or some other base for the input numbers, the algorithm still works. And since the length of an integer in one base is only a constant times its length in some other base, the number of steps only changes by a constant factor.

Division only takes one step on a computer if the numbers are small enough to fit into the register(s) available for division. Here, we want to allow arbitrarily large input numbers; restricting n and m to fit as long integers, or into a standard word size, would be too restrictive. An adaptation of the standard pen-and-paper algorithm for division for dividing m into n takes only $\lceil \log_{10} n \rceil^2$ steps. It is a good exercise to prove this.

In any iteration, $n = km + r$ with $m \geq r$ and $k \geq 1$ (since $n \geq m$). That means $n \geq r + r = 2r$, $r \leq n/2$. Then, n gets m, and m gets r. In every two iterations, $n \leftarrow m \leq (\text{old } n)/2$.So the number of iterations is certainly bounded by $2\lceil \log_{10} n \rceil$. Thus, the order of the algorithm is bounded by a constant multiplied by $\lceil \log_{10} n \rceil^3$, and is polynomial (cubic, though a better analysis would give quadratic) in the size of the input.

Example

Given a Boolean expression e in variables $x_1, \ldots, x_n$, connected by $\wedge$, $\vee$, $\rightarrow$, $\neg$, and grouped with left and right parentheses, with an assignment of 0 (false) or 1 (true) to each of the x_j, is the expression e true?

This is equivalent to evaluating one row of the truth table for the expression e. One approach would be to scan the expression multiple times (certainly less than $|e|$ times). First, replace each x_i with 0 or 1. Second, replace each $\neg 0$ by 1 and $\neg 1$ by 0. Each scan-and-replace can be done in $|e|$ steps, and the procedure is done fewer than $|e|$ times. This means that the entire algorithm is bounded by $|e|^2$.

Another better solution uses two stacks. There is a standard algorithm for evaluating binary (and unary) operations with precedence. Note that $\rightarrow$ is right associative (the other operators are left associative).

Maintain two stacks, one for operators and one for operands. Push left parenthesis and $\wedge$, $\vee$, $\rightarrow$, $\neg$ onto the operator stack; push operands (0 or 1) onto the other. If the current operator is of lower precedence than the one on top of the stack (or same precedence and left associative operation), pop the operands and operator, do operation, and push result.

On a right parenthesis, do all operations back to the matching left parenthesis; pop the "left parenthesis." This algorithm will produce the value of the expression after one pass through the input.

Example

$$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1$$

$$(x_1 \vee x_3 \vee x_5) \wedge (\neg x_2 \rightarrow x_4 \rightarrow x_1) \vee \neg(x_1 \wedge x_4) \wedge x_3 \neg x_1$$

$$(0 \vee 1 \vee 1) \wedge (\neg 0 \rightarrow 0 \rightarrow 0) \vee \neg(0 \wedge 0) \wedge 1 \wedge \neg 0$$

The first four characters are each pushed, which gives us two stacks:

```
operand stack:      0 1
operator stack::          (  ∨
current operator:  ∨  (5th character, same precedence
as top of stack)
  left associative, so pop , 0, 1,  and ∨, evaluate
  and push 0∨1=1
  Push current operator ∨ and 1, current operator
  becomes: )

operand stack:     1  1
operator stack:    (  ∨
current operator:  )
    So evaluate back to the (, pop both stacks, push
    1∨1=1,
    Push the ∧, (,¬ , 0 on appropriate stacks
operand stack:            1  0
```

```
operator stack:            ∧  (   ¬
current operator:  → (lower precedence than ¬)
  Pop 0, ¬: evaluate the  ¬ 0 and push the  1

operand stack:             1  1
operator stack:            ∧   (   →
current operator:   →
  → is right associative; don't evaluate now; push

operand stack:            1  1 0  0
operator stack:           ∧  ( →  →
current operator:  )
  Now we evaluate second →, then first →, popping
  both stacks, pushing results

operand stack:             1  1
operator stack:             ∧
current operator:  ∨  . . . .
  ∨ is lower precedence so . . . The procedure
  continues until the end of the expression, when the
  result will be the only value on the operand stack.
```

Example

Is vertex w reachable from vertex v in a graph G? (That is, is there a path from v to w?) There are also polynomial algorithms for finding the shortest path from v to w, and for finding—in polynomial time—the length of the shortest paths between any two vertices in G.

```
Algorithm: input the graph G = V,E; vertices v and w
    Let Reachable(v) = {v}
    Repeat
      for each vertex v_i in Reachable(v)
      for each vertex v_j in the adjacency list of
      v_i
      if v_j is not in Reachable(v), add it to the
      set until the set Reachable is unchanged
      through an iteration of the loop.
```

Analysis: constant time to initialize all conditions

for each vertex v_i in Reachable(v)//this set has max size $|V|$

for each vertex v_j in the adjacency list of v_i//again, no more than |V|
do something//constant time (but see below)

The size of the loop is thus bounded by $|V|^2$, which is polynomial.

Note: Is this really $O(|V|^2)$? What about the steps in "if v_j is not in Reachable(v), add it to the set"? Can one check that it is not in the set in $O(1)$? Can one add it to the set in $O(1)$? The answer to those questions depends on the implementation of the set Reachable(v). If this is a vector of length |V| of 0's and 1's (0 in vector[j] if v_j is not in the set, 1 if it is), and the vector is implemented in random access memory, then yes to both. If the set is implemented as a linked list, then checking that it is in the list could take up to the length of the list—bounded by |V|, but added to the set it can be done in $O(1)$ if it is added at the head of the list. In either case, the algorithm is still polynomial, though perhaps $O(|V|^3)$.

3.12 BEST AND AVERAGE CASE ANALYSIS

The standard analysis bounds the number of steps taken by any input of size n, so that it is a worst case analysis—if an algorithm is $O(f(n))$, then the algorithm takes no more than $Kf(n)$ steps on any input of size n, including the worst case.

Sometimes it is also useful to speak about the best case, the number of steps required to solve the easiest instance of input of size n, or the average case, the number of steps required averaged over all possible inputs. For example, in an algorithm to find an element in an unordered list, the best case finds it in the first position, the worst case finds it in the last (or not at all), and on average, it should be found approximately in the middle.

The average case can be tricky. For example, consider the case above of finding an element in a list of length n. If the element is in the first position, one test for equality is required; if in the last, n. So, for elements in the last the average number of tests is $(1 + 2 + \ldots + n)/n = (n + 1)/2$. But, what if the element is not in the list—in any such case, n tests are done. So, to do a better average case analysis, it would be necessary to know the probability that the element is found in the list. Here, if it is known that the item is found in the list p fraction of the time ($0 \le p \le 1$), then the average number of tests is $p((n + 1)/2) + (1 - p)n$.

There is another kind of average that would be more useful—that is, for actual inputs to a real-life problem, what is the average running time? For example, in a situation where elements are almost always found, where most recently used elements are moved to the front of the list, and where searches are often for recently used elements, the actual average number of tests for equality done on real inputs might be much less than n/2, and might even be 2 or 3. This kind of analysis is rarely possible by theorists, since it is not usually possible to describe actual average input. However, in some cases, experimental evidence can be quoted to give usual running times for algorithms used in practice.

3.13 TRACTABLE AND INTRACTABLE

The words *tractable* and *intractable* have precise definitions in computer science. These roughly correspond to their meanings in English—if something is tractable, it is well behaved; if something is intractable, it is wild, hard to control, and definitely does not have good manners.

Definition 3.8

A problem is **tractable** (or **feasible**) if it has an algorithm of $O(p(n))$ for some polynomial p(n) in n, the length of the input, that is, there is a constant K so that no input of size n requires more than Kp(n) steps.

The class **P** will be the set of decision problems that have $O(p(n))$ algorithms, for a polynomial p. ■

Definition 3.9

A problem is **intractable** (or **infeasible**) if it has no algorithm of $O(p(n))$ for any polynomial p(n). ■

Note: This means that there are some problems proven to be intractable, but many others that are thought to be intractable, but no proof is known—and someone might put these in the class of tractable problems by coming up with a polynomial algorithm.

Note: Two kinds of problems are in the class of intractable problems under this definition—those with no algorithm at all (the **undecidable**

problems) and those with algorithms, but only algorithms of high order (exponential, factorial, etc.).

What are the reasons behind these two definitions? Several comments are in order.

COMMENTS

1. What about polynomial algorithms with large exponents— perhaps $O(n^{25})$? Is such an algorithm really tractable? Practical? The answer is that we will call this algorithm tractable, although we will admit that we would prefer a faster algorithm.

   By not limiting the exponent in defining tractable, we make life easier for ourselves. We do not have to worry about exactly which exponent is in the polynomial. And, we can combine algorithms in many ways and remain tractable (since the sum, product, and composition of two polynomials are polynomial).

   It is also true that there are very few algorithms known that have large exponents. And, many problems with higher-order exponents are replaced by better and lower-order algorithms—for example, multiplying $n \times n$ matrices has a simple algorithm of n^3, but better algorithms of order $n^2 \ln n$ are known, and there is hope for even lower orders.

   For many computation problems, distributed computation can be used to "parallelize" an algorithm so that each component machine does a small part of the overall work, thus decreasing the overall bound to a lower polynomial; however, an exponential algorithm will always remain exponential.

2. What about intractable problems? Won't they become tractable when someone finds a good algorithm? For some problems, maybe yes. For example, for a long time, linear programming had no known polynomial algorithm. Also, the problem PRIME = {p | p is a prime integer} was only recently found to have a polynomial algorithm.

   Most problems now called intractable are known to be in the class of **NP**-complete problems—defined and discussed in the next several chapters. No problem in this class can have a polynomial algorithm

unless every other problem in the class also has one. The class is large—hundreds of known problems. No polynomial algorithm is known for any of these problems. It is widely accepted that (a) no polynomial algorithm will ever be found for any of them, but also that (b) it will never be proved that polynomial algorithms do not exist.

3. This definition of tractable has good closure properties—more restrictive definitions have problems. We would like, for example, that if f and g are in the set, so is f + g, f*g, and f·g. As we will see in the next chapter, that is true under this definition.

EXERCISES

All logs are base 2 unless otherwise noted.

1. Find three (or more) definitions of *algorithm*. Compare them. Do they say the same thing? If not, how do they differ? Which is easiest to understand?
2. Consider these steps as part of a possible algorithm. Are they allowed? Why or why not?
   a. Heat 2 quarts of water until it boils.
   b. Add 1 C sugar and 1 C butter to bowl; beat until smooth and creamy.
   c. Every 10,000 miles, rotate tires (exchange left front with right rear, right front with left rear).
   d. Add the integers from 1 to 100.
   e. Read and understand chapter 1 of Knuth, *Art of Computer Programming*, volume 1, *Fundamental Algorithms*.
3. How could the following be translated into a decision problem?
   a. Find the quotient of 447 divided by 3.
   b. Find all the prime divisors of 447.
   c. Find the cheapest airfare available on next May 5 from Oakland, California, to Chicago, Illinois.
4. Find the simplest polynomial function f so that each of the following is $O(f)$:

a. $n + 87\sqrt{n}$

b. $\log (n^3)$

c. $(27n + 5)(33n^2 + 21n)$

d. $(\log n)^2$

e. $n \log n + 2.7n^{1.8} + 2n\sqrt{n}$

f. $21n + 51n^2 + n^3$

g. $\log n + \sqrt{n}$

h. $(\log n)^2 + \log (n^2) + \sqrt{n}$

5. How many multiplications does it take to evaluate n^m, for integers n and m? (There are several possible correct answers.)

6.

a. Given an answer to TSP, show how to answer K-TSP in one step.

b. Given an answer to K-TSP for all legal K, show how to answer TSP. Make your answer as simple (fewest number of steps) as possible. If you use a loop, what is the order of your loop?

7. Consider the polynomial $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$.

a. How many multiplications and additions does it take to evaluate it in the form given?

b. How many multiplications and divisions if it is rewritten (Horner's method) as $f(x) = ((a_nx + a_{n-1})x + \ldots + a_1)x + a_0$?

8. For which of the following functions is it true that f(2n) is in $O(f(n))$?

a. $f(n) = 11n$

b. $f(n) = n^3$

c. $f(n) = \log n$

d. $f(n) = n \log n$

e. $f(n) = 2n$

9. You are given the choice of two algorithms: one is order $247{,}853n^{17}$, and the other is order $1.23(2^{1.2n})$. Which do you prefer? Why?

10. Give an algorithm that determines whether or not a given undirected graph G = (V, E) contains a cycle. (A cycle is a path from a vertex back to itself.) What is the order of your algorithm? Can you find an $O(|V|)$ algorithm, that is, linear in the number of vertices and independent of the number of edges?

11. Find the O relationships between:

    a. $f(n) = \log n$ and $g(n) = \log \log n$

    b. $f(n) = \log_2 n$ and $k(n) = \log_3 n$

    c. $f(n) = \log n$ and $g(n) = \log (n^2)$

    d. $g(n) = \log \log n$ and $h(n) = (\log n)^2$

    e. $f(n) = 2^n$ and $g(n) = n!$

12.

    a. Write an algorithm in pseudocode to find the average of n integers.

    b. What is the order of this algorithm if the integers are bounded in size?

    c. What is the order if the integers can be arbitrarily large?

13. Consider two algorithms for division of integers, n divided by m: (a) repeated subtraction and (b) the one you learned for long division in elementary school. Write out both algorithms in pseudocode (using numbers in base 10 or base 2, whichever you prefer) and analyze their orders.

14. Write (pseudocode) algorithms for each of the following, and give their order:

    a. Add an element to the beginning of a linked list of length l.

    b. Add an element to the end of a linked list of length l.

    c. Add an element to the correct location of a sorted linked list of length l.

    d. Multiply two $n \times n$ matrices.

    e. Find the first occurrence of the substring aa in a string of length n.

15. A connected component of the undirected graph G with vertices V and edges E is a set of vertices such that any vertex in the component is connected to all other vertices in the same component. Assume an adjacency list representation for G. Write an algorithm to find the number of connected components in G. What is its order (in terms of |V| and |E|)?

16. The following formula is useful in algorithms analysis:

$$\sum_{i=1}^{n} i = n(n+1)/2$$

Use this, if necessary, to analyze the order of these code fragments:

a.
```
for (i = 1; i<n; i++)
    for (j = 1; j < 6; j++)
        if (a[i,j] == 0) ZeroCount++;
        else if (a[i.j] < 0) NegativeCount++;
        else PositiveCount++;
```

b.
```
for (i = 1; i<5; i++)
    for (j = 1; j < 6; j++)
        if (a[i,j] == 0) ZeroCount++;
        else if (a[i.j] < 0) NegativeCount++;
        else PositiveCount++;
```

c.
```
for (i = 1; i<n; i++)
    for (j = 1; j < n; j++)
        if (a[i,j] == 0) ZeroCount++;
        else if (a[i.j] < 0) NegativeCount++;
        else PositiveCount++;
```

d.
```
for (i = 1; i<n; i++)
    for (j = 1; j < i; j++)
```

```
if (a[i,j] == 0) ZeroCount++;
else if (a[i.j] < 0) NegativeCount++;
else PositiveCount++;
```

CHAPTER 4

Turing Machines

4.1 OVERVIEW

Theoretical computer science needs a model of a computer—a perfect machine, not dependent on electricity, chip design, or anything that might have a flaw. This model could run programs in a high-level language such as C++, Java, or Python—but usually will be much simpler than that. What is needed is a simple machine such that if a problem can be solved on this machine, then it can be solved on a PC, a supercomputer, or whatever—and vice versa. Moreover, one would also want that a solution "in reasonable time" on a standard computer would be possible if and only if (iff) it could be done in reasonable time on the model machine.

There have been several different models proposed—some very closely related, some that do not look like machines at all, but rather are similar to an abstract branch of mathematics. The most useful and widely used model is the one proposed by the English mathematician and code breaker Alan Turing.

The purpose of this chapter is the exploration of this model (in several different flavors), and to explore the idea that some problems are not solvable at all on any variety of computer.

4.2 THE TURING MACHINE MODEL

Informally, a Turing machine is a device with a very, very large memory (so large that no problem can ever exhaust it) and a means to read from and write to that memory—that is, to act on its storage locations in some very precisely defined manner.

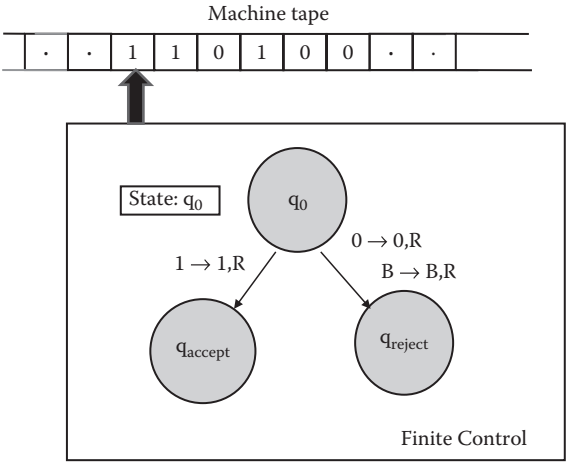


FIGURE 4.1 Sample Turing machine.

For decision problems, the only question is yes/no, whether or not a certain string is in a language or not. The job of the Turing machine is simply to take an input string and determine whether the answer is yes or no.

It does this by following a scheme where it looks at memory locations, one at a time, possibly changing the contents, and moving to the next location either forward or back (right or left). Its set of instructions tells it what to do when reading a certain symbol on its tape.

We think of a Turing machine, then, as an infinitely long tape containing symbols, with a read/write head sitting over one of these symbols. This read/write head is connected to the Turing machine control, which keeps track of which state it is in and gives instructions to the read/write head. A diagram of a sample Turing machine showing the tape, the read/write head, and the control is shown in Figure 4.1.

4.3 FORMAL DEFINITION OF TURING MACHINE

Definition 4.1

A Turing machine $M = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_{start}, \delta)$ in which:

- Σ is the **input alphabet** with blank $\mathbf{B} \notin \Sigma$.
- Γ is the **tape alphabet** with $\mathbf{B} \in \Gamma$ and $\Sigma \subset \Gamma$.
- **Q** is a set of finite **states**.
- $\mathbf{q_{accept}} \in \mathbf{Q}$ is the unique **accept state**.

- $\mathbf{q_{reject}} \in \mathbf{Q}$ is the unique **reject state**.
- $\mathbf{q_{start}} \in \mathbf{Q}$ is the specially designated **start state**.
- δ: $\mathbf{Q} - \{q_{accept}, q_{reject}\} \times \Gamma \rightarrow \mathbf{Q} \times \Gamma \times \{\text{left, right}\}$ is the **transition function**. ■

Turing machines are often written as diagrams. For instance, one of the simplest possible Turing machines is shown in Figure 4.1, with $\Sigma = \{0, 1\}$, $\Gamma = \{B, 0, 1\}$, $Q = \{q_0, q_{accept}, q_{reject}\}$, and δ designated by the arrows. Note that since δ is a function, there are three arrows out of each state (one for each element in Γ), although there are never any arrows allowed out of q_{accept} or q_{reject}.

Let's examine each element of this formal definition in greater detail.

4.3.1 Input Alphabet

The input alphabet Σ contains all the symbols that can be used as input to a particular Turing machine. As each decision problem specifies its input differently, this alphabet changes depending upon the purpose of the machine. A machine to decide PRIME (given an integer, is it a PRIME number?), for instance, would need an alphabet to represent the symbols of its integer input, whereas one to decide the shortest path would need an alphabet to represent a weighted graph (nodes, edges, distances, etc.).

The Turing machine begins with its input written as a string over Σ onto an otherwise empty tape (and the head on the first symbol of the string, if there is one). The unused portion of the tape contains blanks (denoted by the marker **B**). For various reasons (such as finding left and right ends of the string), we restrict the input string by requiring that it not contain this **B** marker. (Side note: If you want to input something that naturally contains blanks—such as the text of a short story—then first replace every blank in the input with something else, such as underscore or # or ~ or whatever.)

Σ is an alphabet, and thus by definition it is a finite set.

4.3.2 Tape Alphabet

The tape alphabet Γ contains all the symbols that could ever be written onto its tape. As with Σ, this alphabet varies depending upon the nature of the problem.

Since the input string is initially written onto the tape, Γ must include every symbol of Σ. Similarly, since the infinite tape initially has the blank **B** written in every noninput location, Γ must also include this symbol.

That is, we require at least (1) $\Sigma \subset \Gamma$ and (2) $\mathbf{B} \in \Gamma$, but $\mathbf{B} \notin \Sigma$.

As with Σ, the tape alphabet Γ must be a finite set.

4.3.3 Set of States

The state set Q contains all the control states of the Turing machine. As the Turing machine processes its input, it moves from state to state to indicate different steps of its problem-solving process. These states can be named anything, but by convention we will name them by subscripting the letter q (such as q_0, q_1, q_3, etc.). Sometimes it makes a Turing machine more readable to give them meaningful names, such as $q_{\text{seen-first-one}}$ or q_{even}.

4.3.4 Accept State

The accept state q_{accept} is used by the Turing machine to indicate that its input string is a member of the language in question. The Turing machine does not return this result as a yes or an accept, nor does it write any special symbol onto its tape. Instead, it simply enters the state q_{accept} and stops its operation (halts).

The accept state is unique and every Turing machine *must* contain q_{accept} in its state set.

4.3.5 Reject State

The reject state q_{reject} is in all respects the counterpart of q_{accept}. The Turing machine enters this state once it has determined that the input string is not a member of its language, and then stops processing.

As with the accept state, q_{reject} must be a unique element of a Turing machine's state set. This means that q_{accept} and q_{reject} are distinct, as the Turing machine cannot answer the same question with both a yes and a no answer.

4.3.6 Start State

The start state q_{start} is the state from which the Turing machine begins its execution. There is only one start state. It is not necessarily unique like q_{accept} or q_{reject}. This means that we could allow either q_{accept} or q_{reject} to be the start state, but that leads to a very uninteresting Turing machine: one

that either accepts or rejects all strings automatically. Thus, although the formal definition requires that a Turing machine have at least two states, we will assume that each machine has at least three.

For simplicity, we will follow the convention that q_0 is always the start state. With this assumption, we will not need to designate a special q_{start} for each new machine.

4.3.7 Transition Function

The transition function (or **delta function**) δ controls the operation of the Turing machine. At each step of its execution, a Turing machine examines both the current symbol under its read/write head and its current state. It then consults its delta function in order to determine how to proceed. The machine replaces the symbol beneath its read/write head with a new symbol, changes its state to a new state, and moves the read/write head either to the left or to the right one tape cell, all as directed by its transition function.

Since δ is a function, it must be defined for *all* possible state and symbol pairs—except of course, the two states where moves are not allowed. Since the Turing machine stops execution once it has entered either q_{accept} or q_{reject}, it need not—in fact, must not—make a move from either of these two states. The transition function therefore takes as its domain $Q - \{q_{accept}, q_{reject}\} \times \Gamma$. Its range is quite naturally $Q \times \Gamma \times \{\text{left}, \text{right}\}$, where left and right are the directions to move the read/write head either left or right, respectively.

Several things should be noted here:

- A Turing machine *must* write a symbol to its tape at every step. However, it can always replace the current symbol under the read/write head with exactly the same symbol that was there.
- A Turing machine must go to a new state on each move, but that state can be the same state that it was in previously.
- A Turing machine *must* move its read/write head either to the left or to the right at every step. It cannot stay positioned over the same symbol, although it can certainly move in one direction and then move back again in the opposite direction in order to regain its original position.

- The Turing machine can operate forever, and since it has an infinite amount of memory available, it cannot ever crash. The only way for the machine to stop execution is by entering either its accept state or its reject state.

It would generally be tedious to write out the entire transition function, as machines may contain large (although still finite) state sets and tape alphabets. We will use the convention here (called the reject convention) that if a move is not specified in the delta function, it is assumed to be

$$\delta(q_i, \sigma) = (q_{reject}, \sigma, \text{right})$$

There is, of course, no reason to move right (or left), but the read/write head has to move in some direction.

For the sake of clarity, it is generally assumed that Q and Γ are disjoint (that is, $Q \cap \Gamma = \varnothing$), and that both left $\notin Q \cup \Gamma$ and right $\notin Q \cup \Gamma$. It would be very confusing if different elements of the Turing machine shared the same name (such as a tape alphabet symbol called q_{accept}). There is nothing, however, in the above formal description to prevent this.

The Turing machine begins its operation with its read/write head positioned over the leftmost symbol of the input string. As all strings are finite (even though the tape itself is not), the input always possesses discernable left and right ends. If the input is the empty string, the read/write head will be over a **B** symbol at the start of computation.

4.4 CONFIGURATIONS OF TURING MACHINES

Definition 4.2

A Turing machine **configuration** is a *complete* description of the machine at some step of its operation. It must specify three things:

1. The current state of the machine
2. The entire nonblank contents of the tape
3. The location of the read/write head ■

There are several standard ways to write a machine configuration. The one used here will be to list the state followed by the tape contents, with the current location of the read/write head underlined.

Example

Let $\Gamma = \{B, 0, 1, 0', 1'\}$, the current state be q_{273}, 0011′010′11 be the nonblank portion of the tape, and the read/write head be positioned over the second nonblank symbol.

The configuration is written as q_{273}; 0$\underline{0}$11′010′11.

An alternate representation (often found in other texts) that is shorter (but perhaps less readable) is to encode the current state directly into the tape contents. This style of representation shows where the read/write head is located by placing the current state immediately in front of the symbol beneath it.

The configuration is sometimes written as 0q_{273} 011′010′11.

This notation (with the state embedded into the string) does have one advantage—it does not require the extra underline to show the location of the head. It also is more economical; the entire configuration is encoded in one string.

If the read/write head is positioned over a blank to the left or to the right of all the nonblank symbols on the tape, that blank must be shown:

Let $\Gamma = \{B, 0, 1, 0', 1'\}$
the current state be q_{273}
0011′010′11 be the nonblank portion of the tape
the read/write head be positioned two positions to the left of the nonblank portion of tape

q_{273}; $\underline{B}$B0011′010′11 or q_{273}BB0011′010′11

(underline notation) (embedded notation)

One more point: It is always assumed that the machine starts in a proper configuration—so there is no need to check that the read/write head is actually over the first symbol, that there are no blanks embedded in the input string, that the symbols on the tape all come from Γ, and so forth.

4.5 TERMINOLOGY

A few basic definitions are needed to describe the behavior of a Turing machine on a particular input:

Definition 4.3

A Turing machine M **accepts** an input string $w \in \Sigma^*$ iff it enters q_{accept}. ■

Definition 4.4

A Turing machine M **rejects** an input string $w \in \Sigma^*$ iff it enters q_{reject}. ■

Definition 4.5

A Turing machine M **halts** an input string $w \in \Sigma^*$ iff it enters either q_{accept} or q_{reject}. ■

Definition 4.6

A Turing machine M **loops** an input string w iff it *never* enters either q_{accept} or q_{reject}. ■

We also need some terminology to describe the behavior of Turing machines on entire languages.

Definition 4.7

A Turing machine M **decides** language L over Σ iff for any string w in Σ^*, M enters q_{accept} if w is in L, and enters q_{reject} if w is not in L. The language L is **decidable** or **recursive**. ■

Definition 4.8

A Turing machine M **accepts** (or **recognizes**) language L over Σ if for any string w in Σ^*, M enters q_{accept} if and only if w is in L. The language L is **recursively enumerable** (RE) or **semidecidable** or **acceptable**. ■

It is important to carefully read and fully understand the above definitions. At a superficial glance, they may look the same. However, the definition of *accepting* does not require the Turing machine to enter q_{reject} for strings not in L. It only forbids that it enter q_{accept}. It can do this by entering q_{reject}, by moving right forever, by alternating between two adjacent tape squares, or by some other set of moves.

And of course, if a Turing machine decides a language L, then it accepts L. If it accepts L, it might also decide L, but would not necessarily do so. "Deciding" is a stronger condition than "accepting."

A word on notation: Various authors will use different words for these classes. The following groups cover most of the variations you might meet:

decidable == recursive == Turing decidable

acceptable == recognizable == semidecidable == Turing acceptable == recursively enumerable == RE

The terms *recursive* and *recursively enumerable* come from mathematical function theory. However, since we are not dealing with recursion as most computer science students know the term, we will use the words *decidable* and *acceptable*.

4.6 SOME SAMPLE TURING MACHINES

For simple languages, it is easy to describe a Turing machine to decide that language, at least in informal English. But even for simple languages, writing out the entire set of states and the entire transition function can be very tedious—and is something that we will rarely do.

Example M_1 (Figures 4.2 and 4.3)

$$L_1 = \{w\text{: w begins with 0}\} \text{ over } \Sigma = \{0, 1\}$$

$$\mathbf{M_1} = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_0, \delta), \text{ in which:}$$

$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, \mathbf{B}\}$
$Q = \{q_{accept}, q_{reject}, q_0\}$
$\delta =$
$(q_0, 0) \rightarrow \{q_{accept}, 0, R\}$

$(q_0, 1) \rightarrow \{q_{reject}, 1, R\}$
$(q_0, \mathbf{B}) \rightarrow \{q_{reject}, \mathbf{B}, R\}$

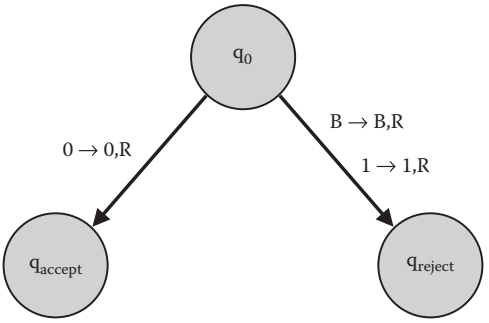
Figures 4.2 and 4.3 both show machine M_1, using—or not using—the reject convention that all arrows not shown go to q_{reject}.

Example M_2 (Figure 4.4)

$$L_2 = \{w\text{: w does not begin with 0}\} \text{ over } \Sigma = \{0, 1\}$$

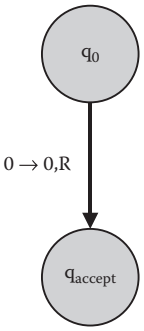
$$\mathbf{M_2} = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_0, \delta), \text{ in which:}$$

$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, \mathbf{B}\}$



M_1 ACTION: If first symbol is "0," accept; else, reject

FIGURE 4.2 Diagram for machine M_1.



M_1 ACTION: If first symbol is "0," accept; else, reject

FIGURE 4.3 Diagram for machine M_1 using reject convention.

$Q = \{q_{accept}, q_{reject}, q_0\}$
$\delta =$
$(q_0, 0) \rightarrow \{q_{reject}, 0, R\}$
$(q_0, 1) \rightarrow \{q_{accept}, 1, R\}$
$(q_0, \mathbf{B}) \rightarrow \{q_{accept}, \mathbf{B}, R\}$

Example M_3 (Figure 4.5)

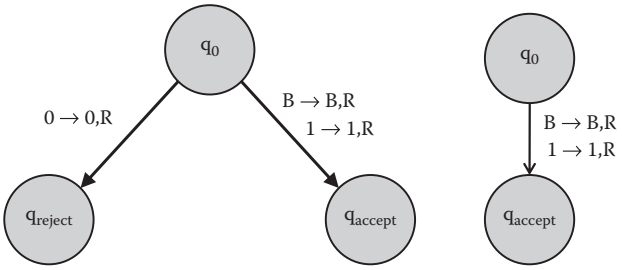
$L_3 = \{w: w \text{ ends with } 1\}$ over $\Sigma = \{0, 1\}$

Example M_4 (Figure 4.6)

$L_4 = \{w: w \text{ ends with } 1\}$ over $\Sigma = \{0, 1\}$

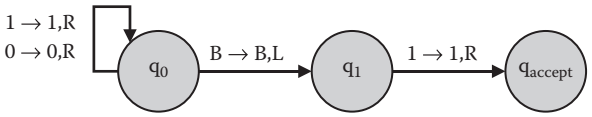
Example M_5 (Figure 4.7)

Other machines are possible for the language of strings that end in 1 (or for any other Turing machine language). The Turing machine for a language is never unique. Trivially, one can modify any machine by



M_2 ACTION: Look at first character
If not 0, accept else reject

FIGURE 4.4 Two versions of machine M_2.



M_3 ACTION:
Scan across w to find a blank
Move left one cell
If last symbol is a "1," accept; otherwise, reject

FIGURE 4.5 Diagram for machine M_3.

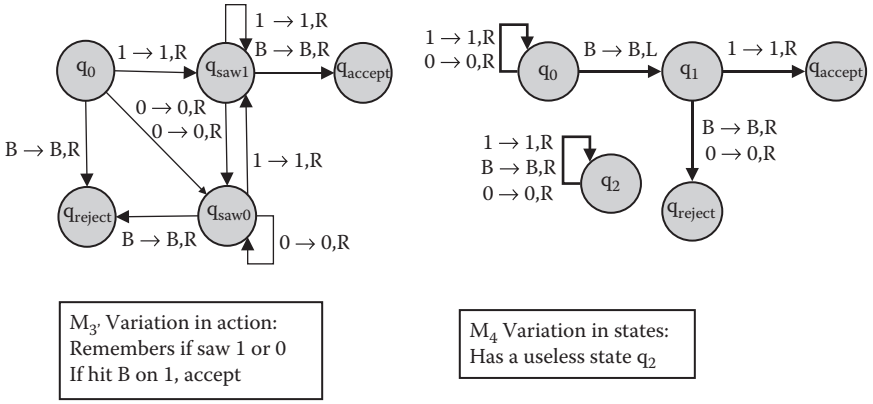


FIGURE 4.6 Two other Turing machines for M_3.

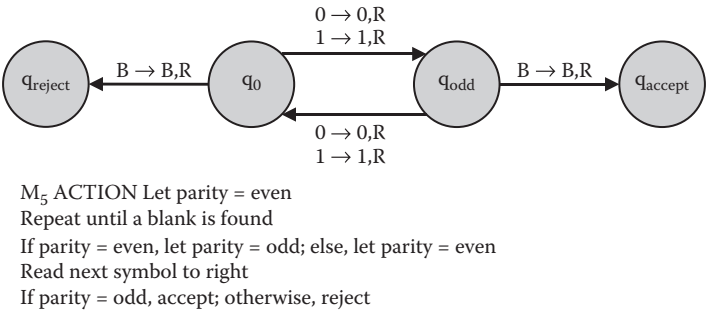


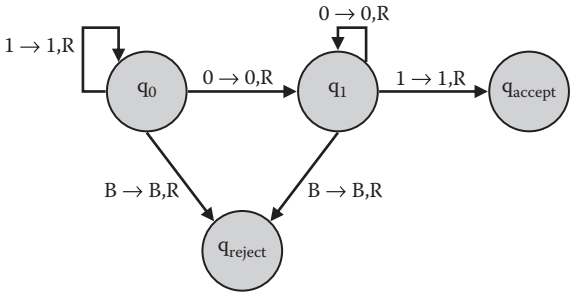
FIGURE 4.7 Diagram for machine M_5.

adding one or more states but with no moves to these states (so they are unreachable). And since any number of new unreachable states could be added, there are an infinite number of Turing machines that accept the same language. They appear different superficially, though actually all act on a string in the same way.

Nontrivially, there are often multiple ways to approach a problem. For example, another way of accepting strings whose last symbol is 1 is to use the state to remember the last symbol read, and accept if the machine hits a blank when the last symbol read was 1.

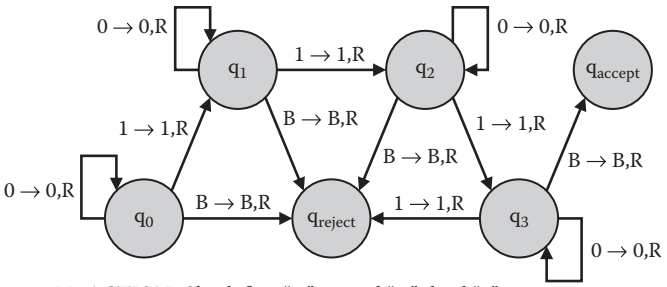
Example M_6 (Figure 4.8)

$$L_6 = \{w\text{: w contains the substring 01}\} \text{ over } \Sigma = \{0, 1\}.$$



M_6 ACTION:
Look for "0," then "1"; if found, accept

FIGURE 4.8 Diagram for machine M_6.



M_7 ACTION: Check first "1," second "1," third "1"
If hit end of string, accept; else reject

FIGURE 4.9 Diagram for machine M_7.

In this machine, note that it stops and accepts as soon as it finds the substring 01; it does not need to read the entire string.

Example M_7 (Figure 4.9)

L_7 = {w: w contains exactly three 1's} over $\Sigma = \{0, 1\}$

Example M_8 (Figure 4.10)

$L_8 = \{w: w = 0^N 1^N \mid N \geq 0\}$ over $\Sigma = \{0, 1\}$

Example M_9 (Figure 4.11)

$L_9 = \{w: w = xx^R\}$ over $\Sigma = \{0, 1\}$

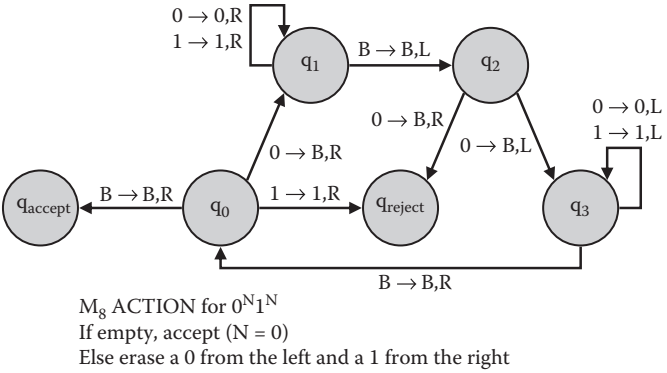


FIGURE 4.10 Diagram for machine M_8.

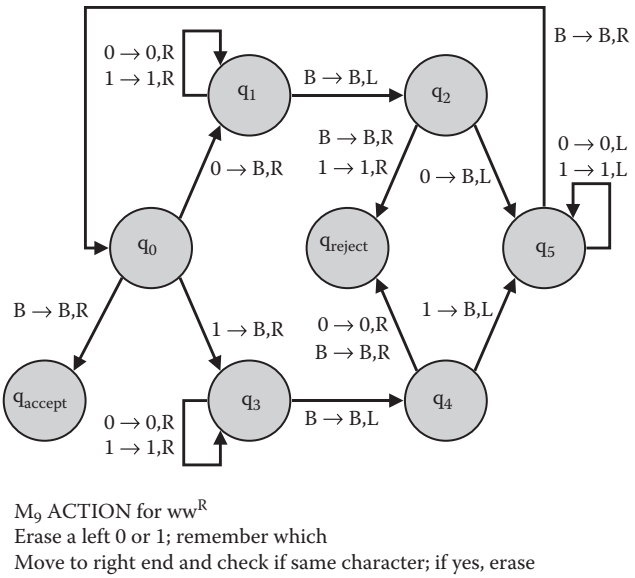


FIGURE 4.11 Diagram for machine M_9.

4.7 TURING MACHINES: WHAT SHOULD I BE ABLE TO DO?

4.7.1 Decide If a Given Diagram Describes a Turing Machine

Suppose you are given a picture such as the diagrams for M_1, M_2, M_3, and so forth, above. Does this diagram really describe a Turing machine? Maybe yes, maybe no. Ask yourself:

- Does it have a start state? An accept state? A reject state?

- Is there exactly one move out of every state on every symbol? (Check the arrows.)
- Are there indeed no moves out of q_{accept} and q_{reject}?

4.7.2 Given a Turing Machine, Trace Its Operation on a Given String

This should be easy. Begin in the start state reading the first symbol of the input string (or reading a blank B if the input string is empty). Trace the sequence of configurations of the Turing machine. At each step, show the (1) state it is in, (2) position of the head, and (3) entire nonblank contents of the tape. You can draw a picture of each configuration, or use the convention of the state followed by the string (with position of head underlined), or the other convention with the state embedded in the string.

4.7.3 Given a Turing Machine, Describe the Language It Accepts

This can be easy or hard or essentially impossible—a lot depends on the complexity of the Turing machine. The tape alphabet should be obvious (just look at the labels on the arrows). Try a few sample strings, running them through the machine. Then, see if you can figure out the logic behind the states and arrows.

4.7.4 Given a Language, Write a Turing Machine That Decides (or Accepts) This Language

The authors believe that each student should write out a Turing machine—or two or three or four—in order to understand their basic operation. This is often tedious, because the operations allowed are very limited.

We recommend beginning with an English description of how the Turing machine will operate—keep this with your final machine. Think of it as comments that allow the reader to understand some very hard-to-read code (i.e., the Turing machine). Then write the Turing machine diagram. Finally, run some strings through the Turing machine to be sure it is operating in the way you expect (that is, test your Turing machine).

For example, consider writing a machine to accept strings of odd length over {0, 1} whose center character is a 1. In English: (1) find the center character (if no center character, string is even length and reject), and (2) accept if this center character is 1.

However, "find the center character" is not so easy. Probably the best way is to cross out (change to blank) one character from the left end of the string, and one character from the right end. But, if the string was of

length 1, you just blanked out the center character and have lost it—you do not know if it was 1 or not.

In modified English:

1. If the string is empty, reject.
2. Check if the string is length 1 (move right one square; if blank, length was 1).

   If length was 1, move back to the single character and accept if it is 1.

3. If length > 1, move to left end and blank out one character; move to right end and blank out another.
4. Repeat steps 1–3 until accept or reject.

EXERCISES

1. Given the Turing machine M_5 (from the text),
   a. Trace its operation (give the sequence of configurations) on the input string 01.
   b. Repeat (a) with a different M_i or with a different string (such as 00, 0011, …).
2. Write out the diagram for a Turing machine that accepts strings of odd length over {0, 1} with center character equal to 1. Trace the operation of your Turing machine on 1, 011, and 001.
3. Design a Turing machine to accept strings over {0, 1} that are palindromes, that is, $w = w^R$. (Hint: Modify the machine that accepts ww^R (even-length palindromes) so that it also accepts 0, 1, 000, 010, etc.)
4. Write out all the moves (diagram) for a standard one-tape Turing machine to accept the language (over $\Sigma = \{a.b\}$) of strings:
   a. Begin with a and end with b.
   b. With an even number of a's.
   c. Length divisible by 3, $|w| = 3k$, and characters in positions k, 2k, and 3k are b's (ababab, etc.).

d. The number of b's is equal to the number of a's.

e. Of the form a^N, where N is a composite (nonprime) number.

f. Have the form www (such as aaabaaabaaab).

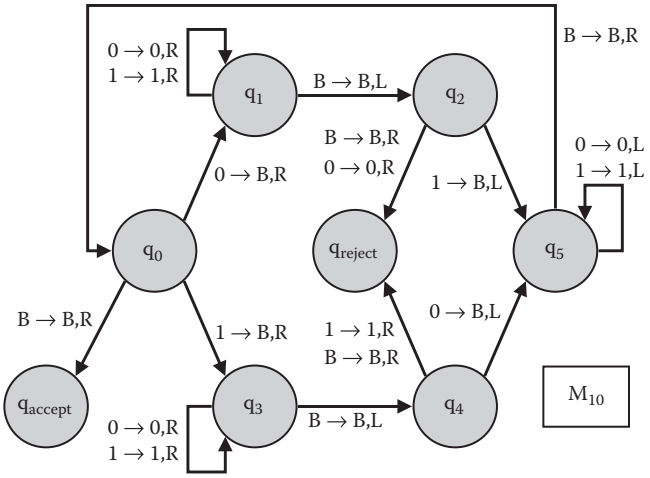
(Note: All of these can be done on a standard Turing machine. Some problems, such as (e) and (f), take careful construction of a complex machine.)

5. Given the diagrams below (Figure 4.12), answer:

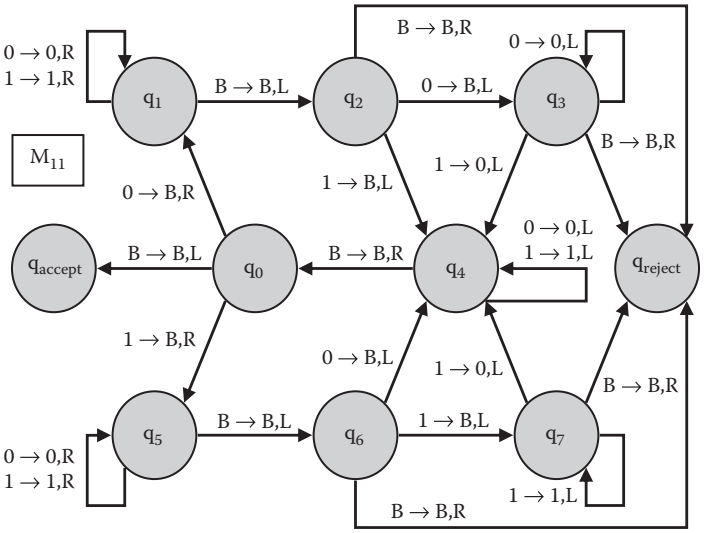
a. Does the diagram describe a legal Turing machine?

b. Trace the operation of the Turing machine on all strings of length 2.

c. Describe the language accepted by this Turing machine.



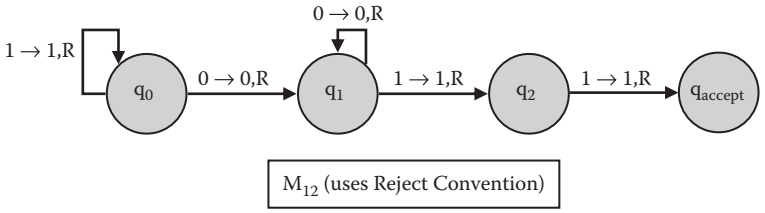
(a)



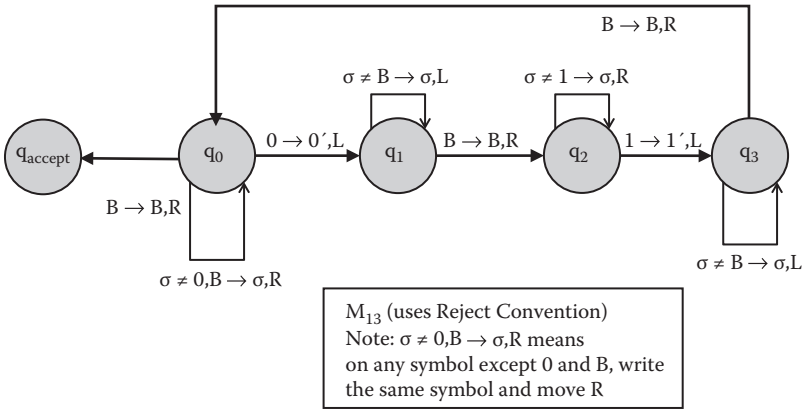
(b)

FIGURE 4.12 Problems: (a) Diagram for possible machine M_{10}. (b) Diagram for possible machine M_{11}.

Continued



(c)



(d)

FIGURE 4.12 (*Continued*) Problems: (c) Diagram for possible machine M_{12}. (d) Diagram for possible machine M_{13}.

CHAPTER 5

Turing-Completeness

5.1 OTHER VERSIONS OF TURING MACHINES

The Turing machine is an extremely **robust** model of computation. This means that many changes can be made to our basic definition of a Turing machine without changing its power as a computing device. In other words, we can "tweak" a Turing machine in various ways, and still be left with a Turing machine—or more precisely, with something that has exactly the same power to solve problems as the standard Turing machine model.

Definition 5.1

A model of computation is **Turing-complete** if it can be shown that it is equivalent in power to a Turing machine. ■

That is, computational model M_1 is equivalent to another computational model M_2 if and only if:

1. Any computation performed by M_1 can also be computed by M_2.
2. Any computation performed by M_2 can be computed by M_1.

Thus to prove two models equivalent, one proves 1 and 2. Proving only one of these points does not show equivalency. A supercomputer can accomplish any task that an abacus can, but this does not mean that they are equivalent computing devices. On the other hand, if an inexpensive personal computer can be shown to be able to perform each of the machine

instructions for a supercomputer, then it can emulate any program on the supercomputer—perhaps taking much longer, but time and efficiency are not part of the definition of equivalent computing power.

Several different changes are proposed to the Turing machine model below. All of these modifications are shown to be Turing-complete.

5.1.1 Semi-Infinite Tape

Many authors use a model of a Turing machine that has a semi-infinite (or one-way infinite) tape—the tape has a left end, but no right end. The machine starts in q_{start} with the input placed on the leftmost portion of its tape and with the read/write head positioned on the first tape location.

The disadvantage of this model is that one must have some convention to deal with a transition function that asks for a move left when the read/write head is on the left end of the tape. Various conventions will work—that the head falls off the left end of the tape and enters a hanging state from which it can never recover, that a move left when on the left end is implemented as remaining on the left end so that the head does not move, or other possibilities. For various formal definitions of semi-infinite tape Turing machines, see Hopcroft et al. (2007), Lewis and Papadimitriou (1997), Sipser (2006), or others.

Having a left end to the tape also makes it awkward to return to the left end of the input. Finding the right end is easy—simply move right to the first blank, and then move to the left one position. Finding the left end is similarly easy if the tape is two-way infinite; there is always a blank to the left of the input, but it is much more difficult with a semi-infinite tape.

The two-way tape seems to allow simpler constructions and simpler combinations of moves. However, it does not add additional power.

Theorem 5.2

Any standard Turing machine M with two-way infinite tape can be emulated by a Turing machine M′ with semi-infinite tape, and vice versa. Thus, the semi-infinite tape model of a Turing machine is Turing-complete.

Proof

There are many ways to put a two-way infinite tape onto a semi-infinite tape. One way is to make a machine M′ by using a special symbol of the

tape alphabet—perhaps ⊢, called a turnstile—in cell 1 as an end marker. The even positions then hold the right side of the two-way infinite tape; the other odd positions hold anything that would be written on the left.

For example, a standard two-way infinite tape Turing machine could begin with tape contents:

$$\ldots \mathbf{BBB}\sigma_1\sigma_2\sigma_3 \ldots \sigma_n, \mathbf{BBB} \ldots$$

It is natural to take the start of the input as the first character after the end marker ⊢—in essence, "folding" the two-way infinite tape there. But any other position on the tape would also work.

If one takes the natural beginning, initialize the tape of M′ with

$$\vdash\sigma_1\mathbf{B}\sigma_2\mathbf{B}\sigma_3\mathbf{B} \ldots \mathbf{B}\sigma_n, \mathbf{BB} \ldots$$

Emulating one move of M will be slightly different, depending on whether the head is to the left or the right of the starting position. The machine must have states for being on the right side of M, and another set of states that mean it should have been on the left side.

For example, state q_{26} in the original Turing machine would result in both $q_{26,\text{left}}$ and $q_{26,\text{right}}$. Figure 5.1 shows what one move on a tape with input 1101101 might look like. The blanks are copies from the left side of the two-way infinite tape; the 11011101BBBB … is a copy of the right side.

If one is working on what would have been the right side of the two-way infinite tape, a move right on the right is done by moving right two "new" cells; likewise a move left is a move left by two.

For example, if the usual Turing machine had the move $\delta(q_{26}, 1) = 1, q_{53}, R$ (that is, write 1, go to state q_{53}, and move the head right), then the semi-infinite tape machine will need to (1) write 1, (2) move right one cell, and (3) read and write the next symbol (for any σ in Σ), move right again, and go to $q_{53,\text{right}}$.

$$\delta(q_{26,\text{right}}, 1) = q'_{26,\text{right}}, 1, R \qquad \delta(q'_{26,\text{right}}, \sigma) = q_{53,\text{right}}, \sigma, R \text{ (for all } \sigma \text{ in } \Gamma)$$

A move right when one is on the left side goes toward the center, or left. If in any move right or left one sees the turnstile character ⊢, one has to switch states and adjust the move. Thus, if the first move of M would be to move left one cell, so it begins on the right side of the turnstile, but just barely, the new machine M′ will move left one cell, see the turnstile, change

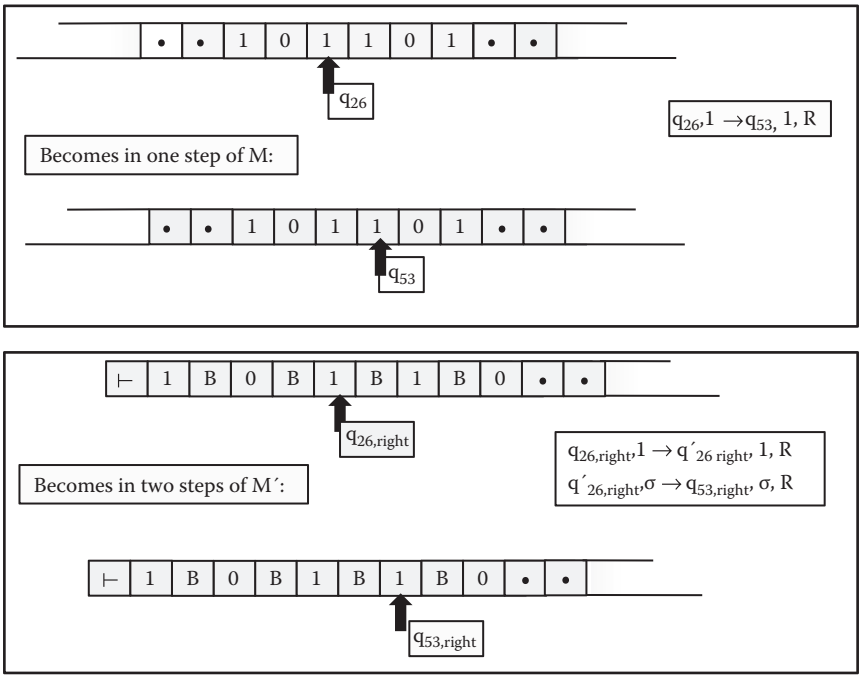


FIGURE 5.1 Emulation of two-way infinite tape.

to the left states, and move right to the first blank. In our example, the move from $q_{26,\text{left}}$ on a 1 will be more complicated—most of the time, it writes a 1 and moves *left* by two cells. But if it hits the turnstile in these two moves, it will have to do a lot of work (which is left to the interested reader).

Of course, a two-way infinite tape machine can emulate the behavior of any one-way infinite tape machine. It simply never uses the available tape to the left of the input string. ■

5.1.2 Stay Option

A Turing machine with a stay option is an extension of the usual Turing machine:

Definition 5.3

A stay option Turing machine $M = (\Sigma, \Gamma, Q, q_{\text{accept}}, q_{\text{reject}}, q_{\text{start}}, \delta)$ is defined as an ordinary Turing machine except that the delta function has the option of remaining on the same cell.

$$\delta: Q - \{q_{accept}, q_{reject}\} \times \Gamma \rightarrow \mathbf{Q} \times \Gamma \times \{\text{Left, Right, Stay}\}$$

is the **transition function**. ■

Theorem 5.4

Any stay option Turing machine M has an equivalent ordinary Turing machine M′.

Proof

Exercise. ■

5.1.3 Multiple Tapes

It is often convenient to have multiple tapes. By convention, a k-tape Turing machine starts with its input on the first tape, and the first head on the first character of the input. The other tapes all begin as blank tapes.

A move of this Turing machine now takes a state, a k-tuple of characters under the k-heads. It writes a character on each of the k-tapes, and moves each of the heads left or right. One can choose any k that is convenient for the problem—often three tapes allow enough flexibility. A sample three-tape Turing machine with input $\sigma_1\sigma_2\sigma_3\sigma_4\ldots$ is shown in Figure 5.2.

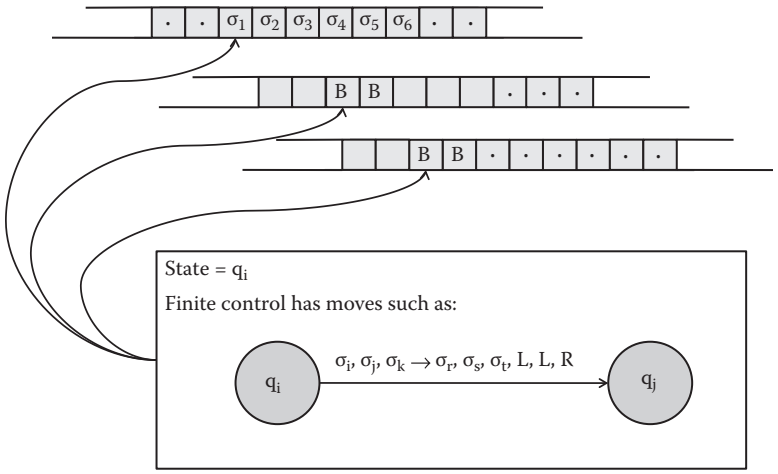


FIGURE 5.2 Partial picture of a three-tape Turing machine.

The formal definition for a two-tape Turing machine is:

Definition 5.5

A two-tape Turing machine $M = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_{start}, \delta)$ differs from the definition of an ordinary Turing machine only in the delta function, which maps a state, a symbol from tape 1, and a symbol from tape 2; writes new symbols to the two tapes; and moves the two heads left or right (unless it also has a stay option).

$$\delta: Q - \{q_{accept}, q_{reject}\} \times \Gamma \times \Gamma \rightarrow \mathbf{Q} \times \Gamma \times \Gamma \times \{\text{Left, Right}\} \times \{\text{Left, Right}\}$$ ■

The convention is that the machine begins with the input written on the first tape in the usual way, and the second tape begins blank.

Note: For many uses of multitape Turing machines, the stay option is very convenient—so that one head can remain in place while another head moves. Thus, we will generally assume a delta function of the form

$$\delta: Q - \{q_{accept}, q_{reject}\} \times \Gamma \times \Gamma \rightarrow \mathbf{Q} \times \Gamma \times \Gamma \times \{\text{Left, Right, Stay}\} \times \{\text{Left, Right, Stay}\}$$

To see how convenient this is, think of a simple example such as L = {strings of 0's and 1's such that the number of zeros is equal to the number of ones}. This can be done on one tape by moving back and forth, crossing out a zero and a one on each pass, and finding if all characters are crossed out on the same pass. But with three tapes, (1) move across the input once, copying each zero to tape 2 and each one to tape 3. Then (2) pass back across these two tapes, checking that heads on tape 2 and tape 3 hit the blank at the same time, so that the count of 0's must have equaled the count of 1's. Or, to accept the language of palindromes, $\{w \mid w = w^R\}$, simply copy the input to tape 2 and run two heads, one forward across the string and the other backward, checking that the symbols match (see Figure 5.3).

Another machine is shown in Figure 5.4 for the language $0^n1^n0^n$. This is an interesting machine because it would be much easier to write if one assumed a stay option for the multitape Turing machine. See Exercises 6 and 7 at the end of the chapter.

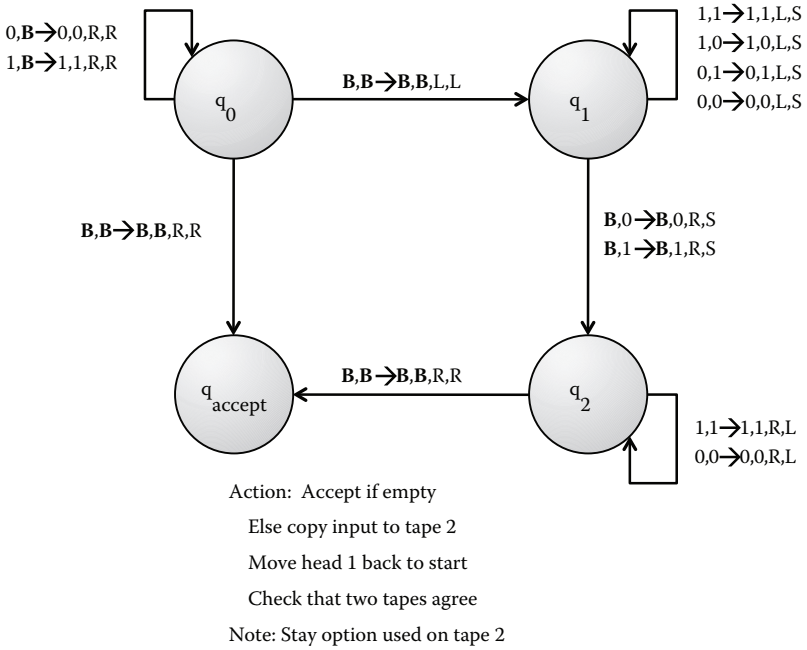


FIGURE 5.3 Two-tape TM for $w = w^R$.

Theorem 5.6

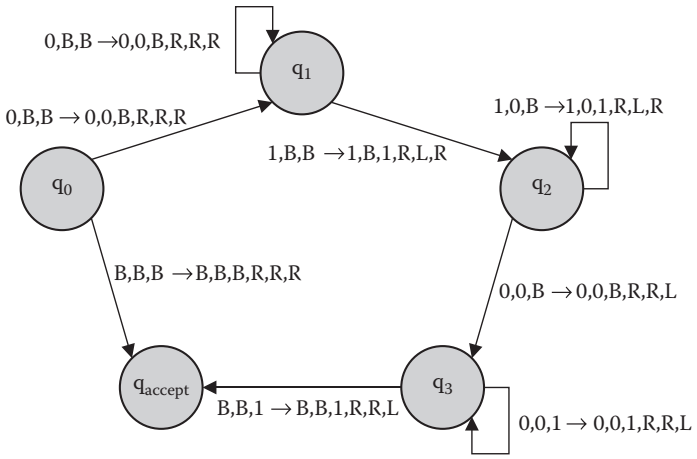
Any k-tape Turing machine M has an equivalent one-tape Turing machine M′. Moreover, t steps of M can be emulated in Kt^2 steps of M′, where K is a constant. Thus, if an algorithm is bounded by t(n) steps on an input of size n on a k-tape machine, it is bounded by $Kt^2(n)$ steps on a one-tape machine.

Proof

As in emulating a two-way infinite tape on a semi-infinite tape, there are many choices for building M′.

All the information from the k-tapes must be put onto one tape. Each tape can have a finite string on it, and each tape has its own head position.

A standard trick for encoding head positions is to double the alphabet—for each symbol σ in Σ, add another symbol σ′. Thus, when tape i is put onto the single-tape machine, the presence of the head on a cell will be encoded as that symbol with its prime.



Action
Copy 0's to tape 2
Copy 1's to tape 3 (while backing up across 0's on tape 2, if too many 1's, reject)
Move right across tapes 1 and 2, left across tape 3 (should end on last 1)

FIGURE 5.4 Three-tape TM for $0^n1^n0^n$; $n \geq 0$.

We will also add the turnstile ⊢ to the alphabet to separate the contents of the k tapes.

Thus, in a three-tape machine with $\Gamma = \{a, b, c, \mathbf{B}\}$, we construct a one-tape machine with $\Gamma' = \{\vdash, a, b, c, a', b', c', \mathbf{B}, \mathbf{B}'\}$.

If the three tapes contain: abba (head on first b), caa (head on c), and cc (head on first c), they would look as in Figure 5.5.

This would be put onto one tape as Figure 5.6, or written as the string abb′a⊢c′aa⊢c′c.

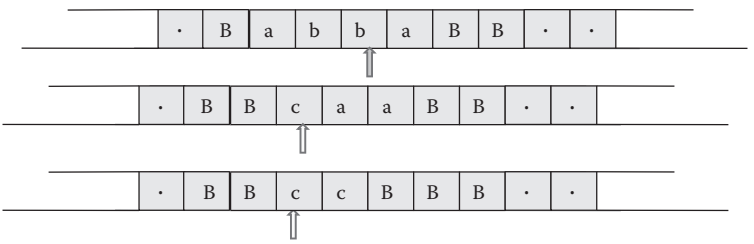


FIGURE 5.5 Three tapes with contents and head positions.

FIGURE 5.6 Contents stored on one tape (whose head may be on any square).

·	B	a	b	b′	a	⊢	c′	a	a	⊢	c′	c	·	

4 + 1 + 3 + 1 + 2 = 11 nonblank characters

Suppose δ (b,c,c) → b,c,c,L,L,L

Then the tape becomes:

·	B	a	b′	b	a	⊢	B′	c	a	a	⊢	B′	c	c	B	·

4 + 1+ 4 + 1 + 3 = 13 nonblank characters

FIGURE 5.7 Emulation of one move on three tapes.

The head of this new machine would have to scan the entire tape, remembering (in its states) what should be written for each primed symbol, and where each head should go. It would then rescan the tape, making the replacements and changing the symbols left or right to primed symbols.

One complication is that new space may have to be created within the string—for example, if all the heads were to move left, the new string would become (if all heads write the same symbol they read) ab′ba⊢**B′**caa⊢**B′**cc, shown in Figure 5.7.

To create space within the string, some part of the string will have to be shifted right (or left, but in the following, all shifts are to the right).

Now count what it takes to carry out one of these steps. If the length of the input, written on the first tape, started as $|I|$, after t steps, the length of the nonblank portion of the first tape is at most $t + |I|$, and of each of the other $(k - 1)$ tapes is at most t. There are $(k - 1)$ turnstile symbols. Thus, storing all k-tapes on one tape takes at most $|I| + kt + (k - 1)$ cells. Note that k is fixed and, for a given input, so is $|I|$, but t increases, so after enough time elapses, $t > k$ and $t > |I|$.

Thus, the length of the nonblank tape (and the time it takes to scan this much of the tape) is

$$|I| + kt + (k - 1) < t + kt + (k - 1) < t + kt + t = (k + 2)t$$
$$\text{when } t > |I| \text{ and } t > k.$$

Suppose one wants to shift a string of length L to the right. One simple way to do that is:

1. Move to right end of string (L steps); move to last symbol (one step).
2. Read this symbol and move right (one step), copy symbol, and move left (one step).

3. Go left to the next symbol to copy (one step).
4. Repeat steps 2 and 3 until all symbols are moved right (3L steps).

$$\text{Total number of steps: } L + 3L = 4L$$

So, if one needs to move part of all of a string of length L to the right, it can be done in 4L steps. Here, we have a string whose length, at time t, is bounded by $(k + 2)t$ when $t > |I|$ and $t > k$.

Inserting one blank into the string means moving part of it right; this is bounded by

$$4(k + 2)t \text{ when } t > |I| \text{ and } t > k$$

There are no more than k spaces or holes that have to be inserted, so this is the time spent inserting spaces and copying is bounded by

$$k4(k + 2)t$$

And the time to do one move is bounded by

Read entire string: $(k + 2)t$
Make at most k holes: $4k(k + 2)t$
Fill each hole appropriately and mark head positions: $(k + 2)t$
Reset head to start of string: $(k + 2)t$

Adding all together, we get $4k(k + 2)t + 3(k + 2)t = [4k + 3)(k + 2)]t = Kt$.

Thus it takes Kt steps to carry out one step of the k-tape Turing machine, or Kt^2 steps to carry out t steps. It follows, of course, that if the k-tape Turing machine halts on input I of length $|I|$ in $t(|I|)$ steps, then the one-tape emulator will halt in $O(t^2(|I|)$ steps.

A k-tape Turing machine can easily simulate the behavior of a one-tape Turing machine. It simply ignores its extra tapes. This completes the proof that one-tape and k-tape Turing machines are equivalent. ■

5.1.4 Nondeterministic Turing Machines (NDTMs)

Nondeterminism is a key concept in theoretical computer science. In general, it means a choice of paths or moves—for a Turing machine, δ

need no longer be a function. Instead, $\delta(q, \sigma)$ could have 0, 1, 2, or more possible moves from a specific state q and on one symbol σ. This would mean that the action of a machine on a string is not well determined, but could have a variety of outcomes. In fact, some of the time it could reach the accepting state, some of the time end in the reject state, and other times loop forever—all of these outcomes could be possible on the same string.

Definition 5.7

A nondeterministic Turing machine (NDTM) **accepts** a string w if *some* (or *any*) sequence of legal moves from the start state on w reaches q_{accept}. ■

It does not matter if it only reaches this state 1% of the time, and ended in q_{reject} the other 99%. It still accepts.

In other words, for a nondeterministic machine, a string is accepted if there is any possible way to accept it. You can think of this as follows: it is accepted if an omniscient omnipotent being were running the machine, telling it which move to pick. Only if there is absolutely no sequence of moves leading to an accepting state will the machine then not accept.

A simple, but perhaps instructive, example would be the design of TM to accept strings that contain the substring 11. Of course, this is easy to do with a standard deterministic TM. But a nondeterministic TM could just guess how many steps to move right. It moves right that many times, and then checks that the current symbol and next symbol are both 1. What if they are not? In a string of length 1000 with only one occurrence of 11 (such as $0^{698}11(01)^{150}$), the chances of landing on that first one are very small! And what if it moves right so far that it is off the string? It does not matter. As long as the string contains 11, then there is some way of moving right to that spot and accepting, so it does.

Figure 5.8 shows a state diagram for this nondeterministic Turing machine (though of course, again, it is simple to write a deterministic Turing machine for this language).

(The enterprising student might like some analogy of nondeterminism on homework or exams. That is, if there is any right choice to make, then we just assume that choice is the one made—if one can choose correctly, then one does choose correctly.)

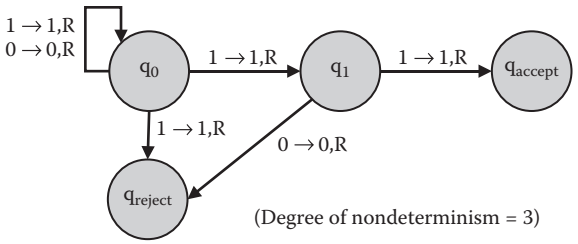


FIGURE 5.8 One possible NDTM for {w: w contains substring 11}.

Definition 5.8

A nondeterministic Turing machine $N = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_{start}, \delta)$ in which:

Σ is the **input alphabet** with $\mathbf{B} \notin \Sigma$.
Γ is the **tape alphabet** with $\mathbf{B} \in \Gamma$ and $\Sigma \subseteq \Gamma$.
Q is a finite **state set**.
$\mathbf{q_{accept}} \in Q$ is the unique **accept state**.
$\mathbf{q_{reject}} \in Q$ is the unique **reject state**.
$\mathbf{q_{start}} \in Q$ is the specially designated **start state**.
δ: $Q - \{q_{accept}, q_{reject}\} \times \Gamma \rightarrow \mathscr{P}(Q \times \Gamma \times \{\text{Left, Right}\})$ is the **transition relation**.

Here, $\mathscr{P}$ is the power set, so each element is mapped to—possibly—a set of combinations. ■

The only difference between this definition and that of its deterministic counterpart occurs in its δ component. Examine this carefully: it maps into the power set! Thus, the value of $\delta(q, \sigma)$ is a set of zero or more triples, and so δ is a relation and not a function. Thus, it is possible that from a certain position (in state q, reading σ) that there is no possible move, one possible move (as in an ordinary TM), or many possible moves.

Theorem 5.9

Any nondeterministic Turing machine M has an equivalent deterministic Turing machine M′. Moreover, t steps of M can be emulated in 2^{Kt} steps

of M′, where K is a constant. Thus, if an algorithm on input of size n is bounded by t(n) on a nondeterministic machine, it is bounded by $2^{Kt(n)}$ on a deterministic machine.

Note: Of course, a deterministic Turing machine is also a nondeterministic machine where the number of choices at each step is exactly one. So the theorem says that a language has an ordinary TM if and only if (iff) it has a NDTM.

Proof (of Theorem 5.9)

The idea is simple: the deterministic Turing machine must simply try all possible sequences of moves that the nondeterministic machine could make, and accept if it ever finds an accepting sequence of moves. The only complication is in how to systematically try all sequences of moves. It cannot take a sequence and follow it until it reaches a resolution, since it might never do so. And, it does not have to be efficient, so it can restart at the very beginning each time.

Thus, it should try:

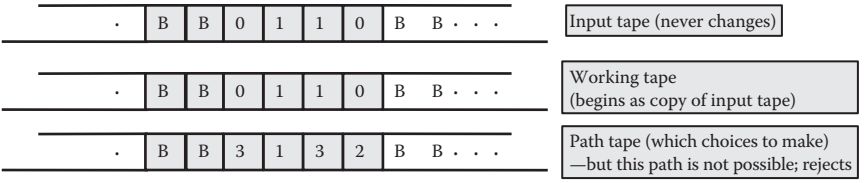
All sequences of moves of length 1 (all possible configurations that it can reach in one move)
All sequences of moves of length 2
All sequences of length 3
…

The action of a deterministic emulator for a NDTM is shown in Figures 5.9 and 5.10.

It keeps generating and emulating the next sequence of moves until it reaches an accepting state, which of course may never happen.

The Turing machine will need some way to keep track of what sequences of moves it has tried and what to try next. A multitape machine will be useful, and one of the tapes will be used to keep track of the sequences of moves that have been tried.

Note that the nondeterministic machine's transition relation may sometimes have no moves it can make (in which case we assume it goes to q_{reject}), or one choice, or more than one choice. But there are only a finite



	State	Read	New State	Write	Move
1st	q_0	1	q_0	1	R
2nd	q_0	1	q_1	1	R
3rd	q_0	1	q_{reject}	1	R
1st	q_0	0	q_0	0	R
1st	q_1	0	q_{reject}	0	R
1st	q_1	1	q_{accept}	1	R

Table for δ function

Paths in Order:
1
2 (no such move)
3 (no such move)
11 (still in q_0)
12 (now in q_1)
13 (now in q_{reject})
. . .
113 (now in q_{reject})
121 (accepts!)

FIGURE 5.9 Emulating NDTM of Figure 5.8 using three tapes.

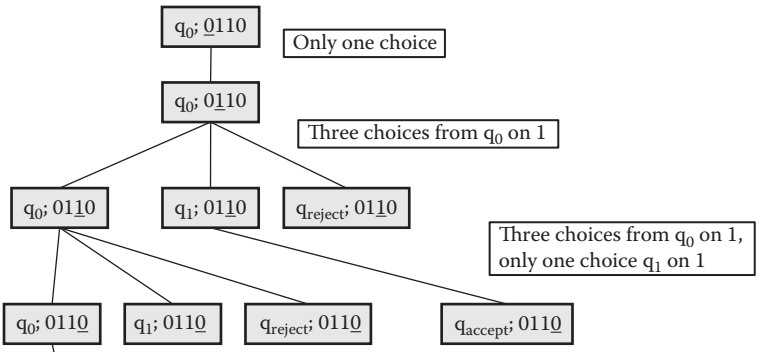


Table for δ function

	State	Read	New State	Write	Move
1st	q_0	1	q_0	1	R
2nd	q_0	1	q_1	1	R
3rd	q_0	1	q_{reject}	1	R
1st	q_0	0	q_0	0	R
1st	q_1	0	q_{reject}	0	R
1st	q_1	1	q_{accept}	1	R

FIGURE 5.10 Emulating NDTM of Figure 5.8.

number of state–character combinations. The maximum degree of non-determinism for all (state, symbol) combinations will be called D. Thus, there are at most D moves from any configuration.

$$D = \text{maximum number of choices in the delta function}$$

Note that

$$D \leq 2\,|Q|\,|\Gamma|$$

and that D = 1 when the machine is deterministic.

Consider the integers base (D + 1) generated in lexicographical order:

1; 2; …, D; all single moves from q_{start}

11; 12; …; DD; all sequences of two moves from q_{start}

111; 112; …; DDD; all sequences of moves of length 3

…

Now a sequence such as 233D2 would mean:

From the start, take the second possible move.

Now, take the third possible move.

And again, take the third.

Now take the Dth possible move.

Finally, take the second possible move.

If this accepts, fine.

Else try the next sequence, which will be 233D3.

Note: By using integers base D, there is no need to be concerned about numbers with more than one digit. It would be possible to use base 10 (or any other base). However, then if D > 9, we will need an extra separator symbol so that one can determine if 233 means (2nd move, 3rd move, 3rd move) or (23rd move, 3rd move) or (233rd move) or …. One can use the semicolon as above between sequences of moves and commas between

individual moves. Thus (2nd move, 3rd move, 3rd move) = 2, 3, 3; but (23rd move, 3rd move) = 23, 3; this extra complication is an "implementation detail" and is entirely avoided if D < 10 or by using integers base (D + 1).

The first tape will hold the input and never change, to make it easy to start over with a new sequence of moves.

The second tape holds the sequence in base (D + 1) as above.

The third tape is the working tape—each cycle begins by copying the input tape to the working tape. The sequence as read from tape 2 is carried out on this working tape. If the machine does not enter q_{accept}, the working tape is erased before the next cycle.

One does need some convention for the action to carry out if the move number called for does not exist. For example, what should one do if one is supposed to make the fourth move from (q_{127}, a), and there is only one move available from that (state, symbol) combination? One can either skip that sequence (just erase it and move on to the next sequence), or one can use the convention that the highest move k is used for all moves k, k + 1, …, D.

This algorithm is written in pseudocode (see Figure 5.11).

How many steps does this take? If the nondeterministic machine accepts after t steps, the deterministic machine will have tried all the paths of length 1, 2, 3, and so on, up to (t – 1), and some of the paths of length t.

```
n = 1; tape 3 = {1}  // try first move out of start
Repeat for all strings on tape 3 of length n
   Copy input tape (tape 1) to working tape (tape 2).
   Repeat until
     (1) no symbol remains on Tape 3   or
     (2) the symbol on Tape 3 is not a valid move or
     (3) a rejecting configuration is found
     Use Tape 2 to simulate one branch of
     nondeterministic computation. Consult Tape 3 at
     every step to determine which choice to make from
     transition function. If an accepting configuration
     is found, Accept.
   Erase tape 2
   Change string on Tape 3 to lexicographic next.
Let n = n+1
```

FIGURE 5.11 Pseudocode for the emulator of NDTM.

There are D paths of length 1, D^2 of length 2, …, up to D^t paths of length t.

$$D + D^2 + \ldots + D^t = (D^{t+1} - D)/(D - 1) = D(D^t - 1)/(D - 1) < 2*D^t = 2*2(^{\log_2 D})^t = 2^{Kt}$$

This proves that the emulator is $0(2^{Kt})$. ■

Important Note: These two theorems above say that a polynomial algorithm on a *k-tape* Turing machine will remain polynomial when transformed to a one-tape machine. That means that if we want to prove that an algorithm has a polynomial order, but we do not care what the order of the polynomial is, we can use any number of tapes that we like.

On the other hand, we cannot use a *nondeterministic* polynomial algorithm to show that a problem has a polynomial algorithm. In moving from nondeterministic to deterministic machines, an algorithm may become exponential.

5.1.5 Other Extensions and Limitations of Turing Machines

There are many other extensions that can be applied to the basic Turing machine model that are Turing-complete:

- Allow multiple heads on one tape.
- Use a two-dimensional infinite tape.
- Replace tape with two stacks of unlimited depth.

The following changes, however, are not Turing-complete:

- Replace tape with one stack of unlimited depth (pushdown automaton or PDA).
- Allow only use of the input section of the tape (linear bounded automaton or LBA).
- Allow only reading of the input I plus log|I| space for calculation (logspace machine).

We will see below (Church–Turing thesis) that no extension adds any power to the Turing machine, though it may add convenience. Some limitations, such as LBAs, in which the tape used is limited to only the cells occupied by the input, do restrict the power of the machine in interesting

and useful ways. The interested reader is directed to any of the standard books on the theory of automata.

5.2 TURING MACHINES TO EVALUATE A FUNCTION

In this course, Turing machines are used to answer decision problems. However, there are other kinds of Turing machines that can be developed.

If *f* is any function from strings to strings (and this is any computational problem that can be imagined), then a Turing machine M **calculates** (or **computes**) the function *f* if it starts with the input(s) $x_1, \ldots, x_n$ to *f* on its tape, and ends (in some specially designated state q_{done}) with $f(x_1, \ldots, x_n)$ on the tape. This kind of Turing machine is sometimes called a **transducer**. A simple example would be a machine to add the numbers n and m, written in decimal.

Addition Turing machine transducer:

Start: q_{start}, n#m

Finish: q_{done}, n + m

5.3 ENUMERATING TURING MACHINES

A Turing machine **enumerates** a language L if it has a special state q_{found_one}. An enumerating TM takes no input. It starts with an empty tape, and lists the strings in L, one by one. Each time it writes a string from L onto its output tape, it announces that it has found one by entering the special state q_{found_one}. It can then erase the tape and work on producing another string from L. Eventually, every string in L must be enumerated (although not in any particular order).

Enumerating Turing machine for L={n|n is prime}
Start: q_{start}, empty tape
Intermediate configurations
: $q_{found_one,}$ 13; q_{found_one} 2; q_{found_one} 7; q_{found_one} … 23 …
(in any order whatsoever, with duplicates allowed)
Finish: never ends (unless L only has a finite number of strings)

It would be easy to design a Turing machine to enumerate the language L = {w | w starts with aa} over Σ = {a, b, c}. It would be less easy, but not difficult, to design one to enumerate L = {w | w is the decimal representation of a prime number}. It would be natural to enumerate the strings

in lexicographic order, but that is not necessary—it can enumerate them in any order whatsoever; it can list the same string more than once; the only requirement is that every string in the language will eventually be enumerated.

There are many variations on Turing enumerators. For instance, it certainly is not necessary to erase the tape between strings; they can be written one after another on the tape with a separator symbol:

13#2#7#23# …

It is not necessary to have a special state q_{found_one} either; the enumerator can use the separator symbol to announce the end of each string that it finds. Finally, it may be useful to have one or more working tapes as well as an output tape.

5.4 THE CHURCH–TURING THESIS

There have been various models of what *computation* means. Programs written in any one of the many programming languages are some of these models, but the idea of computation goes back to well before the existence of computers and computer languages. The question goes back to Leibniz, and was one of the important open problems of the early 20th century (David Hilbert made several references to it). But it was only about 1930 that the problem was made sufficiently precise to be answered. Two logicians/mathematicians who first answered this question were Alonzo Church and Alan Turing.

Alonzo Church was an American mathematician at Princeton. He developed the model of computation called lambda calculus. Alan Turing was a British mathematician. He is well known for his code-breaking work during World War II at Bletchley Park. But, he also wrote (1936) a paper, "On Computable Numbers, with an Application to the Entscheidungsproblem." The Entscheidungsproblem is the problem of finding an algorithm for deciding whether a proposition in first-order predicate logic is always true.

The work of Church and Turing showed that there is no algorithm to solve the Entscheidungsproblem. Part of the work in doing this requires the formalization: What is an algorithm? Church used lambda calculus, Turing used Turing machines (not his name for them). It was fairly easily shown that these two characterizations of algorithms (Church's and Turing's) are equivalent.

Other theoretical ideas of computation have also been in use—recursive function theory, Post systems, Markov systems, register machines. (A register machine is the simplest possible computer; its only instructions are increment, decrement, test-and-branch, halt.) Each of these models can be simulated by a Turing machine.

The Church–Turing thesis states that *every* model of computation—even those not yet in existence—can be done by, and is equivalent to, a Turing machine.

Note that there is no hope of proving this. It is not possible to prove something about a model of computation that has not yet been proposed. However, the thesis has been verified for all known models of computation. It is widely accepted in the communities of logic, mathematics, and computer science. But its unprovability is why it has the name *thesis* instead of *theorem*.

Thus, we will use a Turing machine as our model of computation. Or, stated otherwise;

Computation = expressible on a Turing machine

Algorithm = expressible on a Turing machine that halts

Whatever can be done in some programming language can be done on a Turing machine. And, if someone thinks of a new model of computation, the first question will be if it is Turing-complete. One can also ask if a Turing machine can do anything it can do, but of course we all expect the answer to be yes.

The Church–Turing thesis states that Turing machines can carry out algorithms done by any model of computation whatsoever, certainly including standard computer operations. At first glance, Turing machines certainly seem less powerful than the standard desk PC. The simplicity of the Turing machine is its best trait; it may be hard to write a complete Turing machine to do any interesting task, but theoretically, the operation is very straightforward.

It is not hard to write a program to emulate a Turing machine (in C++ Java, or other language). You can do it yourself, or you can find such programs available on the Web. There may be a problem if the machine runs out of memory—Turing machines have unlimited tape to use—but if necessary, one could suppose adding external memory as necessary. Theoretically, the total amount of external memory available in the world today is limited, but that limit is so large that this may not be a practical limitation.

One can prove that for a fairly standard computer running a standard language, anything that can be done on that computer (for instance, an IBM supercomputer running Ada) can be done on the standard PC (some small machine running C, perhaps). Moreover, if it has a polynomially bounded algorithm on the supercomputer, it also has a polynomial algorithm on the PC and on the Turing machine. The proof here is not easy. It involves:

1. Specifying the standard computer and its entire instruction set
2. Showing how to emulate each of the standard computer's instructions in polynomial time on a Turing machine

5.5 A SIMPLE COMPUTER (OPTIONAL)

The problem here is the amount of detail that one is willing to go through to specify details of an ideal computer, and then showing how to emulate each instruction on a Turing machine. Various authors have done this (Hopcroft et al., 2007; Lewis and Papadimitriou, 1997; Homer and Selman, 2001); each has a slightly different approach.

First, the two big differences between a Turing machine and a computer are:

1. **Sequential access versus random access:** A computer can access any memory cell in one instruction ($O(1)$), but a Turing machine must begin at one end of its tape and move position by position along the $O(|\text{tape length}|$ so access is an operation that is the length of its tape to reach and then access any one memory location.
2. **The instruction set:** A computer will have a whole set of machine instructions, but a Turing machine can only read, write, change state, and move left or right.

Let's suppose a computer with one dedicated register, a program counter (PC). It could also have k other registers, but these are not necessary. The machine below has three address registers, but they could be eliminated. All instructions also work on simple memory locations. On a real computer, it is generally faster to have operands in registers, but only by a factor of a constant. And, for theoretical purposes, there is no distinction between an operation that takes 1 nanosecond and one that takes a constant K nanoseconds, even if K is large—algorithms that differ only by a constant factor are essentially indistinguishable.

We will assume for memory that there are an unbounded number of cells, and that each cell can hold an arbitrarily large number. Again, both of these are unreasonable—computer memories are bounded but very large (especially if one considers CDs, DVDs, USB drives, tape drives, and other external memory as part of memory). There is a bound on the largest number that can be stored in one word of memory (size of an integer, or long integer, or something similar). The way around bounded word size is to store large integers and real numbers in linked lists of memory cells, but we will assume that is not necessary.

Our model of a computer will have:

- Program counter: Automatically incremented at each instruction, except for jumps (PC).
- Three registers: Used for any sequence of bits, generally used as address registers (AR1, AR2, AR3).
- Infinite memory: Each memory word addressed by an integer (1, 2, 3, …); each can hold any sequence of bits.
- Program: Set of instructions, stored in memory.
- Input from some standard device; output to some standard device.

Notice that we assume that each register (PC, AR) and each memory cell can hold arbitrarily long sequences of bits. Despite the limited instruction set and the lack of registers, this makes our model more powerful than the computer that you use.

Instruction set: We will choose an instruction set to be small but not miniscule. Here, one can get into interesting side excursions. A "Turing tarpit" is a computer language designed to be Turing-complete but as small as possible—several different tarpits have been developed. Each must be able to read a symbol, write a symbol, increment (or decrement), and have some kind of branch. At least one language has managed to do all this in one instruction. There is a language whose only instruction is "subtract and branch if negative." There is also the language OISC (a take-off on RISC, a reduced instruction set language)—the one instruction set computer—whose only instruction also combines the increment/decrement and the branch.

The instruction set we will use has a subset of some of the standard three-address instructions. Addressing can be:

```
READ ar                 MEM[ar] LOADED FROM INPUT
READ @ar                MEM[MEM[ar]] := INPUT
WRITE ar                MEM[ar] MOVED TO OUTPUT
ADD ar1, ar2, ar3       MEM[ar3] := MEM[ar1]+MEM[ar2]
ADD @ar1, ar2, @ar3     MEM[MEM[ar3]] =
                        MEM[MEM[ar1]]+MEM[ar2]
HALF ar1 ar3            MEM[ar3] := MEM[ar1]/2
SHFT ar1                MEM [ar1] = MEM[ar1] * 2
JMP C                   PC = C
JPOS ar C               IF MEM[ar] > 0, PC = C
```

The Turing machine to emulate this computer will have several tapes:

Tape 1: PC (holds one address, which can be arbitrarily long)

Tape 2: Address tape, used for fetching operands (holds up to three addresses)

Tape 3: Memory, stored in the form of (address, contents) (unlimited number of pairs)

Tape 4: Input tape (holds whatever the standard input device would contain) (unlimited)

Tape 5: Output tape (TM writes all output to this tape) (unlimited)

Tape 6: Scratch work tape, if desired (unlimited)

Example: ADD 23, @45, 96

1. Write 23 onto tape 2, the address tape.
2. Scan the memory tape until the number 23 is found; replace 23 with contents onto tape 2.
3. Write @45 onto tape 2 (following the number written in step 2).
4. Scan memory tape until 45 is found; copy its contents onto tape 2 (replace the 45).
5. Find this cell on the memory tape; replace the address by its contents.
6. Add these two numbers, using the scratch tape if desired.
7. Find cell 96 on the memory tape.
8. Replace its contents by the sum from tape 2 (perhaps moving tape over to make room).

This simple computer starts its computation with its program stored in memory (tape 3) and all its input on tape 4. The combined length of these tapes is some number I. After t steps of the program, a three-address instruction set cannot have accessed more than 3t memory locations.

At each step, the computer uses at most one new memory location. The number stored in that memory location is at most size (I + t). No result of addition, doubling, or other operation adds more than one digit to a number in base 2. Thus, after t steps, the contents of tape 2 have length bounded by Kt^2.

This means a one-time-unit random access to memory can be emulated by the Turing machine in Kt^2; thus t steps can be emulated in Kt^3.

Now, recall that a multitape Turing machine can be emulated by a single-tape Turing machine, and that the order is at most squared. Thus, the simple computer can be emulated by a standard Turing machine in $K't^6$—and importantly, what is polynomial on this simple computer is still polynomial on a standard Turing machine.

5.6 ENCODINGS OF TURING MACHINES

For many uses of Turing machines, we want to encode them (and their inputs) into some standard alphabet. There are several reasons to do this. First, we are writing different Turing machines for different problems. This is not efficient; we do not build different computers for different problems; rather, we write different programs that run on the same computer. Second, we want a (relatively) short and machine-readable description of a Turing machine—something much more standard and concise than the rather long English descriptions or complicated diagrams we have been writing.

In computer science, this alphabet is most naturally $\Sigma = \{0, 1\}$. To use only two characters, one of the characters must be a separator. Thus, all information must be encoded as strings of 0's; we will need to use the 1 as a separator character (or, of course, vice versa).

One problem is how to encode the alphabet of the Turing machine. If all alphabets were subsets of some standard alphabet such as the ASCII or Unicode sets of characters, there would be less of a problem, since each character in any alphabet could be encoded in Unicode. In this sense, all "regular" characters have already been coded into the alphabet of {0, 1}). But there is no restriction on the alphabets of formal languages requiring them to use any symbol already in existence anywhere.

There is one restriction on tape alphabets Γ. Any alphabet must include the character blank **B**, so it is natural to encode the blank as 0.

One option would simply be to list the characters of the alphabet in some order, and assign them encodings in that order, starting with 00 (since 0 is the encoding of the blank **B**).

Examples

Σ_1 = {a, b, c} would be encoded as:

a = 00

b = 000

c = 0000

Σ_2 = {モ,ム,オ} would be encoded as

モ = 00

ム = 000

オ = 0000

One problem here is that the two different languages,

$$L_1 = a^n b^n c^{2n}$$

$$L_2 = モ^n ム^n オ^{2n}$$

would have exactly the same strings once they were encoded. However, the two languages, though different, are in some sense the same (they are mathematically isomorphic), so this need not be particularly worrisome. This is the approach that we will use in this book (but like many other conventions, other authors will make other choices).

A Turing machine is defined by its: (1) set of states, (2) input and tape alphabets, and (3) delta function. Each (useful) state appears as the object

of a move in the delta function, and each useful input or tape alphabet symbol will appear somewhere also in the arguments or values of the delta function. Thus, we can retrieve all the information about the TM from its delta function. We do need to know what q_{start}, q_{accept}, and q_{reject} are, but we can encode them in a standard way:

$$q_{start} = 0$$

$$q_{accept} = 00$$

$$q_{reject} = 000$$

(The alert reader may complain that this assumes that these are three distinct states, and does not allow, for example, $q_{start} = q_{accept}$. This is true, but it does not eliminate any interesting Turing machines.)

Moves of the read/write head can be encoded:

$$\text{Left} = 0 \qquad \text{Right} = 00$$

Then one move of the Turing machine's transition function will look like:

1state1symbol1new_state1new_symbol1head_move1

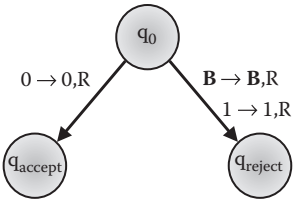
Thus, in a Turing machine with q_{start}, q_{accept}, q_{reject} and four other states $q_1 = 0000$, $q_2 = 00000$, $q_3 = 000000$, $q_4 = 0000000$, a move from q_1 reading c to q_{accept} writing a blank and moving left would look like

100001000010010101 = 1<q_1 encoded>1<c encoded>1<q_{accept} encoded>1<blank>1<left>1

The encoding of the entire TM can be done:

$$1\text{move}_1\ \text{move}_2\ \text{move}_3 \ldots \text{move}_{last}\ 1$$

One more note: A Turing machine with n moves can have them listed in any order—thus, there are n! orderings of the moves, and thus n! different (at least, the strings are different) encodings of the same Turing machine. That is not a problem.



Encoded: $q_0 = 0$, $q_{accept} = 00$, $q_{reject} = 000$
B = 0. 0 = 00, 1 = 000
L = 0. R = 00

Three moves encoded:

1	Leading 1
101001001001001	q_0, 0 → q_{accept} ,0, R
10101000101001	q_0, B → q_{reject} , B, R
101000100010001001	q_0, 1 → q_{reject} , 1, R
1	Closing 1

String encoding the Turing machine:
1101001001001001 101010001010011010001000100010011

Encoding of the string 10: 10001001

TM reading 10: 1101001001001001 101010001010011010001000100010011 10001001

FIGURE 5.12 Encoding of a simple Turing machine.

Strings are easy to encode. The string bbaca would become, under this scheme,

$$1100010001001000010011$$

An encoding of an entire simple Turing machine (M_1) is shown in Figure 5.12.

If one encodes a Turing machine followed by its encoded input string, note that one can find the end of the machine and the beginning of the string at the point where four 1's appear.

Thus any TM, with input, can be encoded as a string of 0's and 1's.

Does any string of 0's and 1's correspond to the encoding of a TM? Not unless we take the convention that it does. Note that no string that we get as the encoding of a Turing machine ever has more than four consecutive 1's (and never has three in a row either). And, the pattern of pairs on 1's is constrained at the start and end of moves. The restriction that the transitions be a function (never more than one move from a given state, symbol pair) makes more constraints on what can be a legal TM encoding.

It would be possible to take a string of 0's and 1's, analyze it, and decide if it corresponded to a legal TM. However, a simpler option is to say that any string that is not a legal TM corresponds to a TM with no moves at

all—and thus, a TM that never accepts any input. If one also takes the convention that an undefined move goes to q_{reject}, then this is a TM that rejects all inputs.

5.7 UNIVERSAL TURING MACHINE

In designing Turing machines for problems, there is a new machine for each new problem. This is not really necessary. It is fairly easy to design a universal Turing machine (UTM). The input to the UTM is: (1) an encoded Turing machine M and (2) the encoded input I to M. Then, the UTM emulates M on I, and can announce whether M accepts or rejects I, or what else it may do.

The input to the UTM is a string of 0's and 1's, the encoding of M and I as outlined in the section above.

One way to build M is as a three-tape Turing machine:

Tape 1: Input tape, holds M and I (encoded).

Tape 2: Working tape, holds the input to M, in encoded form.

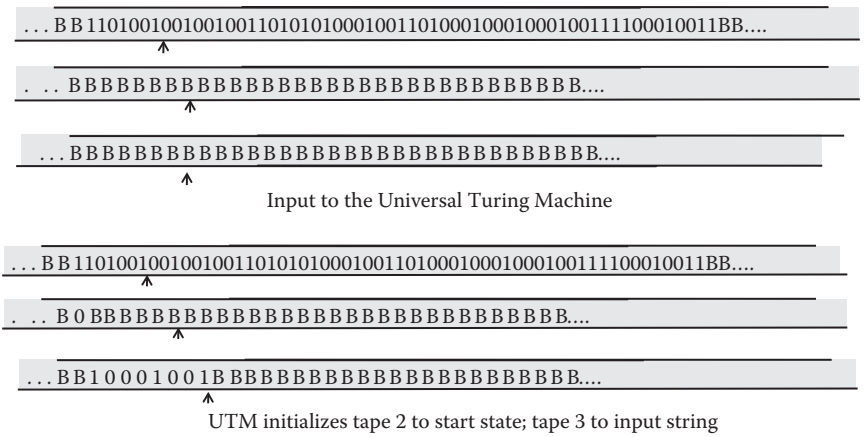
Tape 3: Current state (encoded, a sequence of zeros).

Other tapes can be added if convenient; for example, an extra working tape might be useful in shifting parts of tape 2 to the right or left.

The UTM begins its operation by (1) copying the encoding of the input I onto tape 2, and (2) copying 0 (for q_{start}) onto tape 3.

Now, it looks at the 0's on tape 2 for the current input symbol. It must find the place in the encoding of M that matches this symbol and the state (from tape 3). If it finds a match, it must change tape 2 into the proper configuration (shifting contents left or right as it needs to add or delete some number of 0's). If there is no match, by convention it writes 000 (for q_{reject}) on its tape 3 and halts and rejects (entering its own $q_{UTM\text{-}reject}$). Similarly, if it finds that it writes 00 for q_{accept}, it goes to its own $q_{UTM\text{-}accept}$. A simple encoded Turing machine (M_1) and a demonstration of one step of the universal Turing machine is shown (see Figures 5.13 and 5.14).

The advantage of the universal Turing machine is that it is a machine with a simple standard alphabet ({0, 1}) that can emulate every Turing machine. Its only task is to read a Turing machine (so the machine that it reads is like a program for the UTM), and the input to that TM or program, and run the TM/program on its input.


. . . B B 1101001001001001101010100010011010001000100010011111000 10011BB....
. . . B B B

EXERCISES

1. Repeat Chapter 4, problem 3, but use multitape or nondeterministic Turing machines if convenient.

2. Write out the encoding of the Turing machines from Chapter 4, problem 3.

3. Suppose one is emulating a two-way infinite tape on a semi-infinite tape, in the manner described in the text above. Suppose that the ordinary two-way TM has a move: $\delta(q_5, a) = q_{11}, c, L$.

   Now recall that q_5 needs at least two new states on the semi-infinite emulator, one $q_{5,right}$ for moves on the right half of the two-way infinite tape, another $q_{5,left}$ for moves on the left half.

   Carefully show how to emulate the move listed on the half-infinite tape. Keep it as simple as possible, but you may still need to add states.

4. Prove carefully: Any ordinary one-tape Turing machine with a stay option can be emulated by an ordinary Turing machine. A stay-option TM in a state reading a symbol will move to a new state, write a new symbol, but it may move left, right, or stay—remain—on the same cell.

5. The stay option is particularly useful for a two-tape Turing machine. For example, to accept the language $L = \{w: w \text{ is a palindrome over } \{0, 1\}\} = \{w|w = w^R\}$, one would like to: (i) copy w onto tape 2, (ii) move the head on tape 2 back to the start of the string but not move the head on tape 1, and (iii) move the two heads L and R, respectively, comparing the symbols under each head. Show how to simulate a move that would (a) move head 2 left by one cell, but (b) leave head 1 unchanged.

6. Assuming a stay option, write a simple two-tape Turing machine for the language $0^n1^n0^n$, $n \geq 0$. What is the order O of your algorithm?

7. Consider a machine S that has a set of states Q and two stacks S_1 and S_2 of unlimited depth. It has a read-only tape used for input. Each action considers (or ignores): next input character, top symbol of S_1, top symbol of S_2. In one move, it can (i) advance input (or not), (ii)

change states, and (iii) push new strings (possibly empty) onto both S_1 and S_2.

a. Show how a (possibly multitape) Turing machine can emulate a two-stack machine S.

b. Show how S can emulate a Turing machine.

8. Use the Web to search for arguments against the Church–Turing thesis. Analyze any arguments that you find. Do you find them cogent? Why or why not?

9. Find a Turing machine emulator program. Many are available on the Web. Which particular model does it emulate? Write and run a Turing machine of your design.

CHAPTER 6

Undecidability

6.1 INTRODUCTION AND OVERVIEW

Our definition of an algorithm is a Turing machine that decides a language—that is one that always halts by entering its accept or reject state as appropriate.

Now it is time to tackle what it means to be undecidable. This will take some effort to prove, and also will take effort to understand the proof when we have it. First, we need to lay some groundwork.

What does it mean to be undecidable? The definition will be easy: "no algorithm exists"; that is, there cannot be any Turing machine that halts on all inputs with a yes or no answer. But understanding this definition, and using it to show certain problems are undecidable, is a harder proposition.

In particular, finding the first undecidable problem takes some work. The only way to prove that *something does not exist* will be by contradiction—for a particular problem (acceptance of a string by a Turing machine), assume that an algorithm exists, then show that this produces an impossible situation. That will mean that our assumption about the existence of the algorithm is wrong.

6.1.1 Problems That Refer to Themselves (Self-Reference)

The key argument will come from self-reference. Self-reference means applying something to itself. For example, if I say, "I always lie," that means that every statement I make, including the one "I always lie," is untrue. This may or may not present difficulties. (Consider the difference between the statements "I always lie" and "I always tell the truth"—see the Exercises.)

Turing machines are self-referential. Note that a Turing machine can take any string (over the proper alphabet) as input. All Turing machines and all inputs can be encoded as strings over {0, 1}. The self-referential property will arise when a Turing machine is run on its own description, that is, its own encoding.

Recall that the universal Turing machine takes a machine encoded over {0, 1} and the input to the machine (also encoded) and emulates the actions of the machine on the string. These two strings can be identical, so the universal Turing machine is given a machine M encoded and emulates M on the string that is also the encoding of M.

This may sound odd, since in computer science we rarely run a program on itself. However, we could do so. Consider a C++ program that takes some input. The program is a string of 0's and 1's, of course, once it is stored in a computer. Its input, whether integers, strings, or files—or some combination thereof—is also a string of 0's and 1's. We can use the program string as the input string. It might be too long (in which case the program will only use part of it) or too short (and then the program will probably sit and wait for more input, or will do whatever it has been written to do on insufficient input).

Other problems—and other paradoxes—also present problems when they are self-referential. The whole theory of sets was revised by the difficulties presented by Russell's paradox. One way to understand self-reference is to look at some examples.

6.1.1.1 Problem 1: The Barber

There is a town on an island in the Mediterranean where there is a barber who shaves everyone who does not shave themselves. This means that: (1) everyone shaves—regardless of age and gender, and (2) the barber shaves a person if and only if that person does not shave himself or herself.

This town with this barber B cannot exist as stated. The contradiction is in who shaves the barber? According to the statement, if B does not shave himself, then the barber B should indeed do the shaving. And if B does shave himself, that puts him in the category of people that the barber does not shave.

There is no way around this problem by adding women, children, or robots. The only way around it is to change the wording so that the "B shaves P iff P does not shave himself" only applies to others, and not to the barber.

6.1.1.2 Problem 2: Grelling's Paradox

An adjective is said to be *autological* if the property denoted by the adjective is possessed by the adjective itself, and *heterological* if the property denoted by the adjective is not possessed by the adjective itself. For example, polysyllabic is autological since it has five syllables, and monosyllabic is heterological since it does not have one syllable. The word *black* written in standard black type font is autological; the word *green* is heterological unless we print it in a green font. Question: Is the word *heterological* heterological?

6.1.1.3 Problem 3: Russell's Paradox

Bertrand Russell was deeply involved in formalizing set theory. One concern was sets that are described by some property.

$$S = \{x \text{ from some universe} \mid x \text{ has property } P\}$$

Some sets—most sets—are not elements of themselves. The set $C = \{x \mid x \text{ is a prime integer}\}$ is a set, not a prime integer, so $C \notin C$. However, $A = \{x \mid x \text{ is an infinite set}\}$ is an infinite set, so that $A \in A$.

Now consider $Z = \{X \mid X \text{ is a set and } X \notin X\}$. If $Z \in Z$, then $Z \notin Z$, and if $Z \notin Z$, then $Z \in Z$. (Why?) Neither of these can be true. Discovery and resolution of this paradox made significant changes to set theory, including modifications to how sets can be constructed.

6.2 SELF-REFERENCE AND SELF-CONTRADICTION IN COMPUTER PROGRAMS

There are interesting problems that come up in the analysis of a computer program.

6.2.1 Problem 1: The "Hello World" Writing Detector Program

Some very simple programs write "Hello World." Some others may also write it, but it may not be so easy to decide whether or not they do.

A program that obviously(?) writes "Hello World" is shown in Figure 6.1. Another less obvious program is shown in Figure 6.2. A third program that may—or may not—write "Hello World" appears in Figure 6.3 (interesting since it is unknown if there are any odd "perfect" numbers, but if one exists, it is greater than 10^{300}).

The task (which will turn out to be impossible): Write a program Hello World Detector that takes as input: (1) the code someone has written, and

```
#include <iostream>
int main()
{   cout << "Hello, world!" << endl;
    return 0;
}
```

FIGURE 6.1 Writing "Hello World."

```
#include "stdio.h"                                  [Source: International Obfuscated C Code Contest, 1986, Bruce Holloway]
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
   int a,b,c,d,o[k],n=(int)s;
   if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
   else switch(m1-=h){
      case f:
         a=(b=(c=(d=g)<<g)<<g)<<g;
         return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
      case h:
         for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
      case g:
         if(n<h)return(g);
         if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
         else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
         if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
         return(o[b-g]%n+k-h);
      default:
         if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
         for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
      }
}
```

FIGURE 6.2 Writing "Hello World."

(2) the input to that program (if the program takes input). Determine either: that (a) yes, this program on this input will sometime write "Hello World," or (b) no, this program on this input will never write "Hello World."

You cannot solve this task by running the code on its input—that might work, but the only time you have a guaranteed answer is when you see the "Hello World" appear. And, to be a correct program or algorithm, it must work correctly for all inputs. If it executes the program and sees the words "Hello World" as output, that would be fine. But if no output appears—after a minute, an hour, a day, a week, whatever—it could not say that the code will never write "Hello World." The test program might be just getting to the point where it will write the phrase.

```
#include <iostream>
int main(){
// n is 'perfect' if it is the sum of its divisors (excluding
itself)
// for example, 6 = 1+2+3,  28 = 14+7+4+2+1

{pseudocode – since n must be stored as an arbitrarily
large integer, possibly as a linked list of  n's digits or of
sections of n or …)

 extremely_large_integer n = 1+ 10^300;

 if (n is perfect)
                {cout << "Hello, world!" << endl;
                return 0;
                }
 else n+=2;
}
```

FIGURE 6.3 Writing "Hello World."

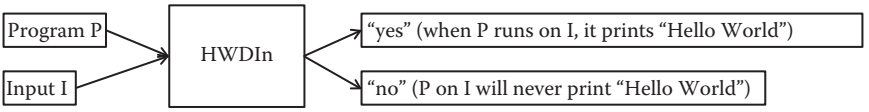


FIGURE 6.4 Action of HWDIn.

To show why the Hello World Writing Detector program cannot be written, we will assume that it can. Then, by using self-reference, we arrive at a contradiction, and we will have proved that the program cannot exist.

Thus, assuming we can write the Writes Hello World Detector for Programs That Take Input program (call it HWDIn for short), we have the machine shown in Figure 6.4.

If we could write HWDIn, it would also work on programs without input. For example, the first and third programs above take no input. The second does take input, but one could hard code values for its input into the program without changing how it runs (except for the inconvenience of rewriting and recompiling for new input values).

So assuming that HWDIn can be written, so can this second Hello World Detector (named simply, HWD), which works on programs with no input. HWD is HWDIn, but any input has been hard coded into HWD. It is shown in Figure 6.5.

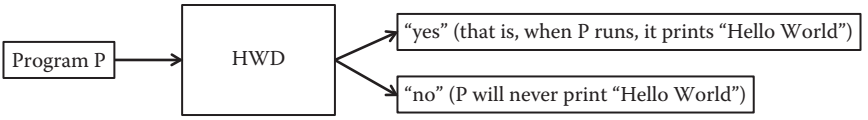


FIGURE 6.5 Action of HWD.

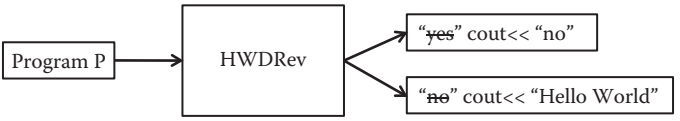


FIGURE 6.6 Action of HWDRev.

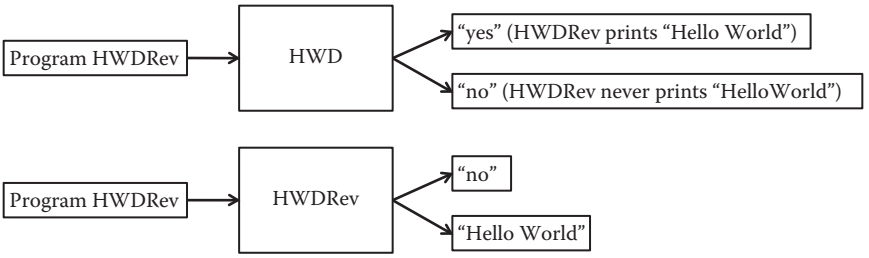


FIGURE 6.7 Action of HWDRev on itself.

Now the only two possible outputs from HWD are yes and no. Suppose that we change each occurrence of the string yes in an output statement to no, and every occurrence of no to "Hello World." Call this new program HWD Revised or HWDRev for short. It is shown in Figure 6.6.

This is (as in all of these cases) a problem of self-reference. What would happen if we were to run HWDRev on itself? We do that by sending HWDRev to HWD (see Figure 6.7). And what would the original HWD say about HWDRev? If HWD were to say yes, then HWDRev should print "Hello World." But, the way it is written, the yes was changed into a no.

And, if HWDRev never writes the phrase, then HWD says no, but by the way the code was altered, HWDRev now writes "Hello World."

In short, HWDRev writes "Hello World" if and only if it does not write "Hello World." This cannot happen! So HWDRev cannot exist—and neither can HWDIn or HWD. All this is summarized in Figure 6.8.

6.2.2 Problem 2: The Infinite Loop Detector Program

Write a program P whose input is a program S from a C++ programming class, along with the input file I for those programs. P's task is to take the

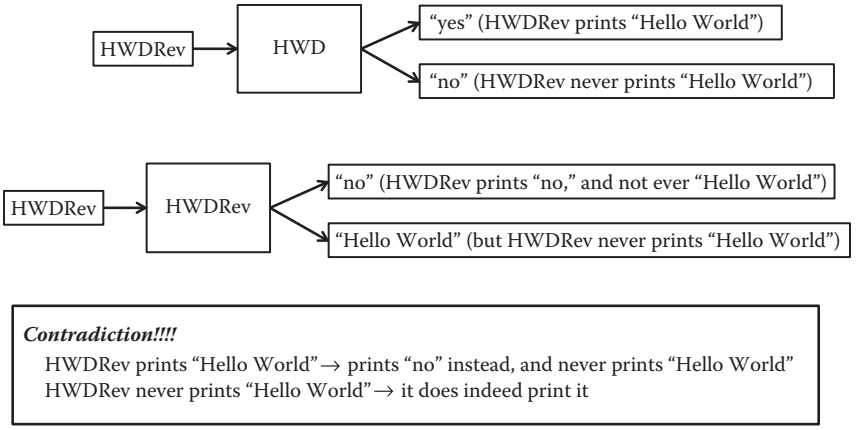


FIGURE 6.8 Summary.

input I and the student program S and determine whether S will go into an infinite loop on I, or whether S will halt on I in a finite amount of time. P is to run on a computer "at least as powerful as a Turing machine," which, since C++ is considerably more powerful than the Turing machine (TM) instruction set, means that it has as much memory as it needs for the problem. This is not quite true. No real computer has infinite memory—and a Turing machine does. However, if one is smart enough and rich enough, one can use network storage, multiple disks, and other external storage so that in practice, memory is essentially unlimited.

One might think that if one were a good enough programmer, one could write the program P. However, that is impossible—program P cannot exist.

Note that P must work without running S on I. It could try doing that—run S on I—but then what? If S on I halts within some reasonable time (say 10 minutes), then P can answer "yes, halts." But after 10 minutes, if S on I is still running, P cannot know whether it will stop soon or not. Similarly, after 100 minutes or 100 years or 100 millennia, the only thing known is that P has not yet terminated.

Why can't P exist? First, note that input files and programs are both strings of binary digits—so there is no real distinction between "program" and "input data." One of the possible inputs to a program is a data file with the same string of digits as the program itself. Thus, we can say that you can run a student program S on itself ("run S on S"). This "trick," which is not really a trick, is needed to end up with the contradiction via self-reference.

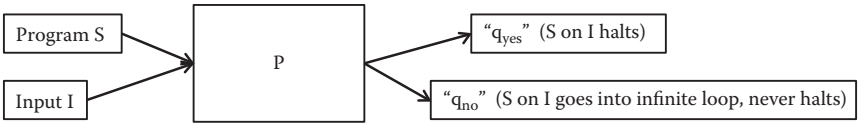


FIGURE 6.9 Action of infinite loop detector P.

If P exists, it must take any program S with input I and announce "yes, halts" or "no, never halts." It might do this announcement by writing one of the two phrases to the screen. This is shown in Figure 6.9.

We can easily change P to P_Rev (read as "P revised") with the following three changes.

1. Use self as input: Instead of inputs S and I, now P_Rev takes only one input S and will decide what S does on S.
2. Turn "yes" into a loop: If P would write "yes, halts," then P _Rev goes into an infinite loop.
3. Turn "no" into "yes": If P would write "no, never will halt," then P_ Rev writes "yes, halts" and stops.

Imagine how to do this: do a pattern search through P. Any statement of the form

```
writeln ("yes, halts")
```

is replaced by

```
while true do x := 1
```

And any line of the form

```
writeln ("no, never will halt")
```

is replaced by

```
writeln("yes, halts"); exit(0);
```

This revised program P_Rev works on all possible input programs, since P does. In particular, one can ask what P _Rev does when fed its own code.

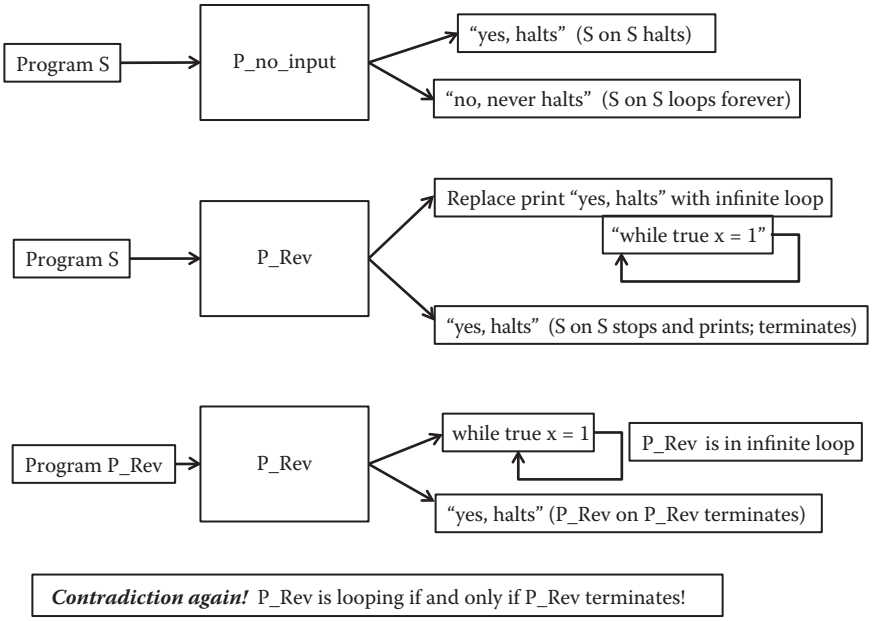


FIGURE 6.10 P revised with two simple revisions.

Suppose *P _Rev halts on P _Rev*. That means that when we sent P_Rev into P, P analyzed what P_Rev would do on P_Rev and said "stops," but in the action of P_Rev this has become the "while true x=1" infinite loop and *P_Rev loops on P_Rev*.

Suppose *P _Rev never halts on P _Rev*. That means that P would have known this but the revision of P_Rev is written "yes, halts" and stops, that is, *P _Rev halts on input P_Rev*.

In other words, P_Rev halts on itself exactly when it does not halt on itself. That cannot happen—so P_Rev, and thus P, cannot exist. Again, this is summarized in Figure 6.10.

This is a version of the halting problem—answering the question whether a given Turing machine, or a given program running on some computer, will ever halt. And, as the above discussion shows, this is undecidable.

6.3 CARDINALITY OF THE SET OF ALL LANGUAGES OVER AN ALPHABET

We can arrive at proofs of some facts about the relationship of languages and Turing machines by considering the cardinality of the set of all languages, and the cardinality of the set of all Turing machines. This is definitely worth doing.

	Column headers: Strings over {0,1} in lexicographic order								
	ε	0	1	00	01	10	11	000	...
L_{all}	1	1	1	1	1	1	1	1	...
{w = 0*}	1	1	0	1	0	0	0	1	...
{w = 1*0*}	1	1	1	1	0	1	1	1	...
Lε	1	0	0	0	0	0	0	0	...
L_{even_length}	1	0	0	1	1	1	1	0	...
{w ends in 1 }	0	0	1	0	1	0	1	0	...
...	...	...	...	...	...	...	...	...	...
Row labels: Languages over {0,1}—in no particular order									

FIGURE 6.11 Characteristic vectors for some languages over {0, 1}.

Recall that the set of natural numbers **N** has cardinality $\aleph_0$, and that the set of infinite sequences over {0, 1} has the much larger cardinality $\aleph_1$.

Let Σ be any alphabet. By listing the strings in standard lexicographical order (v < w if v is shorter than w; if v, w have the same length, then v < w if it precedes it in the dictionary), we have an easy one-to-one correspondence between the strings over Σ and the natural numbers. We have seen this before; we know that $|\Sigma^*| = \aleph_0$.

$$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$$
$$= \{1st, 2nd, 3rd, 4th, 5th, \ldots\}$$

Take any language L over {0, 1}. It has a characteristic vector that is 0 in the ith position if the ith string is not in L, and is 1 in the ith position if the ith string is in L. See Figure 6.11 for the characteristic vectors of some languages.

These characteristic vectors make a one-to-one correspondence between the set of languages over Σ^* (the subsets of Σ^*) and set of infinite sequences over {0, 1}. But, we know that there are $\aleph_1$ infinite sequences over {0, 1}, so of course, there are $\aleph_1$ languages over {0, 1}.

This means that there are $\aleph_1$, or uncountably many languages, over Σ, for any alphabet Σ.

6.4 CARDINALITY OF THE SET OF ALL TURING MACHINES

Each Turing machine has an encoding as a finite string of 0's and 1's. There are many different ways to do this encoding; the scheme from the previous chapter is one way. Note that each Turing machine is then a positive natural number represented in binary.

1	↔	no-move-TM
10	↔	no-move-TM
11	↔	no-move-TM
100	↔	no-move-TM
...		
some integer	↔	TM that accepts strings beginning with '00'
...		
some other integer	↔	TM that accepts strings of odd length
...		
another integer	↔	TM that accepts strings {n:n is prime in binary}
...		
...		
...		

FIGURE 6.12 Enumeration of the Turing machines.

Not all positive natural numbers correspond to encodings of Turing machines. For example, the substring 11111 never appears in any encoding using the scheme from Chapter 1, so numbers such as 11111, 11111011, and 11111011111 do not correspond to any Turing machine. This is not really a problem—it would only mean that there are more positive integers than there are Turing machines. Another simple solution is to say that each number *does* correspond to some Turing machine, but if it cannot be decoded into a Turing machine, then we say that it corresponds to a Turing machine with no moves at all (Turing machine that rejects all inputs). Thus, 11111, 11111011, and 11111011111—and all the other non-Turing machine strings—correspond to this no-move TM.

We can enumerate the Turing machines—take each encoding, and put them in order. See Figure 6.12.

This means that there are $\aleph_0$ Turing machines. There are, however, $\aleph_1$ languages. This means that there are *countably* many Turing machines—including all the deciding and accepting Turing machines—and *uncountably* many languages.

In short, there are far more languages than Turing machines—in fact, not just more, but uncountably more. By this argument, there are languages—infinitely many of them—that are not decided, or even accepted, by a Turing machine.

This is what we wanted to prove, but it is an existence proof only. In the next section, we will actually construct an example of a language not decided by any Turing machine, and another language not even accepted by any Turing machine.

We restate this as a theorem, to stress its importance:

Theorem 6.1

There are many (uncountably many) languages that are not decided—or accepted—by any Turing machine. ■

6.5 CONSTRUCTION OF THE UNDECIDABLE LANGUAGE $ACCEPT_{TM}$

Now we know undecidable languages exist (by counting arguments), but we have not yet produced an example. We have shown that there are uncountably many languages and only countably many decidable languages (since countable many Turing machines), so with an uncountable number of undecidable languages, should it not be easy to actually display one? Well, not quite. Essentially any description of a language that you are likely to come up with will have a deciding Turing machine. We do want to produce an undecidable language.

This will be done by constructing a language that is not decidable. This is essentially the same proof (by contradiction) as the proofs that writing "Hello World" or detecting infinite loops are not decidable.

Let $ACCEPT_{TM}$ = {M, w: M is the encoding of a Turing machine, w is the encoding of a string, and M accepts w}

Theorem 6.2

The language $ACCEPT_{TM}$ is not decided by any Turing machine.

Proof

By contradiction, suppose $ACCEPT_{TM}$ is decided by a Turing machine M_{ACCEPT}. That is, M_{ACCEPT} takes an encoded Turing machine M and string w and goes to q_{accept} exactly when M accepts w; otherwise, it will terminate in q_{reject}. Of course, it terminates in one or the other since it is a *deciding* TM. This is shown in the first part of Figure 6.13.

Now, we modify M_{ACCEPT} so that it takes only machines as input—that is, its input is an encoded machine M. M_{ACCEPT} will tell if the machine M accepts its own description.

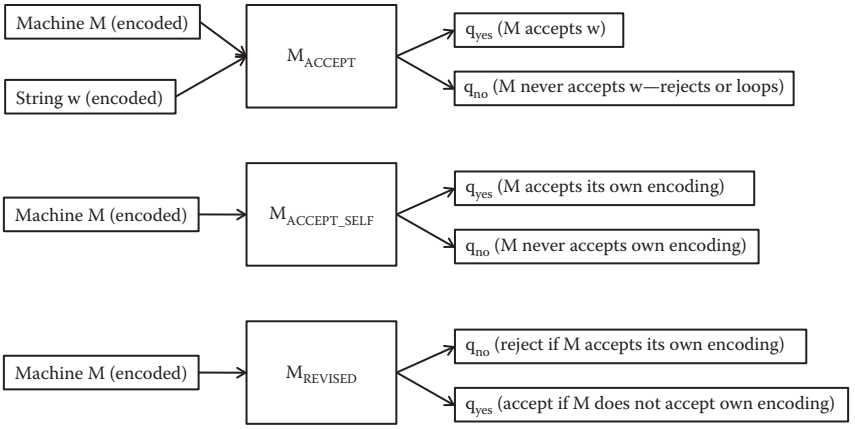


FIGURE 6.13 Deciding Turing machine M_{ACCEPT} for ACCEPT (and two modifications).

One more simple transformation is in order—interchange the accepting and rejecting states. Call this machine $M_{REVISED}$, or M_R for short.

Now, when we use self-reference (that is, send the encoding of M_R into M_R), we get a contradiction, or impossible situation. M_R should go to q_{reject} (reject M_R) exactly when M_R accepts M_R.

And, it goes to q_{accept} exactly when M_R does not accept M_R. This cannot happen, but it is a direct result of the one assumption that M_{ACCEPT} exists as a deciding Turing machine. The simple changes made to M_{ACCEPT} cannot be the problem. Thus, the contradiction must be in the existence of the deciding Turing machine M_{ACCEPT}. ■

The important point here is that: $ACCEPT_{TM}$ is not *decidable*; however, it is Turing *acceptable* (by the universal Turing machine).

The above proof, with its contradiction, only works for deciding Turing machines. If not all strings end up in one of q_{accept} and q_{reject}, there is no contradiction in interchanging these states. The strings that did not end up in either one still do not. What gives the contradiction is that since we are assuming a deciding TM, M_R on M_R only has the choice of q_{accept} and q_{reject}, and both of these are impossible.

Corollary 6.3

$\overline{ACCEPT_{TM}}$ is not accepted by any Turing machine.

Proof

$ACCEPT_{TM}$ has an accepting Turing machine (essentially the universal Turing machine). If its complement were also accepted by some other Turing machine, then $ACCEPT_{TM}$ would be decidable. ■

EXERCISES

1. Prove that the following sets have the same cardinalities.
   a. The natural numbers (1, 2, 3, …) and the odd natural numbers (1, 3, 5, …)
   b. The natural numbers and the rational numbers (n/m, where n, m are natural numbers m ≠ 0)
2.
   a. Consider the sentence "This sentence is false." Is this sentence true or false?
   b. Consider the number N = *the smallest positive integer not nameable in under 11 words*—the smallest element of the set of numbers that take many words to write out in English (such as "eight hundred twenty-two million three hundred forty-six thousand two hundred fifty-one"). Is there a problem here?
   c. Consider the sentences "I always lie" and "I always tell the truth." Are these sentences self-referential? Are there problems with either one?
   d. What do parts a and b have to do with this course?
3. Look up Russell's paradox. How was it resolved in set theory?
4.
   a. Consider the **intermediate value theorem** from calculus.

      If a real valued function *f* is continuous on the closed interval [a, b] and if N is a number strictly between *f*(a) and *f*(b), then there exists a number c in (a, b) such that *f*(c) = N.

Is this a constructive or existential theorem?

What (if any) would be the advantages of a constructive theorem?

What (if any) would be the advantages of an existential theorem?

b. Consider this quote from an article on routing in networks: "We give a nonconstructive proof of the existence of linear time algorithms that check whether a desired routing scheme exists for a given graph."*

Why would one prefer or not prefer a constructive proof?

5. Definition: An algebraic number is any number that satisfies an equation of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

where $n > 0$, every a_i is an integer, and a_n is nonzero

a. Show that the number $\sqrt{2}$ and the complex number $i = \sqrt{-1}$ are both algebraic.

b. Show that the cardinality of the algebraic numbers is countable.

c. What does this say about the existence of nonalgebraic numbers? Can you produce a nonalgebraic number?

6. Let $F = \{f: f$ is a function from the reals ($\mathbf{R}$) to the reals$\}$. Show that the cardinality of F is greater than $\aleph_1$.

* Hans L. Bodlaender, and Jan van Leeuwen. 1997. On Interval Routing Schemes and Treewidth. *Information and Computation*, 139, 92–109.

CHAPTER 7

Undecidability and Reducibility

7.1 UNDECIDABLE PROBLEMS: OTHER EXAMPLES

Once we have the first undecidable problem (whether the Turing machine M accepts the string w), it is generally not necessary to use contradiction or the property of being self-referential again. Because this language is the basis for so much of what follows, we have given it a special name ($ACCEPT_{TM}$) and will often be referring back to it. Other texts have also given it special names; Sipser (2006) calls it ATM for acceptance-by-Turing-machine; Cohen (1996) calls it Alan in honor of Alan Matheson Turing.

$$ACCEPT_{TM} = \{M, w: M \text{ is a Turing machine that accepts } w\}$$

(where by M, w we might more properly say "encoding of M, encoding of w").

This is something that occurs several times in theoretical computer science—the first problem is difficult to prove, and after that, one can use the first problem and the idea of reducibility to prove other problems are in the same category.

The basic idea of reducibility is simple: if one has two questions, and if one knows the answer to question 1 by the answer to question 2, you do not need both answers. Answering question 2 will answer both of them. In some sense, question 2 must be harder (or at least as hard) as question 1, since an answer to it answers both.

Example

Question 1: Given a quadrilateral, is it a square (all sides and angles equal)?
Question 2: Given a polygon with $n \geq 3$ sides, are all sides the same length—is it equilateral and equiangular?

Clearly, if you had an answer to question 2, it would also answer question 1.

Example

Suppose there is a person, Professor Jones, who carries an umbrella exactly when it is raining (so rain → umbrella; no rain (even if storms are forecast) → no umbrella).

Question 1: Is it raining?
Question 2: Is Professor Jones carrying an umbrella?

In this (idealized) case, if you are sitting in a room with no windows and Professor Jones walks in, you can tell whether it is raining outside by noting whether Professor Jones came in with an umbrella or not.

Decision problem 1: At time t, is it raining?
Decision problem 2: At time t, is person X carrying an umbrella?

It is certainly debatable whether this situation can be called real life; there are very few people as this. Professor Smith, who carries an umbrella when it is raining (or if he thinks it will rain), would not be acceptable in place of Professor Jones.

Example

Remote sensors are another example. A warning light for the temperature in a motor behaves so that:

Temperature too high → warning light on

Temperature not too high → warning light not on

Question (or decision problem) 1: Is the temperature too high?
Question (or decision problem) 2: Is the warning light on?

Example

Sam sees some movies; Ann is a movie critic and sees every movie that comes out. Moreover, her taste exactly agrees with Sam's. If Sam likes a movie, so does Ann; if Sam does not like a movie, Ann does not like it either.

Question 1: We are told that Sam saw movie X. Did he like it (yes/no)?
Question 2: What did Ann think of movie X (Does she like it? yes/no)?

When is a problem harder (or more precisely, as hard or harder) than some other problem? We would surely say that Y is harder than X if knowing the solution to Y would also give us the answer (yes or no) to X.

For example, in mathematics, you might want to know if a particular third degree polynomial has an integer root. Your book has an algorithm that works for all polynomials of degree up to 4. So this algorithm will also work for your third degree polynomial. That is, obviously, the problem "integer root of polynomial of degree up to 4" is harder than the same problem with degree up to 3.

And clearly, an answer to a question about all polygons would also answer a question about quadrilaterals.

The key here is that there is a mapping from one problem to another, and that yes answers of the first map to yes answers of the second, while no's of the first become no's of the second. Thus, instead of asking the question about the original problem, we only need to ask it for the second. Yes of the second means yes of the first, no of the second means no of the first. Answering the second question gives you a two-for-one bonus; you get answers to both problems.

Note that an answer to the first problem gives you no bonus; you cannot (necessarily) get all the answers to problem 2 from problem 1. You *can* get some answers to problem 2—a yes/no of problem 1 goes to a yes/no of problem 2, so by following the mapping, you get an answer for *that* instance of problem 2. If the mapping were *onto*, that is, if every instance of problem 2 were an image of something from problem 1, and the mapping P_1 to P_2 had an inverse mapping, then the problems would be equivalent (and answering either would answer both).

7.2 REDUCIBILITY

Definition 7.1

Suppose that P_1 and P_2 are decision problems. Then P_1 is **reducible** to P_2, written $P_1 \leq P_2$, if there is a mapping f from all possible instances (yes or no) of P_1 to possible instances of P_2 such that:

1. f is computable (that is, some Turing machine—or transducer—can compute it).
2. If X is a yes instance of P_1, then $f(X)$ is a yes instance of P_2.
3. If X is a no instance of P_1, then $f(X)$ is a no instance of P_2. ■

Note that 3 can be rewritten equivalently as:

3′. If $f(X)$ is a yes instance of P_2, then X is a yes instance of P_1

since 3′ is the contrapositive of 3. That is, "no goes to no" is equivalent to "yes comes from yes." This second form is often easier to use in proofs.

This mapping can be visualized; see Figure 7.1.

Conditions 2 and 3 say that answering P_2 will also answer P_1. That is, if one looks at $f(X)$ and answers if it is a yes (or no) instance of P_2, then it immediately follows that X was a yes (or no) instance of P_1.

Be careful; as was said before, nothing is said that answering P_1 will also answer P_2. The mapping must be defined on all instances of P_1—both

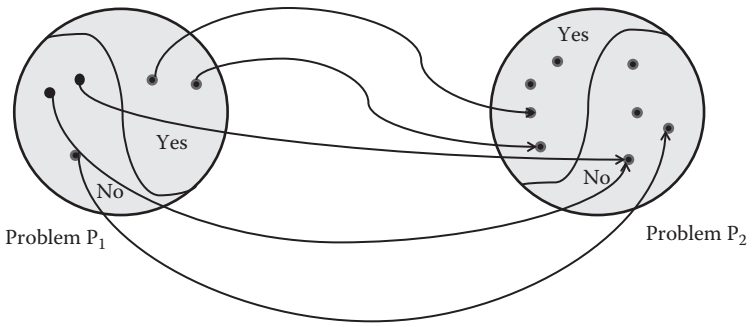


FIGURE 7.1 Condition 1 does not impose any true restrictions on the problem, since anything that one can imagine as a well-defined function of numbers, strings, and so on, is computable. We make no restriction here on how long it takes for the transducer to do this transformation, only that it can be done.

yes and no—but there may be many instances of P_2 that are not related to anything in P_1.

Note on the name for ≤: Despite the best efforts of Donald Knuth, who did succeed in standardizing the terms **NP**-hard and **NP**-complete (see the next chapter), there are various terms used for what we have defined as reducible. Emil Post, in 1944, called this "many-one reducible" since the mapping may take many yes instances from P_1 to a single yes instance of P_2. Some authors use "mapping reducible"; others do not use a function. Instead, they speak of constructing a yes/no instance of P_2 from an instance of P_1.

There are other kinds of reducibility. If we add the requirement that the function be calculated in polynomial time, we get "polynomial time mapping reducible." We will find this definition very important in Chapters 8 through 10.

Definition 7.2

A decision problem P_1 is **polynomially reducible** to decision problem P_2, written $P_1 \leq_{\mathbf{P}} P_2$, if there is a mapping f from possible instances (yes or no) of P_1 to possible instances of P_2 such that:

1. For any yes/no instance X of P_1, $f(X)$ can be computed by a Turing machine (transducer) in time bounded by a polynomial $p(|X|)$, $|X|$ = length of the input.
2. If X is a yes instance of P_1, then $f(X)$ is a yes instance of P_2.
3. If X is a no instance of P_1, then $f(X)$ is a no instance of P_2. ■

The only difference between definitions 7.1 and 7.2 is the addition of the restriction that the function can be computed in polynomial time.

Turing reducibility is a slightly different and more general concept in which the answer to the right-hand problem can be used to answer the left-hand problem. It is not used in this text. *Log space reducibility* is another different and more restrictive reducibility in which the function f is only allowed to use a work space that is bounded by a constant times the log of the input size.

It would be tempting to think that answering P_1 would also answer P_2. That would be the case for possible yes or no instances that are in the range of f—that is, they come from something in P_1. Thus, in the two instances

above, an answer to temperature too high will answer whether the warning light is on. An answer to raining/not raining will answer whether Professor Jones has an umbrella or not. In these two cases, the mapping is onto, or surjective, but the range is very small—{warning light on, warning light off} and {Jones has umbrella, Jones does not have umbrella}.

The mapping from the movies Sam sees to those that Ann sees is not onto. We need to assume Ann sees any movie that Sam has seen (otherwise, the domain of the function is too small), but if the movie is one Sam has not seen, we cannot tell what Ann thought of the movie.

Figure 7.1 demonstrates these ideas. Every member of P_1 must be the end of an arrow—it must be mapped to an element of P_2, with, of course, yes going to yes and no going to no. However, the function is not necessarily onto, and some elements of P_2 are not the ends of any arrow.

7.3 REDUCIBILITY AND LANGUAGE PROPERTIES

If you know something about one language L_1 and know how it is related to another language L_2 by reducibility, then some properties follow easily. These include:

Theorem 7.3

If $L_1 \leq L_2$ and L_2 is decidable, then so is L_1.

Theorem 7.4

If $L_1 \leq L_2$ and L_1 is undecidable, then so is L_2.

Theorem 7.5

If $L_1 \leq L_2$, then $\overline{L_1} \leq \overline{L_2}$.

Theorem 7.6

If $L_1 \leq L_2$ and $L_2 \leq L_3$, then $L_1 \leq L_3$.

Proof

Exercises. ■

7.4 REDUCIBILITY TO SHOW UNDECIDABILITY

So far, it has been proved that the one problem $ACCEPT_{TM}$ = {<M, w> | M and w are encoded Turing machines and strings, and M accepts w} is undecidable. For a different problem, we will be able to show P is undecidable by finding a mapping from $ACCEPT_{TM}$ to P.

There is a fairly standard procedure for building this mapping from $ACCEPT_{TM}$ to P.

1. Build a yes/no instance L′ = L(M′) of P. M′ will be a Turing machine with input x′ over some alphabet Σ′.

   How we do this building and what Σ′ is may vary. One thing that almost never varies though is that M′ has M and w built into it—that is, M′ depends on M and w, or M′ has M and w *hard-coded* into it. You can think of M and w as built into the state information of M′, or as written on two internal tapes of M′, or simply as part of M′. How one actually would build M and w into M′ is not important. It would take a lot of time and effort to actually do this, but it is not hard to see that it can be done.

2. The trick is to build M′ so that the language it accepts is one of two possible different languages, one that is a yes instance of P and one that is a no instance.

   Whether it is a yes or no instance should depend on whether M accepts w or not.

   There are times when M′ must look at its input and possibly accept before it runs M on w. However, that is not the usual case.

3. Usual way to build M′:

   First, run M on w. In doing this, three things could happen: (a) M accepts w, (b) M rejects w, or (c) M is looping on w, and has not—and will not—ever accept or reject.

   What M′ will do may vary, but usually:

In case a, M′ now—for the first time—looks at its input x′ and accepts if x′ is of some form. It might just accept x′ no matter what it is—in this case, $L(M') = \Sigma'^*$.

In case b, M′ usually goes to its own reject state; so $L(M') = \varnothing$.

In case c, we do not have to specify what M′ does, since it has not—and will not ever—get out of its first step, which is running M on w. Since it will not ever finish this, it can never even inspect its own input, and $L(M') = \varnothing$.

Theorem 7.7

Given a Turing machine, it is undecidable whether the Turing accepts the empty string ε.

Proof

This follows the classic pattern. Start with a Turing machine M and its input w. Now, we want to build a new M′ so that

M accepts w ⇒ M′ accepts ε (and maybe other strings too).
M does not accept w ⇒ M′ does not accept ε.

To do this, we build an M′ that has M and w built into it.
Action of M′ on its input x:

Ignore x while M′ runs M on w.
If M accepts w, M′ enters its own accept state $q_{M'\text{-accept}}$ (so accepts x).
If M rejects w, M′ rejects its input, entering its reject state.

(Should M be looping on w, we do not have to state what to do. M′ will wait for M to finish on w, and that will never occur. So M′ is simply waiting, waiting, waiting….)

Now the language of M′ depends on what M does on w:

M accepts w ⇒ $L(M') = \Sigma^*$ ⇒ M′ accepts ε.
M does not accept w ⇒ $L(M') = \varnothing$ ⇒ M′ does not accept ε. ■

Theorem 7.8

Given a Turing machine, it is undecidable whether the Turing machine accepts any string at all (whether its language is nonempty).

Proof

Exactly the same construction of M′ from M will work.

M accepts w $\Rightarrow$ L(M′) = Σ^* $\Rightarrow$ L(M′) $\neq \varnothing$.
M does not accept w $\Rightarrow$ L(M′) = $\varnothing$. ■

Theorem 7.9

Given a Turing machine, it is undecidable whether the Turing machine accepts all strings (whether its language is Σ^*).

Proof

Once more, exactly the same construction of M′ from M will work.

M accepts w $\Rightarrow$ L(M′) = Σ^* $\Rightarrow$ M′ accepts all strings.
M does not accept w $\Rightarrow$ L(M′) = $\varnothing$ $\Rightarrow$ M′ does not accept all strings. ■

Theorem 7.10

Given a Turing machine and tape symbol τ, it is undecidable whether the Turing machine ever writes τ during computation on input string z.

Proof

A similar construction will work. We will build M′ with input x; it will write τ if and only if M accepts w.

First step of M′: Run M on w.

If M accepts w, M′ looks at its input. If x = z, then M′ writes τ on its tape. (If x is not z, M′ can do whatever it wants: it can simply halt and reject or whatever.)

If M does not accept w, M′ goes to its reject state without writing anything on the tape.

If M is looping on w, so is M′, simply waiting or looping until M finishes.

M accepts w ⇒ M′ on z writes τ.

M does not accept w ⇒ M′ never writes τ on z (or any other string). ■

Theorem 7.11

Given a Turing machine M and a string w, it is undecidable whether M halts on w.

Note: This is the **Halting problem**, and the well-known language $HALT_{TM}$ can be defined as

$$HALT_{TM} = \{M, w: M \text{ halts on } w\}$$

We say that M *halts* on w if it enters either of the states q_{accept} or q_{reject}. These are the only two states without moves out of the state; every other configuration has a next configuration. If M on w does not halt, we say that it loops, although there are many ways in which it can loop (cycle back and forth between adjacent tape squares, move right forever, etc.).

Proof

We want to show $ACCEPT_{TM} \leq HALT_{TM}$. In this case, we will do it slightly differently, but again, we build a machine M′ so that M′ halts on w if and only if M accepts w. But here, we can do that by letting M′ be a simple modification of M. Take M and change its state q_{reject} into a new state q_{new}. This is a simple cut-and-paste; every occurrence of q_{reject} in the delta function is replaced by q_{new}. In q_{new}, we add moves so that on every symbol, M′ writes a blank and moves right. (Technically, we require that M′ have its own state q_{reject} so we can add that state, but add no transitions to it, so it is unreachable.)

Now, if M accepts w, M′ halts on w. And if M does not accept w, it must either loop on w or reject it—and in either case, M′ does not halt on w. Thus, yes goes to yes, no goes to no, and answering whether M′ halts on w also answers if M accepts w. ■

7.5 RICE'S THEOREM (A SUPER-THEOREM)

In one sense, one theorem—Rice's theorem—makes all the above theorems unnecessary. What it says, in brief, is that essentially every question about what language a Turing machine accepts, what states it enters, or what symbols it writes—these are all undecidable. Thus, we could simply have begun with Rice's theorem. Then, all of the above theorems would follow immediately.

Theorem 7.12 (Rice's Theorem)

A question about Turing machine languages is said to be nontrivial if it is true for some, but not all, Turing machine languages. Any nontrivial question about languages accepted by a Turing machine is undecidable.

Note: It is actually hard to think of trivial Turing machine questions. Anything interesting ("is finite," "has at least five strings," "contains the empty string") is nontrivial—some languages are yes instances, some are no instances. Only questions such as "is over some alphabet," "contains zero or more strings," "is a subset of itself" are trivial—the answer is yes (or no) for all languages.

Proof

We need two languages, one that is a yes instance of this question and one that is a no instance. Since the property is nontrivial, there must be both yes and no examples.

Consider the empty language $\varnothing$. Suppose it does not have the property (if it does, we need to repeat this proof with modification; see Exercises). Let L be a language that does have the property. L is a Turing machine (TM) language, so it is accepted by a TM M_L.

Now we need to build M′, again with a given M and w built into it.

M′ works as follows on its input x:

Run M on w.
If M accepts w, run M_L on x and accept if M_L does.
If M rejects w, go to the reject state.

Now:

M accepts w $\Rightarrow$ L(M′) = L(M_L) $\Rightarrow$ L(M′) is yes instance.
M does not accept w $\Rightarrow$ L(M′) =$\varnothing$ $\Rightarrow$ L(M′) is no instance.

Thus, if we can answer whether M′ has the property or not, we can answer whether M accepts w. This last question is undecidable. ■

In other words, all nontrivial questions about the language accepted by Turing machines are undecidable. There are not any interesting trivial questions—that would be a contradiction. "Is composed of strings" is a trivial question. On the other hand, other questions ("Is the language finite?" "Does it contain some string of length 5?" "Does it contain all strings with prime length?") are nontrivial questions.

Questions about the Turing machine itself are often—but not always—undecidable. For example, consider the following questions about a Turing machine:

Decidable: "Does the machine have a state q_{271}"

"Does it enter q_{271} on the empty string in less than 572912 steps"

Questions are decidable if they can be answered by inspection of the transition function, or if they ask about a limited operation of the Turing machine.

Undecidable: "enters q_{271} on the empty string"

"enters q_{271} on any string whatsoever"

Questions are undecidable if they relate to what strings the machine accepts. The questions above would be more obviously undecidable if q_{271} were replaced by q_{accept}.

Undecidable does not mean "never has an answer." For some machines, one could look at the transition function and perhaps note that q_{271} never appears on the right-hand side of any move. Then, it would never be possible to enter that state. What *undecidable* does mean is that there is no way of answering the question for all possible legal inputs. More about this in the next section.

7.6 UNDECIDABILITY: WHAT DOES IT MEAN?

A question is undecidable if there is no Turing machine that can answer the question and that halts on all inputs. Or, equivalently, a question is undecidable if it has no algorithm.

Again, undecidable does not mean that the question can never be answered. Answers can be found in some, maybe even many, cases.

What it does mean is that there is absolutely no way to write a program that will answer the question in all possible cases. Solving the question is not a matter of being clever or smart enough—it is simply impossible.

For example, take the very first question of ACCEPT; that is, does M accept w? One way that will often answer the question is to feed M and w to the universal Turing machine (UTM), and run the UTM for a fixed time or a fixed number of steps (it would be easy to encode the UTM in a standard programming language, and let this program run for, say, 30 min).

It is possible—even likely—that the universal Turing machine will have found that M on w enters one of q_{accept} or q_{reject}. In the first case, one can say yes, M accepts w; in the second, no, that M does not accept w.

However, if the universal Turing machine has not yet announced that M has accepted or rejected w, one is out of luck. There is nothing to say. Running the machine for another fixed amount of time might—but might not—answer the question.

7.7 POST CORRESPONDENCE PROBLEM (OPTIONAL)

One of the most common, easily accessible problems from language theory is: Given a context-free grammar, is it ambiguous? That is, is there some string that has two (or more) different parse trees? This sometimes has an answer—for example, if one can find a string with two parse trees, or if one can prove the grammar meets certain requirements of compiler theory (is it LL(1) or LR(1), etc.). But there is no algorithm that will answer the question for all grammars; that is, the problem is undecidable.

Unfortunately, there is no easily accessible proof of this important fact. To be able to prove that the question is undecidable, the first step will be to consider the Post Correspondence Problem (PCP), which is easy to state, fairly easy to understand, and is important because of its applications to problems such as the ambiguity of a grammar.

Imagine a set of dominos, with the top of each domino containing some string of characters, and the bottom another string. There are n of

these dominos. The goal is to arrange some subset of these dominos in a line, so that the string across the top matches the string across the bottom.

Single dominos can be repeated—have duplicates. That is, there are n different dominos, but each domino has an unlimited number of copies. Think of the problem as having a finite number of patterns, but an infinite number of copies of each pattern. Thus, you are not limited to a string of length n or less, but can have arbitrarily long strings.

Note that if any domino has the same string on both the top and the bottom, there is a trivial solution. For example, if a domino is

$$\frac{\text{aabc}}{\text{aabc}}$$

then that domino by itself satisfies the criteria.

Note also that once there is a solution, then any concentrated repetition of this solution is also a solution. Thus, there is a shortest solution, but no longest solution, since the supply of dominos never runs out.

Example

The dominos are

$$\frac{\varepsilon}{\text{b}}, \frac{\text{a}}{\text{aa}}, \frac{\text{c}}{\text{b}}, \frac{\text{c}}{\text{cc}}, \frac{\text{abbb}}{\text{b}}$$

This has one solution:

$$\frac{\text{a}}{\text{aa}}\ \frac{\varepsilon}{\text{b}}\ \frac{\text{abbb}}{\text{b}}\ \frac{\text{c}}{\text{b}}\ \frac{\text{c}}{\text{cc}}$$

Note that the only possible first tiles are

$$\frac{\varepsilon}{\text{b}}, \frac{\text{a}}{\text{aa}}, \frac{\text{c}}{\text{cc}}$$

These are also possible as the last tiles, but so is

$$\frac{abbb}{b}$$

This example allows the empty string in a domino; PCP itself will not.

Definition 7.13

Post Correspondence Problem (PCP): Given Σ, and n pairs (a_i, b_i), $i = 1, \ldots, n$, with a_i and b_i strings from Σ^+ (so no string can be empty), is there a sequence $i_1, i_2, \ldots, i_k$ of integers, not necessarily distinct, $1 \le i, j \le n$, so that

$$a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_k} = b_{i_1} b_{i_2} \ldots b_{i_k}$$ ■

Note that here we are writing the top and bottom of the tile as the pair (a_i, b_i). That is, (a_i, b_i) is another way of writing

$$\frac{a_i}{b_i}$$

This problem will turn out to be undecidable—there is no algorithm to solve it. If one bounds the problem—adds a number K so that $k \le K$, then the problem is solvable in exponential time (try all subsequences), and is **NP**-complete (see Chapters 8 and 9).

Theorem 7.14

PCP is undecidable.

Proof

Nontrivial. First, we will prove that (1) the problem modified PCP (MPCP) $\le$ PCP, and then (2) MPCP is undecidable.

MODIFIED PCP (MPCP)

To prove PCP, we begin with a modified version. The idea is to simplify the proof. In this modified version, there are two special dominos, or four

special strings—a starting pair a_{start}, b_{start}, and a final pair a_{last}, b_{last}, plus a set of (a_i, b_i), i = 1, n.

Then the question becomes, does there exist a sequence so that

$$a_{start} a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_k} a_{last} = b_{start} b_{i_1} b_{i_2} \ldots b_{i_k} b_{last}$$

Example:

$$\frac{a_{start}}{b_{start}} = \frac{1}{10}, \frac{a_1}{b_1} = \frac{0100}{10}, \frac{a_2}{b_2} = \frac{01100}{0}, \frac{a_3}{b_3} = \frac{1}{011}, \frac{a_{last}}{b_{last}} = \frac{1}{01}$$

Then we can make the string 1 0100 1 1 01100 1 = $a_{start}a_1a_3a_3a_2a_{last}$ = = $b_{start}b_1b_3b_3b_2b_{last}$.

10 10 011 011 0 01

Note that one of the starting strings must be a prefix of the other; similarly, one of the last strings must be a suffix of the other. For these reasons, even without the restrictions of MPCP, a_1 could not start or end the matching.

Theorem 7.15

$$\text{MPCP} \leq \text{PCP}$$

Proof

We need to find a transformation from a possible instance of MPCP to one of PCP such that a yes goes to a yes, and a yes of PCP must have come from a yes of MPCP.

Consider an instance of MPCP with (n + 2) pairs or dominos

$$\frac{a_{start}}{b_{start}}, \frac{a_{last}}{b_{last}}, \quad \text{and} \quad \frac{a_i}{b_i}, i = 1, \ldots, n$$

Assume that the symbols [,], and # are not in Σ (if they are, choose different characters). Now, we will make a set of strings for PCP that use these

characters to force the correct starting and ending pairs in their proper positions with # moving between them.

Suppose that $a_{start} = \sigma_1\sigma_2 \ldots \sigma_k$. We make a string $x_0 = [\#\sigma_1\#\sigma_2\# \ldots \#\sigma_k$.

If $b_{start} = \sigma_1'\sigma_2' \ldots \sigma_j'$, we make a string $y_0 = [\#\sigma_1'\#\sigma_2'\# \ldots \#\sigma_j'\#$.

From the two ending strings, if $a_{last} = \tau_1\tau_2 \ldots \tau_r$, we make a string $x_{n+1} = \#\tau_1\#\tau_2\# \ldots \#\tau_r\#]$, and if $b_{last} = \tau_1'\tau_2' \ldots \tau_s'$, we make a string $y_{n+1} = \tau_1'\#\tau_2'\# \ldots \#\tau_s'\#]$.

Now for the

$$\frac{a_i}{b_i}$$

with $a_i = \alpha_1\alpha_2 \ldots \alpha_t$ and $b_i = \beta_1\beta_2 \ldots \beta_v$,

$$x_i = \#\alpha_1\#\alpha_2 \ldots \#\alpha_t \qquad y_i = \beta_1\#\beta_2\# \ldots \beta_v\#$$

Example

$a_{first} = 1$, $b_{first} = 10$, $a_1 = 0100$, $b_1 = 10$, $a_2 = 01100$, $b_2 = 0$, $a_3 = 1$, $b_3 = 011$, $a_{last} = 1$, $b_{last} = 01$.

$$\frac{a_{first}}{b_{first}} = \frac{1}{10}, \frac{a_1}{b_1} = \frac{0100}{10}, \frac{a_2}{by_2} = \frac{01100}{0}, \frac{a_3}{b_3} = \frac{1}{011}, \frac{a_{last}}{b_{last}} = \frac{1}{01}$$

$$\frac{x_0}{y_0} = \frac{[\#1}{[\#1\#0\#}, \frac{x_1}{y_1} = \frac{\#0\#1\#0\#0}{1\#0\#}, \frac{x_2}{y_2} = \frac{\#0\#1\#1\#0\#0}{0\#},$$

$$\frac{x_3}{y_3} = \frac{\#1}{0\#1\#1\#}, \frac{x_4}{y_4} = \frac{\#1\#]}{0\#1\#]}$$

Now, instead of (n + 2) dominos with two constrained to be first or last, we have (n + 2) new dominos with no constraints.

Claim: MPCP solution => PCP solution. This is very easy, just use the same strings.

Claim: PCP solution => MPCP solution:

1. It is clear that strings that were not marked first and last in the modified problem (such as a_1, a_2, and a_3 in the example above) cannot be either first or last since they have unmatched first and

last characters. They cannot be first since the top string begins with # and the bottom with a character from Σ; similarly, they cannot be last since the bottom string ends with # and the top with a symbol from the alphabet. Here, it is important to note that the strings are not allowed to be empty.

Thus, suppose the PCP problem has a solution. Then the modified a_{start} is indeed first, and the modified a_{last} is indeed last.

2. The modified

$$\frac{x_0}{y_0} \quad \text{and} \quad \frac{x_{n+1}}{y_{n+1}}$$

could appear in the middle of the PCP solution, but only as a pair

$$\frac{x_{n+1}}{y_{n+1}}, \frac{x_0}{y_0}$$

if x_0/y_0 were to appear in the middle with some other string before it, the bottom string would contain #[, and there is no way to get that substring on the top half. Thus, the match from

$$\frac{x_0}{y_0}$$

to the first occurrence of

$$\frac{x_{n+1}}{y_{n+1}}$$

would also be a match. This is the MPCP solution. ■

Theorem 7.16

$$\text{ACCEPT}_{TM} \leq \text{MPCP}$$

(where ACCEPT_{TM} = {M, w: M and w are encodings of TM M and string w, M accepts w}).

Proof

For a Turing machine configuration, we will embed the state into the string just before the symbol being read—this is one of the places where this convention is very useful.

Now we need to show how to transform the action of this Turing machine M on w so that:

1. If M accepts w, then the constructed MPCP has a solution.
2. If the constructed MPCP has a solution, then that M accepted w.
3. The construction can be carried out by a Turing machine or transducer.

The idea is that the pairs of strings in MPCP must, as they build up the matching string, simulate the action of M on w.

Thus, let $M = (\Sigma, \Gamma, Q, q_{accept}, q_{reject}, q_{start}, \delta)$ be a Turing machine. Let # be a symbol not in Γ (if it is in Γ, then use some other symbol in what follows). The ## will separate one configuration from another. It will mark the left and right boundaries of the input string, plus any space left or right of w that was used during computation.

We make these strings in MPCP:

$$\frac{a_{start}}{b_{start}} = \frac{\#\#}{\#\#q_0 w \#\#}$$

$$\frac{a_{last}}{b_{last}} = \frac{q_{accept}\,\#\#\#}{\#\#}$$

That is, we start out with the start state and w on the tape, and end in the accepting state. It will not matter what is on the tape once the machine is in q_{accept}.

Now, the δ function reads a symbol σ, writes a new symbol τ, and moves left or right. We deal with these moves separately.

Right moves: If $(q, \sigma) = (p, \tau, \text{right})$ add the pair

$$\frac{q\sigma}{\tau p}$$

Left moves: These are more complex since the state will move to the left across whatever symbol was there. Thus, if q reading σ moves left, writes τ, and goes to state p, then for every symbol *v* in the tape alphabet, add

$$\frac{\nu q \sigma}{p \nu \tau}$$

Extending a configuration left and right of head: These are simple but necessary. For every σ in Γ, add

$$\frac{\sigma}{\sigma}$$

Also add

$$\frac{\#}{\#}$$

Extending a configuration to new areas of tape: For each state q in Q, add

$$\frac{q\#}{qB\#} \quad \text{and} \quad \frac{\#q}{\#Bq}$$

These moves allow the string to expand to the left and right if the head reaches the markers #. These begin marking the start and end of the input string. They continue to mark either the start/end of input or the furthest left/right that the head has reached so far during its computation.

Now, notice that each string either keeps the top and bottom strings built so far equal in length, or lengthens the bottom string. The next tiles or pairs of strings are the only ones that are longer on top than on the bottom.

Finishing up from accept state: Add

$$\frac{q_{\text{accept}}\,\#\#\#}{\#} \quad \text{(top length = 4, bottom length = 1)}$$

For each symbol σ in Γ, add

$$\frac{\sigma q_{accept}}{q_{accept}}, \frac{q_{accept}\sigma}{q_{accept}} \text{ (top length 2, bottom length 1)}$$

These moves containing q_{accept} are the only string pairs with a top longer than the bottom.

Example

Consider a very simple TM that accepts strings over {0, 1} with the second symbol equal to 1. One TM for this language (though perhaps not the simplest machine, note that this is not a deciding machine) is shown in Figure 7.2. It is not a deciding machine.

This accepts 0101 by the following sequence of configurations (written with the head position shown by embedding the state in the string)

$$q_0010 \rightarrow 0q_110 \rightarrow 01q_{accept}0$$

The start domino, or pair, for MPCP is

$$\frac{\#\#}{\#\#q_0010\#\#}$$

The first move is q_0, $0 \rightarrow q_1$, 0, R, so we use the tile that corresponds to this move, and extend the string by

$$\frac{q_00}{0q_1}$$

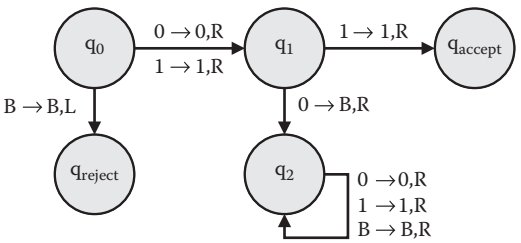


FIGURE 7.2 Accepting TM for L = {w/second symbol for w is 1}.

and then by the "extra" symbols

$$\frac{1}{1}, \frac{0}{0}, \frac{\#}{\#}, \frac{\#}{\#}$$

to give the partial match

$$\frac{\#\#}{\#\#q_0010\#\#}\frac{q_00}{0q_1}\frac{1}{1}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#} = \frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}$$

Now use the move-matching pair from $f(q, 1) = q_{accept}, 1, R$

$$\frac{q_11}{1q_{accept}}$$

to give

$$\frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}\frac{0}{0}\frac{q_11}{1q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}$$

(note the insertions of the plain pairs 0/0, 1/1, and #/# to make good extensions).

Then,

$$\frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}\frac{0}{0}\frac{q_11}{1q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}\frac{q_{accept}\#\#\#}{\#}$$

is our goal, but it is not quite correct. We need to match $01q_{accept}0$:

$$\frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}\frac{0}{0}\frac{q_11}{1q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}\frac{0}{0}\frac{1.q_{accept}}{q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}$$

eliminates the 1 and

$$\frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}\frac{0}{0}\frac{q_11}{1q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}\frac{0}{0}\frac{1.q_{accept}}{q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}\frac{0q_{accept}}{q_{accept}}\frac{0}{0}\frac{\#}{\#}\frac{\#}{\#}$$

eliminates zero.

Now try inserting

$$\frac{.q_{accept}\,0}{q_{accept}} \text{ (eliminate other 0)}$$

and

$$\frac{\#}{\#}$$

also adding

$$\frac{.q_{accept}\,\#\#\#}{\#\#}$$

at the end to obtain:

$$\frac{\#\#q_0010\#\#}{\#\#q_0010\#\#0q_110\#\#}\ \frac{0}{0}\ \frac{q_11}{1q_{accept}}\ \frac{0}{0}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{0}{0}\ \frac{1.q_{accept}}{q_{accept}}\ \frac{0}{0}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{0q_{accept}}{q_{accept}}$$

$$\frac{0}{0}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{q_{accept}\,0}{q_{accept}}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{q_{accept}\,\#\#\#}{\#}$$

To continue the proof of the theorem, we have two claims to prove:

Claim 1: If the Turing machine M accepts w, then the PCP instance has a solution. This is true because the sequence of moves shows how to construct the MPCP match. Follow the sequence of moves to q_{accept}. Each move of the TM has a corresponding string pair.

Each move has string pairs so that the bottom configuration of the Turing machine can be duplicated on the top until one reaches the accept state. Then—and only then—can you shrink the top string to match the bottom.

Claim 2: If there is an MPCP matching, then the Turing machine must accept w. This happens since we begin with $\#/\#q_0w\#\#$—the first tile has a top string of length 1, and a bottom string of

length |w| + 5 > 1. The only way to allow the bottom to catch up with the top is to use the strings from q_{accept}—so that state must be reachable.

Example

A simple TM to accept strings ending in 1, is shown in Figure 7.3.

q_0, B → q_1BB → q_{reject}BBB

$q_0$101 → 0$q_0$01 → 00q_0 1→ 001q_0B → 00$q_1$1B→ 001q_{accept}B

Right moves: $q_0$0/0q_0 $q_0$1/1q_0 $q_1$0/0q_R q_1B/Bq_{reject} $q_1$1/q_{accept}

Left moves: Zq_0B/q_1ZB Z=0,1,B

string extension: 0/0 1/1 #/#

blank tape extension q#/qB# #q/#qB for all q in Q

string shortening q_{accept}B/q_{accept} Bq_{accept}/q_{accept} q_{accept}0/q_{accept} 0q_{accept}/q_{accept} q_{accept}1/q_{accept} 1q_{accept}/q_{accept}

start: ##/##q_0w## end: q_{accept}###/#

Example

A simple TM to accept strings with an even number of 0's. Moves right to a blank, accepts if number of 1's seen so far is even; shown in Figure 7.4.

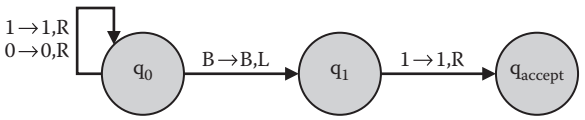
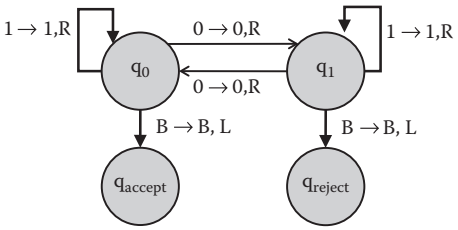


FIGURE 7.3 Turing machine for L = {w | w ends in 1}.



Start: $\dfrac{\#\#}{\#\#q_0\#\#}$

Finish: $\dfrac{q_{\text{accept}}\#\#\#}{\#}$

4 Right Moves: $\dfrac{q_0 1}{1 q_0}\ \dfrac{q_0 0}{0 q_0}\ \dfrac{q_1 1}{1 q_1}\ \dfrac{q_1 0}{0 q_1}$

4 String Extension Moves: $\dfrac{\#}{\#}\ \dfrac{0}{0}\ \dfrac{1}{1}\ \dfrac{B}{B}$

6 Left Moves: $\dfrac{X q_0 B}{q_{accept} X B}\ \dfrac{X q_1 B}{q_{reject} X B}\quad X = 0,1,B$

8 Tape Ext. Moves: $\dfrac{q\#}{qB\#}\ \dfrac{\#q}{\#Bq}\quad q = q_0, q_1, q_{accept}, q_{reject}$

6 String Shortening Moves: $\dfrac{q_{accept} X}{q_{accept}}\ \dfrac{X q_{accept}}{q_{accept}}\quad X = 0,1,B$

MPCP emulation of q_0,B → q_{accept}, B,L:

$$\frac{\#\#}{\#\#q_0\#\#}\ \frac{q_0\#}{q_0B\#}\ \frac{\#q_0}{\#Bq_0}\ \frac{B}{B}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{Bq_0B}{q_{accept}BB}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{q_{accept}B}{q_{accept}}\ \frac{B}{B}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{q_{accept}B}{q_{accept}}\ \frac{\#}{\#}\ \frac{\#}{\#}\ \frac{q_{accept}\#\#\#}{\#}$$

FIGURE 7.4 TM and tiles for L = {w | w contains an even number of zeros}.

Corollary 7.17

MPCP is undecidable. ■

7.8 CONTEXT-FREE GRAMMARS (OPTIONAL: REQUIRES SECTION 7.7)

Much of the theory of programming languages and compiler design is based on context-free grammars (or Backus Naur form grammars), since these are a convenient way to specify most of what makes a program syntactically legal. It is easy to write grammars to specify the legal statements in most languages.

Definition 7.18

A **context-free grammar** over an alphabet Σ (sometimes called the terminal symbols) is a set of variables (or nonterminals) V, which includes the start symbol S, and a set of rules of the form

variable → some string of terminals and nonterminals ■

A **leftmost derivation** is a chain of rules, beginning with the start symbol and ending with a string over Σ^*, such that at each step, the leftmost variable is replaced by one of its right-hand sides.

Example

In C, a block statement begins with { and ends with }. Inside the braces, there can be any number of statements. Each statement is followed by a ;.

For this example, let s, standing for statement, be a terminal:

$$\Sigma = (s, \text{‘\{‘}, \text{‘;’}, \text{‘\}’}\} \qquad V = \{S, SL\}$$

$$S \rightarrow \{ SL\}$$

$$SL \rightarrow s; SL$$

$$SL \rightarrow \varepsilon$$

When two or more rules have the same left-hand side, they can be written together with | (for or). For example, $SL \rightarrow s; SL \mid \varepsilon$.

Leftmost derivation for {s; s; s;}:

$$S \rightarrow \{SL\} \rightarrow \{s ; SL\} \rightarrow \{s; s; SL\} \rightarrow \{s; s; s; SL\} \rightarrow \{s; s; s;\}$$

Definition 7.19

A grammar is **ambiguous** if some string that it generates can be generated in more than one way (has more than one leftmost derivation). ■

Example

Simple grammar for arithmetic expressions over the variables a, b, c:

$$\Sigma = \{a, b, c, +, -, *, /, (,)\} \qquad V = \{E\}$$

$$E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E)$$

$$\rightarrow a \mid b \mid c$$

It is easy to find two leftmost derivations for the string a + b * c (and many other strings).

Ambiguity in grammars is important to language design and compiler writing. For example, no ambiguous grammar is suitable for either LL or LR parsing (although there are sometimes ways to get around the problem and use a grammar, even though it is ambiguous). Thus, knowing whether or not a grammar is ambiguous is a question of interest.

Theorem 7.20

It is undecidable whether a Context Free Grammar (CFG) G is ambiguous.

Proof

By reduction from PCP, that is, PCP ≤ Grammar_G_ambiguous. Given an instance of PCP with tiles

$$\frac{a_1}{b_1},\ldots,\frac{a_n}{b_n}$$

the task is to construct a grammar G that is ambiguous if and only if the instance of PCP has a solution.

The construction, due to Floyd (1962), builds two grammars with start symbols A and B, and combines them into one grammar with start S and rules $S \rightarrow A|B$.

The A grammar uses the a_i strings from the top of the tiles, plus "index variables" $r_1, \ldots, r_n$.

The rules are: $A \rightarrow a_1Ar_1 \mid a_2Ar_2 \mid \ldots \mid a_nAr_n \mid a_1r_1 \mid a_2r_2 \mid \ldots a_nr_n$.

This grammar is not ambiguous, and generates strings of the form (some string of a_i) (some string of r_j) with matching lengths and reversed indices.

The B grammar is analogous, but uses strings from the bottom of the tiles

$$B \rightarrow b_1Br_1 \mid b_2Br_2 \mid \ldots \mid b_nBr_n \mid b_1r_1 \mid b_2r_2 \mid \ldots b_nr_n$$

Again, this grammar is unambiguous by itself, and generates

$$b_{i_1} b_{i_2} b_{i_3} \ldots b_{i_k} r_{i_k} r_{i(k-1)r} r_{i(k-2)} \ldots r_{i_2} r_{i_1}$$

If the grammar generated from S is ambiguous, some string has a derivation both from A and from B. The parts of those strings with the r_i removed are a solution to the PCP.

If the PCP problem has a solution, that string can be generated from both the A and B grammars (with suitable r_i) and will be an ambiguous string in G. ■

Theorem 7.21

The following questions about CFGs G_1 and G_2 are undecidable:

1. $L(G_1) \cap L(G_2) = \varnothing$
2. $L(G_1) = L(G_2)$
3. $L(G_1) \subseteq L(G_2)$
4. $L(G_1) = \Sigma^*$

Proof

All are reductions from PCP to either the question or its complement. Recall that either (yes goes to yes and no to no) OR (yes goes to no and no goes to yes) from an undecidable problem will show that this problem is undecidable. ■

Comment: There are some questions about context-free languages that have algorithms—for example, whether a string w is in L(G), whether L(G) is empty, or whether L(G) is infinite. Thus, unlike questions about Turing machines (no nontrivial question about the language accepted is decidable), there are questions about grammars that have algorithms. These algorithms are often nontrivial; for more on them, see any standard text on formal language theory. Unfortunately, as the previous two theorems show, there are many questions that have no algorithms.

Thus in an automata course, one finds that: (1) almost every question about deterministic finite automata is decidable, (2) some questions about context-free grammars are decidable, and (3) no nontrivial question about

the language accepted by Turing machines is decidable. Thus, as the power of the machine increases, the number of decidable questions decreases.

EXERCISES

1. Prove each of the following is undecidable. Do not use Rice's theorem.
   a. The writing problem: Given a symbol σ, does the Turing machine ever write σ during its computation on any string whatsoever?
   b. The halt-on-something problem: Given a Turing machine and a string, does the machine halt (enter either) on any string in Σ^*?
   c. The emptiness problem: Given a Turing machine, does the machine accept any string at all (is its language nonempty)?
   d. The reject anything problem: Given a Turing machine, does the machine reject any string at all (is its language non-Σ^*)?
2. For each of the following questions, show whether the question is decidable or undecidable. Explain each answer.
   a. Given a Turing machine M, does it have a state q_{73}?
   b. Given a Turing machine M, does it have any transition into state q_{73}?
   c. Given a Turing machine M and a string w, does it ever enter state q_{73} while processing w?
   d. Given a Turing machine M, does it ever enter state q_{73} on any string whatsoever?
3. Given two languages L_1 and L_2, and the language $L = L_1L_2$ (the concatenation of the first languages), prove:
   a. If L_1 and L_2 are Turing decidable, so is L.
   b. If L_1 and L_2 are Turing acceptable, so is L.
4. Are the following decidable or not? Justify your answers.
   a. Given a Turing machine M with input w, does M ever move its head to the left during its computation on w?
   b. Given a Turing machine M with input w, does M ever write any symbol on its tape?

5. Let L_{321} = {M | TM M halts on all inputs of length less than 321} over Σ = {a.b}.

   a. Prove L_{321} is accepted by some Turing machine.

   b. Prove L_{321} is not decided by any Turing machine.

   c. Prove that $\overline{L_{321}}$ (the complement of L_{321}) is not decided or accepted by any Turing machine.

6. In compiler design class, one learns: (a) an algorithm for deciding if a grammar is LR(1), and (b) that all LR(1) grammars are unambiguous. Explain why or why not: the LR(1) algorithm is an algorithm for deciding if a grammar is unambiguous.

7. Consider these PCP problems, each with three tiles or pairs of strings:

$$\text{(i)}\ \frac{11}{10}, \frac{101}{1}, \frac{1}{110} \qquad \text{(ii)}\ \frac{11}{10}, \frac{101}{1}, \frac{1}{1110}$$

   a. In problem i it is easy to argue that the tile

$$\frac{11}{10}$$

   cannot be first. Explain. It is also possible to argue that the tile

$$\frac{101}{1}$$

   cannot be first. Again, explain.

   b. Can either of these tiles be last? Explain.

   c. Find a solution to i.

   d. (Not easy!) Find a solution to ii. Again, you cannot start with the first or second tile.

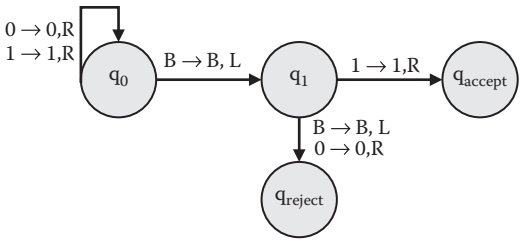


FIGURE 7.5 Assignment problem.

8. Prove or disprove: (a) PCP over a unary alphabet is undecidable. (b) PCP over a binary alphabet is undecidable. (Unary alphabets contain only one symbol; binary alphabets contain two symbols.)

9. Prove that the construction MPCP ≤ PCP does not work if some strings are allowed to be empty. (Hint: a_{start} = 1/10, a_1 = 0/eps, a_{last} = 0/00 has no solution but the PCP problem does.)

10. Build the MPCP problem from the Turing machine in Figure 7.5 (accepting strings that end in 1); show that the MPCP action on q_0, ε has no solution, but that on q_0, 01 does.

11. Prove:

    Theorem: If $L_1 \le L_2$ and L_2 is decidable, then so is L_1.

    Theorem: If $L_1 \le L_2$ and L_1 is undecidable, then so is L_2.

    Theorem: If $L_1 \le L_2$, then $\overline{L_1} \le \overline{L_2}$.

    Theorem: If $L_1 \le L_2$ and $L_2 \le L_3$, then $L_1 \le L_3$.

CHAPTER 8

Classes **NP** and **NP**-Complete

8.1 THE CLASS **NP** (NONDETERMINISTIC POLYNOMIAL)

Most of the problems met in the standard computer science curriculum are in the class **P**, defined in Chapter 3 (see Definition 3.6), of problems that have polynomial solutions. One spends a lot of time in classes in data structures and algorithms analysis considering whether an algorithm is of order n^2, n^3, n ln n, and so on. You might note that one often dealt with *search* problems, and in this class, we are dealing with *decision* problems—not "What is the shortest path?" but "Is there a path from A to B shorter than L?"

Now, we introduce a class of problems, **NP**, which (almost certainly—more about this later) do not have polynomial solutions.

You should be asking yourself while you read this chapter:

Do I understand what makes a problem belong to the class **NP**? Can I show that a problem is in **NP**?

What kinds of problems are in **NP**?

What problems are *not* in **NP**? Do I know they are not in **NP**?

If I were given the task of writing an algorithm for a problem in **NP**, what are my options?

8.2 DEFINITION OF **P** AND **NP**

Recall the definition of **P**—those decision problems that are easy to solve.

Definition 8.1

The class **P** is the set of all decision problems that have polynomial solutions (that is, they are in $O(p(n))$ for some polynomial p in the size of the input n). ■

In contrast, **NP** will consist of problems that are not easy to solve (though no one has been able to prove this!), but which have solutions that are easy to check.

There are two standard ways to define the class **NP**. Nothing mathematical should have two definitions, but these two definitions are equivalent. So, we will give both, and immediately show that they are equivalent. Sometimes it will be easier to use one definition than the other, so it will be useful to have both available.

Note that a yes instance of a decision problem can have a **certificate**. A certificate is something that shows how the decision problem is satisfied, that is, what proves that it is a yes instance. For example, it is almost always possible to think of some kind of certificate for a problem. An actual solution is always a certificate. And, for many problems, there are short simple certificates—certificates that are shorter than the actual solutions.

COMPOSITE = {n | n is a positive integer and n is not prime}.

Certificate: A pair of numbers $1 < x,y < n$ such that $xy = n$.

Verifier: Multiply $x*y$, check that the product is n.

PRIME = {n| n is prime positive integer} has no obvious short simple certificate.

Certificate: Listing of all primes p from 2 to n/2, with their nonzero remainders ($n \bmod p \neq 0$).

Verifier: Check these approximately n/4 divisions (or see Pratt, 1975) for a polynomial certificate and verifier).

TAUTOLOGY = {Boolean expressions that are true for all truth assignments}.

Certificate: No known short certificate, but certainly the complete truth table would work as a certificate; verify that each row is true.

Verifier: Calculate truth table, check that each row evaluates to true.

PATH = {G = (V, E) with vertices v, w, and there is a path from v to w in G}.

Certificate: Listing of the vertices in the path.

Verifier: Check that the path is legal.

The class of problems that we are next interested in are those that have short certificates, but no known short solutions (where "short," as usual, means bounded by a polynomial in the length of the input). Informally, these are problems where it is hard to find a solution, but easy to check a solution works if you are given one.

The class of problems with short certificates, **NP**, is called the class of **nondeterministic polynomial (decision) problems**. Of course, **P** $\subseteq$ **NP**, so the problems that are interesting are those in **NP**-**P**.

Definition 8.2

A decision problem is in **NP** if there is a nondeterministic Turing machine (TM) M that for any yes instance X_Y, M reaches q_{accept} in time bounded by a polynomial in the size of the input to the machine, $|X_Y|$. ■

That is, if X is a yes or no instance of the problem, and the length (size) of the encoding of X is n (written $|X| = n$), then there is a nondeterministic Turing machine (NDTM) $\mathbf{M_N}$, and a polynomial p such that:

If X is a yes instance, $\mathbf{M_N}$ has a path of length $\leq p(n)$ to the accept state.

If X is a no instance, $\mathbf{M_N}$ has no path to the accept state.

Definition 8.3

A decision problem is in **NP** if and only if it has a polynomial verifier. ■

That is, if X is a yes instance of the problem, then X has a certificate C, and there is a (deterministic) Turing machine M_D that can check (verify) C—that it is valid—and this check is done in a number of steps bounded by a polynomial in the length of the problem.

If the length of the encoding of X is n, and the length of the certificate C is m, then there is a polynomial p′ such that the deterministic Turing machine (DTM) M_D runs in steps bounded by p′(n).

Note: The length of the certificate m is not mentioned in the definition, but it must be bounded by a polynomial in n; otherwise, the DTM could not even read the entire certificate in p′(n) steps.

Note: The initials N and P do *not* stand for nonpolynomial! That is a common misconception, but incorrect for several reasons: (1) it has *not* been proved that there are no polynomial algorithms for the problems in **NP**, (2) **P** $\subseteq$ **NP**, so some problems in **NP** (those in **P**) *do* have polynomial solvers, and (3) many nonpolynomial problems are not in **NP** (for example, the undecidable problems). Instead, **NP** stands for nondeterministic polynomial.

Theorem 8.4

The two definitions above (Definitions 8.2 and 8.3) are equivalent.

We need to show that an NDTM solver exists if and only if (iff) a polynomial verifier exists.

Direction 1: Assume the first definition is satisfied—the problem has an NDTM, and a yes instance has a path to the accept state bounded by p(n).

The certificate here is the path to the accept state. Remember that a certificate can be anything that proves that this is a yes instance—so showing how to accept will do that. To verify the certificate, just follow its path and see that one has reached accept. The path is bounded by a polynomial—this takes the same polynomial number of steps.

Direction 2: Assume the second definition is satisfied, that is, a yes instance of the problem has a certificate with a polynomial verifier with bound p′(n). Now show that a polynomial-bounded NDTM exists.

We construct the NDTM as follows. Given X, a yes or no instance of the problem, we construct a two-phase machine. The first phase will be to guess the certificate. This can be done by looking at the form of the certificate. At each step of the first phase (1, 2, …, p′(n)), the choices of the NDTM are: (1) write down one possibility for this step, or (2) stop writing the certificate and proceed to the second phase. Once the bound is reached, the NDTM must stop writing the certificate.

The second phase of the NDTM is simply to run the deterministic polynomial verifier on the guessed certificate.

The machine so constructed is an NDTM. In its first phase, it has (probably many!) choices at each step. But both phases are polynomial. The first phase is polynomial since it must stop guessing after p′(n) steps, and its second phase is just the deterministic verifier bounded by p′(n). Thus, the entire NDTM has an accepting path bounded by p′(n)*[length of time to make a guess for the next step] + p′(n) = CONSTANT* p′(n). ■

This may not seem intuitive, and it certainly will not seem efficient. For example, suppose you want to show there is a path from A to H in the graph:

A – B – C – D – E – F – G – H

A path is a sequence of nodes—so at each step, the NDTM certificate builder will: (1) guess that the path is finished (if path length has reached |V|, then it knows the path must be done), and (2) if not finished, guess the next node. It is very unlikely to guess the path above—it is even unlikely to guess a valid path with nonrepeating nodes. It is also allowed to stop the path after 2 or 3 or 4 nodes, or to keep going after 8 to 9, 10, or more, up to |V| + 1.

For another example, 5041 is a composite number (it is divisible by 71, though not by any other primes). The NDTM begins by guessing a sequence of digits to write down—it could write one, two, three, or four digits. It then checks that the number it wrote will indeed divide 5041. There are 5039 possibilities (2 … 5040) for the number it guesses if we limit the guess by 5041 (or 9998 guesses if we just limit it to four digits, 2 … 9999). The inefficiency does not matter. It also does not matter that only one of the 5039 (9998) possibilities works, so its chances of guessing correctly are less than 1 in 5000 (or 10,000). As long as: (1) there is some way to guess correctly, and (2) we can verify that the guess is a certificate in polynomial time. One can find 5041/71 in time bounded by a polynomial in |5041| = 4 —that is all that we care about.

In other words, do not worry about missing the correct certificate. As long as there is a short certificate that can be checked correctly, that is sufficient. If an omniscient omnipotent guesser could make the correct guess, that is enough.

8.3 POLYNOMIAL REDUCIBILITY

We already have seen when it is useful to reduce one problem to another:

$$P_1 \leq P_2$$

so that answering P_2 will give answers to both P_1 and P_2. Intuitively, P_2 is harder than P_1 (or more correctly, P_2 is *at least as hard as* P_1, or P_2 is *as hard or harder than* P_1).

Here, because we are also concerned with the length of time—polynomial or whatever—we must also assume that this mapping is of reasonable length. If the mapping took too long to compute, a fast solution to P_2 would not necessarily be a fast solution to P_1.

Thus, we will put a restriction on the definition of reducibility, so that solving the second problem not only solves the first, but fast solutions to the second are also fast solutions to the first.

Definition 8.5 (Definition 7.2 Restated)

A problem P_1 is **polynomially reducible** to a problem P_2 if there is a mapping f from the domain of all instances of P_1 into the set of instances of P_2 such that:

1. If A is a yes instance of P_1, then f(A) is a yes instance of P_2.
2. If A is a no instance of P_1, then f(A) is a no instance of P_2.
3. For any yes/no instance A of P_1, f(A) can be computed in polynomial time (polynomial, as usual, in the size of the input). ■

One can rephrase the second requirement as 2′: If f(A) is a yes instance of P_2, then A was a yes instance of P_1.

We write:

$$P_1 \leq_P P_2$$

The above means (more formally) that if the length of the encoding of A is |A|, then there is a polynomial p of degree n such that the number of steps to compute f(A) is bounded by p(|A|).

Sometimes, this mapping is trivial. For example, if P_1 is a subset of P_2, then we can just use the identity mapping. This makes sense; restricted problems are easier (or at least no harder) than their generalizations.

Recall the problem where you wanted to find rational roots of a polynomial of degree 3 or less, and someone handed you an algorithm to solve all polynomials up to degree 4. Then:

Polynomial of degree <= 3 with rational root $\leq_P$

Polynomial of degree <= 4 with rational root

You may ask why answering P_1 does not also answer P_2. Since yes goes to yes, and no goes to no, all you have to do to answer P_2 is look at where it came from. That is true, but there is nothing in the definition to say either (a) that it must have come from something—as *f* is not necessarily onto, or (b) if it did come from something, that it is easy to figure out where it came from (function *f* may not have an inverse, and even if it does, it may not be easily computable).

In the polynomial example, the function is not onto. If one had solutions to polynomials of degrees up through 3, that would not help in finding the roots of $x^4 - x + 19 = 0$.

8.4 PROPERTIES

There are several theorems on polynomial reducibility that are similar to those on the reducibility discussed in Section 7.3.

Theorem 8.6

If $L_1 \leq_P L_2$ and L_2 is in **P**, then so is L_1.

Theorem 8.7

If $L_1 \leq_P L_2$ and L_1 is not in **P**, then L_2 is not in **P**.

Theorem 8.8

If $L_1 \leq_P L_2$ and L_2 is in **NP**, then so is L_1.

Theorem 8.9

If $L_1 \leq_P L_2$, then $\overline{L_1} \leq_P \overline{L_2}$.

Theorem 8.10

If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$. ■

Proof

Exercises. ■

8.5 COMPLETENESS

Another definition is in order—one that applies to this class of problems, but also to any other class *C*.

This will give us a formal definition for what it means to be the hardest kind of a problem in *C*—to be at least as hard as any other problem in the class.

Definition 8.11

A problem Y is "complete for the class *C*" or ***C*-complete** if:

1. Y is in the class *C*.
2. For any problem X in *C*, $X \leq_P Y$. ■

Definition 8.12

A problem Y is ***C*-hard** if for any problem X in *C*, $X \leq_P Y$. ■

Thus, any *C*-complete problem is also *C*-hard. A problem can be hard without being in the appropriate class. A problem might be *C*-hard without even being decidable.

These definitions of *hard* and *complete* do not work well for the class **P** of decision problems with polynomial solvers. Using these definitions, every nontrivial decision problem in **P** would be **P**-hard and **P**-complete

(Exercise 10). Thus, there is a more restrictive version sometimes used for this class, but we will not consider it here. See Bovet and Crescenzi (1994).

Definition 8.13

A problem Y is **NP**-complete if:

1. Y is in the class **NP**.
2. For any problem X in **NP**, $X \leq_P Y$. ■

Definition 8.14

A problem Y is **NP**-hard if for any problem X in **NP,** $X \leq_P Y$. ■

There are two requirements for a decision problem to be **NP**-complete. One must show that it satisfies both 1 and 2. Showing 1 is often easy, and both texts and students tend to neglect it—either forget to mention it or say something such as "obviously in **NP**," but one must not simply ignore this constraint.

The class of **NP**-complete problems is important. These are the hardest problems in **NP**.

8.6 INTRACTABLE AND TRACTABLE—ONCE AGAIN

Definition 8.15

A problem is **intractable** if it is **NP**-hard. A problem is **tractable** if it is in **P,** the class of problems with polynomial solutions. ■

Why these definitions? First, why consider any problem in **P** to be manageable or tractable or easy? Problems with high-order polynomial solutions, such as an n^{10} algorithm or worse, are surely not easy to solve! There are several reasons, though, why we will call n^{10}—or even higher polynomial degrees—easy. First, compared to the exponential growth, n^{10} is not big. Second, if we defined easy problems to be bounded by small polynomials, we would first have to decide how big a polynomial could be and still be called small—n^2, n^3, or n^4—and then we would have some difficulties with

the composition of problems. One easy problem composed with another easy problem would not necessarily be easy, since if $f(n) = n^3$ and $g(n) = n^2$, the composition of the two functions, $f \bullet g$, has order n^6.

If we can agree on the class as being the tractable or easy problems, we have the nice properties listed in the next theorem:

Theorem 8.16

If two functions f and g are in **P**, then so are (1) $f(n) \bullet g(n)$, (2) $f(n) + g(n)$, and (3) $f(g(n))$.

Proof

Exercise. ■

This means that if two problems are tractable, then one can combine the easy problems in all standard ways and the result is still tractable.

But why are **NP**-hard problems not easy? Why call them intractable? This brings us to the boundaries of computer science knowledge.

There is no known polynomial algorithm for any **NP**-hard problem. However, it has never been shown—and will (almost certainly) never be shown that **NP**-hard problems do not have polynomial solutions. The famous incompleteness theorem of Kurt Godel shows that any theory will have some statements that are true but can never be proved, and (most) computer scientists think that the nonexistence of polynomial solutions for **NP**-hard problems is in this category.

In other words, it is believed that no one will find a polynomial time algorithm for any **NP**-hard problem. It is also believed that no one will prove that a polynomial time algorithm does not exist. The person who can do either one of these—find a polynomial algorithm for some **NP**-hard problem or show that one cannot exist—will be immediately famous, at least in the world of theoretical computer science. See the Web site referenced in Exercise 11 at Claymath.org for a temptation to take on this problem.

If either of these is achieved, the whole theory of complexity will essentially collapse, and the phrases **NP**-hard and **NP**-complete never again be mentioned in any computer science class, except perhaps in passing reference to the ignorance of the ancients.

To reiterate the important points: (1) no **NP**-hard problem has a known polynomial algorithm, but (2) we cannot show that no polynomial algorithm exists.

8.7 A FIRST **NP**-COMPLETE PROBLEM: BOOLEAN SATISFIABILITY

So far, we have defined **NP**-complete and **NP**-hard, but have given no examples. The first example, and the proof that the problem is in these two classes, is basic to all further discussion.

Recall Boolean logic, with expressions consisting of Boolean variables, with values either true or false, connected by the standard Boolean operations of negation ($\neg$), conjunction ($\wedge$), or ($\vee$), and implies ($\rightarrow$).

For example, with the three Boolean variables x, y, z, one can write many Boolean expressions such as:

$$\neg(x \wedge \neg y) \rightarrow (z \wedge \neg y)$$

This expression is true for many assignments to x, y, z. We say that a set of assignments to the variables x, y, z that makes it true **satisfies** the expression. For example, x = false, y = false, z = true (x = 0, y = 0, z = 1) will satisfy the expression above.

This expression is not in the special form called conjunctive normal form.

Definition 8.17

A Boolean **clause** is a collection of terms connected by $\vee$. A term (or literal) term is either a single variable (x_i) or the negation of a single variable ($\neg x_i$). ■

Definition 8.18

A Boolean expression E is in **conjunctive normal form** if it consists of a set of clauses $C_1, C_2, \ldots, C_m$ connected by $\wedge$'s. ■

Fact

Any Boolean expression can be converted into conjunctive normal form. ■

$$\neg(x \wedge \neg y) \rightarrow (z \wedge \neg y) == (x \vee z) \wedge (\neg y \vee z) \wedge (x \vee \neg y) \wedge (\neg y \vee \neg y)$$

Clause list:

$$x \vee z \qquad \neg y \vee z \qquad x \vee \neg y \qquad \neg y \vee \neg y$$

Fact

An expression in conjunctive normal form (CNF) is true if and only if each of its clauses is true. Thus, some texts do not speak of an expression in conjunctive normal form; instead, they speak of a set of clauses that all have to be satisfied.

Decision problem: CNF satisfiability (abbreviated CNF-SAT).
Instance: A Boolean expression E over variables $x_1, \ldots, x_n$ in conjunctive normal form.
Question: Is there an assignment of truth values to the variables $x_1, \ldots, x_n$ such that E is true (that E is satisfied)?

This is our basis problem, the first **NP**-complete problem. The theorem is: easily stated; the main part of the proof is the next section of the text, but the proof begins below.

Theorem 8.19

(Cook–Levin) CNF-SAT in **NP**-complete.

Clearly, CNF-SAT is a decision problem. It is trivial to find yes and no instances—one can do so even with n = 1 and the single Boolean variable x_1. (Yes instances: x_1, $x_1 \vee \neg x_1$, $x_1 \vee x_1$; no instance: see Exercise 14a).

Claim: CNF-SAT is in **NP**.
A certificate for CNF-SAT is the assignment of true and false to the variables that satisfy the expression—make the expression true, or equivalently, make every clause true. We *do not* have to find the certificate—we guess it (omnipotently, omnisciently) or that it is simply handed to us.

We *do* have to show that we can verify that this assignment of truth values makes the whole expression true, and the verification is bounded by time polynomial in the length of the input.

If we wish to be very precise, we will have to spend a lot of detail encoding the input and measuring its length. Suppose

$$E = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_m, \text{ each } C_1 \text{ a clause containing some of } x_1, \ldots, x_n$$

We can imagine a Turing machine with input alphabet $\Sigma = \{x, 0, 1, \ldots, 9, \wedge, \vee, \neg, (,)\}$. (Aside: All of these symbols are certainly not necessary. We can eliminate the x and some of the logical symbols, and the parentheses (). How? We can rewrite decimal in binary, of course, to eliminate 2, …, 9.)

The Turing machine would begin with the certificate (a string of length n of 0's and 1's for the values, false or true, for the $x_1, \ldots, x_n$), followed by the expression itself. One could encode $x_{59} \vee \neg x_{67}$ as $x59 \vee \neg x67$, or just as $59 \vee -67$ or for the example $(x_1 \vee x_3 \vee x_5) \wedge (\neg x_2 \vee x_4 \vee x_1) \wedge \neg(x_1 \vee x_4) \wedge x_3 \wedge (\neg x_1)$ with x_1, x_2 and x_3 true, x_4 and x_5 false, might be encoded as:

$$11100(x1 \vee x3 \vee x5) \wedge (\neg x2 \vee x4 \vee x1) \wedge \neg(x1 \vee x4) \wedge x3 \wedge (\neg x1)$$

which has 45 characters, but it could be encoded in other ways with fewer characters.

It might be easiest to begin by copying the certificate to a second tape. One approach would be to make n passes across the expression. At the ith pass, the machine would look at the value (true or false) for, and mark any clause true containing, x_i (if true) and $\neg x_i$ (if false). After n passes, the expression is true if all clauses are marked true. Getting the truth assignment for x_i would take less than n steps; making a pass over the expression would take less than $|E|$ steps, and is done n times, so the entire procedure would take less than $n \cdot |E|$ steps.

The quantities n, m, and $|E|$ are related. For interesting clauses, n is probably much less than $|E|$, although it would be possible to have an expression over many variables where only a few of them appeared at all. It is left as an exercise to explore in more detail the relationship between n and $|E|$. The important number for us is the length of the input, $|E|$. The bound of $n \cdot |E|$ is polynomial in $|E|$.

8.8 COOK–LEVIN THEOREM: PROOF

The basis for much of the rest of the proof will be the theorem that shows that CNF-SAT is **NP**-hard (and since CNF-SAT is in **NP**, it is therefore **NP**-complete).

What is necessary is to show that for any problem X *whatsoever* in **NP**, that $X \leq_p$ CNF-SAT.

This seems incredibly difficult. We know absolutely nothing about X except that it is in **NP**. It might be a graph problem, a Boolean expression problem, a problem with numbers, or something altogether different. But we must find a way to transform X into CNF-SAT so that: (1) the reduction can be done in polynomial time, (2) a yes instance of X becomes a satisfiable expression in CNF-SAT, and (3) a no instance of X becomes an expression in CNF-SAT that cannot be satisfied by any assignment of truth values to the variables.

All that we know about X is that it is in **NP**; that is, it has a nondeterministic Turing machine $\mathbf{M_X}$ that has a short (polynomially bounded) path to q_{accept} for yes instances, but no path ever to q_{accept} for no instances. The proof, due to Steven Cook (1971, a paper based on his PhD thesis) and proven independently by Levin (1974), using another method, is ingenious. Cook received the prestigious ACM Turing Award in 1982 for his work. His Turing Award lecture (1983) is worth reading. Hartmanis and Hopcroft (1971) present an overview of some the early work in complexity theory.

The idea will be to take the Turing machine $\mathbf{M_X}$, and to write a set of Boolean clauses that effectively and completely describe its operation and the path to the accepting state.

The Boolean expression to be satisfied is then the conjunction of all the clauses. The clauses should be $\vee$'s of variables and their negations. These clauses must specify that:

- The Turing machine starts in the proper configuration.
- Each configuration is legal for the Turing machine. This will mean: the TM tape head is on one and only one cell, its tape cells each contain one and only one symbol, and the machine is in one and only one state at each step.
- Each move follows one of the legal moves in its transition function.
- The Turing machine reaches its accept state.

Further, it must be shown that one can construct these clauses in polynomial time, and that yes goes to yes, and no goes to no.

8.8.1 Proof Part I: Construction of the Clauses

We begin with a nondeterministic Turing machine M that accepts an arbitrary language L in **NP**. We know that M is defined by its states Q, its alphabets Σ and Γ, and its transition function δ. An input string x of length $n = |x|$ has a path to q_{accept} if and only if it is in L, and the length of this path is bounded by a polynomial in n, p(n).

We can order Q in the same order as for the encoding of Turing machines:

$$Q = \{q_{start} = 0, q_{accept} = 1, q_{reject} = 2, \ldots, \text{other states}\}$$

We will need to identify the squares on the Turing machine tape. We can number them with the integers:

$$\ldots -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, \ldots$$

We will put the number 1 on the first square of the input x, so that is the position of the head at the start of the computation. If $x = \varepsilon$, the head will be on a blank.

As there is an infinite amount of tape, how much of it do we need to consider in an accepting computation? The accepting computation is of length p(n)—in that many time steps, it could not be farther right than square [p(n) + 1], or farther left than [–p(n) + 1]. Thus, we can restrict attention to those tape squares. All other tape squares remain blank during the accepting computation.

We also need to identify the steps in the accepting computation. We will use the variable t to denote time after t steps, $0 \le t \le p(n)$. Thus, we have bounded both the amount of space and the amount of time that we need to consider.

The Boolean variables that we need are:

Q[t, s]	At time t, the machine is in state s.
H[t, i]	At time t, the machine head is on square i.
S[t, i, j]	At time t, the square i contains jth symbol of Γ.

How many of each of these variables do we need? In each case, only a number that is polynomial in p(n), since:

Time from 0 to p(n): $0 \le t \le p(n)$

State from 0 to $|Q| - 1$: $0 \le s \le |Q| - 1$

Square identifier from $-p(n) + 1$ to $p(n) + 1$: $-p(n) + 1 \le i \le p(n) + 1$

Symbol number in the alphabet: $0 \le j \le |\Gamma| - 1$

1. The machine starts correctly, so at time $t = 0$:
   a. In start state: $Q[0, 0]$
   b. Head on square 1: $H[0, 1]$
   c. Input $w = \sigma_1 \ldots \sigma_n$, where σ_i is the k_i^{th} symbol of $\Sigma \subseteq \Gamma$: (For example, if $\Gamma = \{B, a, b, c\}$ and $w = ccba$, then $k_1 = k_2 = 4$, $k_3 = 3$, $k_1 = 2$.)

$$S[0, 1, k_1]$$

$$S[0, 2, k_2]$$

$$\ldots$$

$$S[0, n, k_n]$$

   d. The other possibly reachable tape squares contain blanks (assume B is encoded as 0, the first symbol of Γ).

$$S[0, j, 0] \text{ for } -p(n) + 1 \le j \le 0 \text{ and } n < j \le p(n) + 1$$

2. The machine enters the accept state (1) at some time before p(n). Here, we should take the convention that the δ transition function is defined for all states and all times up to p(n); if q is q_{accept} or q_{reject}, $\delta(q, \sigma) = (q, \sigma, \text{stay})$ for all σ—this essentially extends the transition function to remain in accepting or rejecting states once there.

$$Q[1, 1] \vee Q[2, 1] \vee Q[3, 1] \ldots \vee Q[p(n), 1]$$

or just

$$Q[p(n), 1]$$

Groups 3 to 8 are Boolean expressions to force the TM into a valid configuration at each time step t.

3. At each time t, $1 \le t \le p(n)$, M is in some legal state:

$$Q[1, 0] \vee Q[1, 1] \vee Q[1, 2] \ldots \vee Q[1, |Q| - 1]$$

$$Q[2, 0] \vee Q[2, 1] \vee Q[2, 2] \ldots \vee Q[2, |Q| - 1]$$

$$\ldots$$

$$Q[p(n), 0] \vee Q[p(n), 1] \vee Q[p(n), 2] \ldots \vee Q[p(n), |Q| - 1]$$

4. At each time t, $0 \le t \le p(n)$, M is in only one state: for each $j \ne i$, $Q[t, i] \Rightarrow \neg Q[t, j]$, $0 \le i, j \le |Q| - 1$ (this is not in CNF, but easily transformed).

5. At each time t, $1 \le t \le p(n)$, M's head is between $-p(n) + 1$ and $p(n) + 1$ (head position at time $t = 0$ already specified). Similar to 3; left as an exercise.

6. At each time t, $0 \le t \le p(n)$, M's head is on only one square. Similar to 4; left as an exercise.

7. At each time t, $1 \le t \le p(n)$, each tape square between $-p(n) + 1$ and $p(n) + 1$ contains a symbol.

$t = 1$

$$S[1, -p(n) + 1, 0] \vee S[1, -p(n) + 1, 1] \vee \ldots \vee S[1, -p(n) + 1, |\Gamma| - 1]$$

$$S[1, -p(n) + 2, 0] \vee S[1, -p(n) + 2, 1] \vee \ldots \vee S[1, -p(n) + 2, |\Gamma| - 1]$$

$$\ldots$$

$$S[1, p(n) + 1, 0] \vee S[1, p(n) + 1, 1] \vee \ldots \vee S[1, p(n) + 1, |\Gamma| - 1]$$

$t = 2$

$$S[2, -p(n) + 1, 0] \vee S[2, -p(n) + 1, 1] \vee \ldots \vee S[2, -p(n) + 1, |\Gamma| - 1]$$

$$S[2, -p(n) + 2,0] \vee S[2, -p(n) + 2, 1] \vee \ldots \vee S[2, -p(n) + 2, |\Gamma| - 1]$$

$\ldots$

$S[2, p(n) + 1, 0] \vee S[2, p(n) + 1, 1] \vee \ldots \vee S[2, p(n) + 1, |\Gamma| - 1]$

$\ldots$

$\ldots$

$t = p(n)$

$S[p(n), -p(n) + 1, 0] \vee S[p(n), -p(n) + 1, 1] \vee \ldots \vee S[p(n), -p(n) + 1, |\Gamma| - 1]$

$S[p(n), -p(n) + 2, 0] \vee S[p(n), -p(n) + 2, 1] \vee \ldots \vee S[p(n), -p(n) + 2, |\Gamma| - 1]$

$\ldots$

$S[p(n), p(n) + 1, 0] \vee S[p(n), p(n) + 1, 1] \vee \ldots \vee S[p(n), p(n) + 1, |\Gamma| - 1]$

8. At each time t, $0 \leq t \leq p(n)$, each tape square between $-p(n)$ and $p(n)+1$ contains only one symbol. For $j \neq j'$, and for all appropriate time t and square i:

$$S[t, i, j] \Rightarrow \neg S[t, i, j']$$

9. Tape squares cannot change unless the head is on that square, for each time t, $0 \leq t \leq p(n)$, each square $-p(n) \leq s \leq p(n) + 1$, and each σ in Γ, so for each j, $0 \leq j \leq |\Gamma| - 1$,

$$S(t, s, j) \wedge \neg H(t, s) \Rightarrow S(t + 1, s, j)$$

which becomes

$$\neg S(t, s, j) \vee H(t, s) \vee S(t + 1, s, j)$$

And finally, there must be clauses taken from the δ function. If at time t, the head is at h, in state s, reading symbol k, and if there are multiple choices—j of them—for the move $m_1, \ldots, m_j$ for the delta function, then the machine must be in one of these j choices at time $t + 1$.

10. At each time step t, $0 \le t \le p(n) - 1$, the machine moves to a legitimate choice from its δ function. We can start writing this as follows: suppose that in state s, reading symbol k, there are j possible moves m_i in the δ relation.

    That is, in time t, if the machine is in state s and the current symbol is k (so the head must be on a square h that contains the symbol k), then at time (t + 1) the configuration must change to one of the j choices:

    $$Q[t, s] \wedge H[t, h] \wedge S[t, h, k] \Rightarrow m_1 \vee m_2 \vee \ldots \vee m_j$$

    Each m_i will be of the form $Q[t + 1, s_i'] \wedge H[t + 1, h'] \wedge S[t + 1, h, k']$, with $h' = h \pm 1$, and s' and k' the new state and symbol from δ.

    With some effort and applications of laws from Boolean logic, these become these clauses:

    $$(\neg Q[t, s] \vee \neg H[t, h] \vee \neg S[t, h, k] \vee Q[t, s_i'])$$

    $$(\neg Q[t, s] \vee \neg H[t, h] \vee \neg S[t, h, k] \vee H[t + 1, h_i'])$$

    $$(\neg Q[t, s] \vee \neg H[t, h] \vee \neg S[t, h, k] \vee S[t, h, k_i'])$$

    (We can drop the $\wedge$'s and make these three separate clauses.)

8.8.2 Proof Part II: Construction in Polynomial Time and Correctness

1. Polynomial time: Each group of clauses consists of a polynomial number of polynomial length clauses. Thus, putting these altogether with $\wedge$'s to make a Boolean clause in conjunctive normal form, we get a long—but still polynomially bounded—clause.

2. Yes goes to yes: If the Turing machine accepts the string, then there is a path—a sequence of configurations—from the start state reading w to an accept state. That sequence will guide how to set the Boolean variables—at each step t, set them to match the configuration and the moves used. Now, the clause will be satisfied.

3. Yes comes from yes (no goes to no): Suppose that the Boolean clause is satisfiable. Then the first set of clauses says that the Turing machine began as it should, and the clause that says that it has reached q_{accept} by time p(n) says that it accepted its input.

8.9 CONCLUSION

The above transformation takes any nondeterministic TM that can reach an accept state and transforms it into a set of Boolean clauses that can be satisfied, and vice versa. This shows that any problem that is in **NP** can be polynomially transformed to CNF-SAT.

Thus, the collection of **NP**-complete problems now has its first known member.

This is critical, for now other members of **NP** can be added to the set of **NP**-complete problems just by producing a polynomial transformation from a known **NP**-complete problem. There is only one known **NP**-complete problem now to use, but there is now a starting place. Each problem that is added to the set of **NP**-complete problems will yield one more language to use as a starting point.

EXERCISES

1. Describe a certificate for each of these decision problems. Choose short certificates where possible—if your certificate is/is not polynomial, show why/why not:

   a. A map can be colored (with no two adjacent countries the same color) with four colors.

   b. A graph contains a subgraph that is a triangle.

   c. A graph can be colored with three colors so that no two adjacent nodes have the same color.

   d. A set of integers contains a subset, and the sum of the numbers in that subset is 1000.

   e. A Boolean formula in disjunctive normal form (OR of ANDs such as $x_1 \wedge \neg x_2) \vee (x_3)$) is satisfiable.

   f. An $n \times n$ matrix M is symmetric.

   g. An $n \times n$ matrix M is not symmetric.

   h. An integer n is a perfect square ($n = k^2$ for some integer k).

   i. An integer n is not a perfect square ($n \neq k^2$ for any integer k).

2. Write out yes and no instances for each decision problem in problem 1.

3. Consider this reduction from three-coloring of a graph G = (V, E) to CNF-SAT.

   For each vertex i = 1, |V|, v[i, 1] ∨ v[i, 2] ∨ v[i, 3].

   For all i, 1 ≤ i ≤ |V| and all pairs j, k, with 1 ≤ j < k ≤ 3, v[i, j] ⇒ ¬ v[i, k].

   For each edge ij in E, and for 1 ≤ k ≤ 3, v[i, k] ⇒ ¬ v[j, k].

   Is this a valid reduction? If yes, show; if not, correct.

   Is it polynomial? If yes, show; if not, correct.

   Assuming that you now have a valid polynomial reduction, and if it is known that CNF-SAT is **NP**-complete, what can you say about three-coloring?

4. Prove:

   a. The class **P** is closed under union.

   b. The class **NP** is closed under union.

5. Write out the correct form for the clauses of parts 5 and 6 in the proof of the Cook–Levin theorem—those that say that the head must be on some square, and on only one square.

6. Use de Morgan's laws to transform the clauses of part 10 of Theorem 8.19 above into the proper form.

7. Count the number of clauses of each type (1 to 10) in the proof that CNF-SAT is **NP**-complete. Your answer should be some polynomial function of p(n).

8. A number *n* is *perfect* if the sum of its factors (except *n* itself) sum to exactly *n*. For example, 28 is perfect, since its factors (1, 2, 4, 7, 14) sum to 28. But 12 is not perfect: its factors (1, 2, 3, 4, 6) sum to 16, which is not 12. The PERFECT problem has as its input a number *n* written in binary and should output yes exactly when *n* is perfect. Show that PERFECT is in **NP**.

9. The **P** and **NP** classes only apply to *decision problems*, those having yes or no answers. This restriction generally does not hurt, since almost all problems can be phrased in terms of a decision problem.

In each case below, show how a polynomial solution to the decision problem will yield a polynomial solution to the search problem.

a. Decision problem: Given n and k, does n have a factor greater than k?

   Search problem: Given n, find its prime factorization.

b. Decision problem: Given a graph G with vertices v and w and a number k, does G have a path from v to w of length at least k?

   Search problem: Given G with vertices v and w, what is the length of the longest path from v to w?

c. Decision problem: Given G = (V, E) and integer k, is G k-colorable?

   Search problem: Given G = (V, E), find the smallest k such that G is k-colorable.

10. Let Y be any nontrivial problem in **P**, the class of problems with polynomial solutions. Show that Y is **P**-complete. (A problem is *nontrivial* if it has both yes and no instances.)

11. Get rich (quick?)! See http://www.claymath.org/millennium/P_vs_NP/. (Compare with Exercise 9 in Chapter 7.)

12. Prove:

    Theorem: If $L_1 \leq_P L_2$ and L_2 is in **P**, then so is L_1.

    Theorem: If $L_1 \leq_P L_2$ and L_1 is not in **P**, then L_2 is not in **P**.

    Theorem: If $L_1 \leq_P L_2$, then $\overline{L_1} \leq_P \overline{L_2}$.

    Theorem: If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

13. Prove: Theorem: If two functions f and g are in **P**, then so are: (a) $f(n) \bullet g(n)$, (b) $f(n) + g(n)$, and (c) $f(g(n))$.

14. Give yes and no instances of CNF-SAT (different from the examples in the text) using:

    a. The single Boolean variable x_1

    b. At least two Boolean variables (x_1, x_2, …)

    Write out the encodings over $\Sigma = \{x, 0, 1, \ldots 9, \vee, \wedge, \neg, (,)\}$ of instance of CNF-SAT from:

c. Part a

d. Part b

(Describe the encoding scheme you use.)

Describe an encoding scheme and write the encodings over $\Sigma = \{0, 1\}$.

e. Part a

f. Part b

15. In a Boolean expression in CNF, with n = number of Boolean variables, m = number of clauses, and |E| = total length of expression, find:

    a. The upper bound for m in terms of n (assume no duplicate clauses).

    b. The upper bound for n in terms of m, assuming that there are no duplicate variables in a clause—so that in a clause for a variable i, either x_i is missing or one of x_i or $\neg x_i$ is present).

    c. Upper bound for |E| in terms of n and m.

16. Decision problem: Large enough factor: Given integers n and k, does n have a factor (besides n itself) that is greater than or equal to k?

    a. Give yes and no instances.

    b. Show how to use a polynomial time algorithm for this problem (if it exists) to construct a polynomial algorithm to find a factor of n. (Hint: Binary search.)

    c. From b, show how to construct a polynomial algorithm for factoring n.

Hint (for Exercise 8): To quickly add together the factors of a number, note that if we have the prime factorization

$$n = 2^a * 3^b * 5^c * \ldots$$

then the sum of all factors (including *n*) is

$$(1 + 2 + \ldots + 2^a) * (1 + 3 + \ldots + 3^b) * (1 + 5 + \ldots + 5^c) * \ldots$$

The certificate includes the prime factorization of n, followed by certificates that each of the given factors is prime. Note that there are at most $\lg n = O(s)$ prime factors, since each is at least 2 and they multiply to get n. To verify a certificate, we check that all the given factors multiply to n, and we check that all the given factors are prime using the primality certificates at the end (however they work). Then we verify that n is perfect by determining whether the sum of all factors (including n) is $2n$, which we can do using the technique outlined in the hint, which involves at most $\lg n$ additions and $\lg^2 n$ multiplications.

Hint (for Exercise 16): Given n and k, one decision problem asks whether n has any factors greater than or equal to k, aside from n itself. Given a polynomial time algorithm for this, we can find a factor of n by performing a binary search to find the smallest value of k where the algorithm returns yes; this value must be a factor of n. We can find the other factor by dividing n by k. If n is written in $s = O(\log n)$ bits, and the algorithm for the decision problem takes $O(n^d)$ time, then it will take us $O(\log n) = O(s)$ calls to find the first factor using binary search, taking so we have $O(s^{d+1})$ time, plus the time to do division, which is of course polynomial.

CHAPTER 9

More **NP**-Complete Problems

9.1 ADDING OTHER PROBLEMS TO THE LIST OF KNOWN **NP**-COMPLETE PROBLEMS

The next order of business will be adding members to the collection of **NP**-complete problems. No other proof will be like that of Cook–Levin, beginning with arbitrary members of the class **NP** and depending on the definition of **NP**-completeness. Instead, all these proofs use polynomial reducibility.

Recall that to show that Y is **NP**-complete, show that:

1. Y is in **NP**.
2. $X \leq_P Y$, for some known **NP**-complete problem X (to begin with, there is no choice, X must be conjunctive normal form satisfiability [CNF-SAT]).

9.2 REDUCTIONS TO PROVE **NP**-COMPLETENESS

These proofs of polynomial reducibility vary in difficulty from the very easy to the very hard. Garey and Johnson (1979) classify these proofs into three kinds.

9.2.1 Restriction (Also Called Generalization)

X is a special case of Y, and so every yes or no instance of X is already a yes or no instance of Y. No modifications are necessary. Thus, an identity mapping takes a yes instance of X to a yes instance of Y, and takes a no instance of X to a no instance of Y.

We have seen examples of generalization earlier: every third degree polynomial is also a fourth degree polynomial (though not vice versa). So, if we took X to be the language {p(x): p is a polynomial of degree ≤ 3 with an integer root for some integer n}, and Y is the generalization of X to include fourth degree polynomials, then a yes or no instance of X is, without modification, a yes or no instance of Y.

$$X = \{p(x)\text{: p is polynomial, degree } p \le 3,\ p(n) = 0 \text{ for some } n\} =$$
$$X = \{x^3 + 8,\ x - 5,\ x^2 - x - 6,\ \ldots\} \text{ (yes instances)}$$

$$\overline{X} = \{x^2 + 1,\ x^4 - 16,\ \ldots\} \text{ (no instances)}$$

$$Y = \{p(x)\text{: p is polynomial, degree } p \le 4,\ p(n) = 0 \text{ for some } n\} =$$
$$Y = \{x^3 + 8,\ x - 5,\ x^2 - x - 6,\ x^4 - 16\ \ldots\} \text{ (yes instances)}$$

$$Y = \{x^2 + 1,\ \ldots\} \text{ (no instances)}$$

Boolean expressions in conjunctive normal form are certainly Boolean expressions, so CNF Boolean expressions are special cases of general Boolean expressions, and GEN-SAT is a generalization of CNF-SAT.

Theorem 9.1

GEN-SAT (or general first-order Boolean expression satisfiability) is **NP**-complete.

Proof

An instance of GEN-SAT is a Boolean expression E in variables $x_1, \ldots, x_n$, using all the standard connectors ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $=$), not in any particular form. Again, the certificate is an assignment of truth values to the $x_1, \ldots, x_n$ variables, and therefore it is easy to show (so do it!) that this certificate

can be used to verify a yes instance in time bounded polynomially in the length of the expression |E|.

Now an expression in conjunctive normal form is certainly a Boolean expression, so that

$$\text{CNF-SAT} \leq_P \text{GEN-SAT} \qquad \blacksquare$$

This is a proof by generalization—or what Garey and Johnson call restriction. GEN-SAT is a more general case of CNF-SAT, so the polynomial transformation is very easy.

Note: Many texts use the abbreviation SAT for conjunctive normal form satisfiability. This is tempting since it is probably the most used **NP**-complete problem. In this text, the simple term SAT is not used, so that it is clear what (if any) conditions are placed on the Boolean expressions.

9.2.2 Local Replacement

In a proof of this kind, one takes reasonably small subparts of the known **NP**-complete problem, and transforms each subpart into a piece of the new problem. The term *gadget* is often used for the pieces constructed by this transformation.

Theorem 9.2

3-CNF-SAT (Boolean satisfiability in which each clause in conjunctive normal form has three terms, not necessarily distinct) is **NP**-complete.

Proof

Using local replacement we can take each clause of a yes or no instance in CNF-SAT, and transform that clause into a group of clauses, each of which has three terms. The original set of clauses is satisfiable if and only if the transformed set of clauses is also satisfiable.

The conversion will depend on the length k of the clause to be converted. For $k \leq 3$, the method is straightforward.

If $k = 1$, repeat the term three times; so (x_i) becomes $(x_i \vee x_i \vee x_i)$.

If $k = 2$, duplicate the second term: so $(x_i \vee \neg x_j)$ becomes $(x_i \vee \neg x_j \vee \neg x_j)$.
If $k = 3$, no change is needed, as the clause is already in 3-CNF.

If $k > 3$, the original clause will be of the form $x_{j1}, \ldots, x_{jk}$ (where any of the terms might be the negation with $\neg$). The transformation of this clause requires the creation of $r = (k - 3)$ new variables $y_1, \ldots, y_r$. The idea is to "pull apart" the original clause into multiple clauses bridged by the y_i's and their negations as follows:

$(x_{j1} \vee x_{j2} \vee y_1)$	First two terms of the clause
$(\neg y_1 \vee x_{j3} \vee y_2)$	Third term of the clause
$(\neg y_2 \vee x_{j4} \vee y_3)$	Fourth term of the clause
…	
$(\neg y_r \vee x_{j(k-1)} \vee x_{jk})$	Final two terms of the clause

For example, the clause $(x_3 \vee \neg x_2 \vee x_1 \vee x_4 \vee \neg x_3 \vee \neg x_1)$ becomes:

$(x_3 \vee \neg x_2 \vee y_1)$
$(\neg y_1 \vee x_1 \vee y_2)$
$(\neg y_2 \vee x_4 \vee y_3)$
$(\neg y_3 \vee \neg x_3 \vee \neg x_1)$

YES GOES TO YES

For clauses of length $k \leq 3$, the true term that appears in the old clause also appears in the new clause, and so the new clause is still true.

For clauses of length $k > 3$, we have a whole new set of clauses, and they must all be true. At least one of the x_{ji} from the original clause was true. There will be a way to choose truth values for the y_i so that the new clauses are true.

If x_{i1} or x_{i2} was true:
  Set y_1 to false.
  Set all of the other y_j to true.
If x_{ir} was true for $2 < r < (j_k - 2)$:
  Set y_m $1 \leq m < r$, to false.
  Set all of the other y_j to true.
If x_{jk-1} or x_{jk} was true:
  Set y_r to true.
  Set all of the other y_j to false.

NO GOES TO NO (OR, EQUIVALENTLY, YES COMES FROM YES)

As is often the case, it will be easier to prove the contrapositive; that is, a yes instance of the transformed clauses must have come from a yes instance of the original clause. (Remember this, as it will occur often.)

So, suppose that each of the clauses on the right is satisfiable with some assignment of truth values to the $x_{j1}, \ldots, x_{jk}$ and the $y_1, \ldots, y_r$. For clauses of length up to three, the new clause is really identical to the old: if one is true, so is the other. For the clauses containing the $y_1, \ldots, y_r$, the key is that the set of clauses cannot be made true just by choosing truth values for the y_j because of the way variables and their negations are threaded through the new clauses.

This can be proved on a case-by-case basis. Try to make all the clauses true, starting from left (y_1 true), right (y_r false), or middle (either y_m true or y_{m+1} false). Wherever you start, propagate left or right (or both), attempting to make all the clauses true using only the $y_1, \ldots, y_r$. This cannot be done. ■

9.2.3 Component Design

Sometimes there is no way to take the local properties of one decision problem and make changes to each of them to arrive at a valid transformation so that $X \leq_P Y$.

If this cannot be done, the whole structure of X must be considered. The problem Y may be constructed out of many parts of X, with changes, reorganizations, additions, or deletions. This often takes considerable ingenuity, and leads to long, complex, hard-to-read proofs. One proof by component design may look nothing at all like another proof; knowing how to do one reduction may be no help in doing others.

We actually have seen a component design transformation in the Cook–Levin theorem. Here, moves and paths in a nondeterministic Turing machine (NDTM) were transformed into Boolean expressions. States, moves, configurations, and the accepting condition became expressed in a large set of nontrivial Boolean clauses. Other examples of component design will be somewhat shorter. No other proof will require an entire chapter, although some of them can be complicated.

Note that 3-CNF-SAT is more restrictive than CNF-SAT, so that we could use the identity mapping to show that

$$\text{3-CNF-SAT} \leq_P \text{CNF-SAT}$$

This says nothing about the **NP**-hardness of 3-CNF-SAT, though it would prove that 3-CNF-SAT is in **NP**.

Theorem 9.3

3-DISTINCT-CNF-SAT (Boolean expressions in CNF in which each clause has three distinct terms) is **NP**-complete.

Proof

Exercise. ■

9.3 GRAPH PROBLEMS

So far, we have discussed only problems involving Boolean expressions. These form a large group of important decision problems, but there are many other kinds of problems. Some of the most interesting problems come from graph theory, for example, the classic traveling salesman (or woman!) who wants to visit n cities, traveling the least distance possible, while making the circuit. However, as noted before, as stated the Traveling Salesman Problem (TSP) is not a decision problem—but it has a closely related version that is a decision problem.

The first graph problem is not as intuitive or as easy to state as TSP. It—like other **NP**-complete graph problems—will require some definitions that you probably have not met before, but which are not difficult.

Graphs can be undirected or directed, as shown in Figure 9.1.

In this book, unless otherwise specified, a graph denotes an undirected graph. Graphs consist of nodes or vertices $v_1, \ldots, v_n$ and edges $\{v_i, v_j\}$ or, for directed graphs, directed edges or "arrows" (v_i, v_j). We will often write G = (V, E); that is, G is composed of a set v of vertices $v_1, \ldots, v_n$, $|V| = n$, and a set of edges E, $|E| \leq n^2$. We will write directed graphs as G = (V, A) with arrows instead of edges.

Graphs may also be weighted, in which case there is some positive number (a weight) associated with each edge. Weighted graphs are often undirected complete graphs. A graph G is **complete** if there are edges between all pairs of vertices.

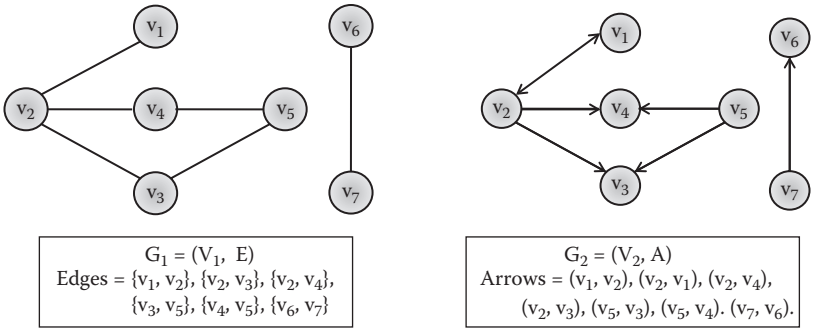


FIGURE 9.1 Example undirected and directed graphs.

Definition 9.4

Two vertices are **adjacent** if there is an edge between them. ■

Definition 9.5

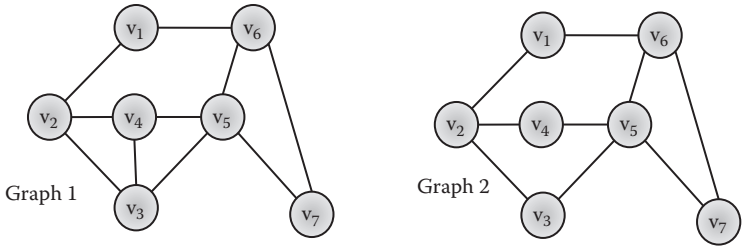
A (simple) **path** in G from v_i to v_j is a sequence of nodes ($v_i = v_{k1}, \ldots, v_{km} = v_j$) so that:

1. There exists an edge between each pair ($v_{kr}\, v_{k(r+1)}$) in E, $1 \le r \le m$.
2. No node is repeated ($v_{kr} \ne v_{ks}$, $r \ne s$) except possibly the first and last ($v_i = v_j$). ■

Sometimes, paths are allowed to have repeated vertices—be non-simple—but in what follows, we will assume that paths are simple unless stated otherwise.

Figure 9.1 showed a graph with a list of its edges. There is an easy simple path from v_1 to v_5; there is no path from v_1 to v_6. Graph G_1 in Figure 9.1 is not connected since (for example) there is no path from v_2 to v_6. However, it has two connected subgraphs $\{v_1, v_2, v_3, v_4, v_5\}$ and $\{v_6, v_7\}$, often called its connected components.

A path in a directed graph has arrows between successive nodes, rather than edges. A directed graph is shown in Figure 9.1.



GRAPH 1:
Hamiltonian Circuit: $v_1, v_6, v_7, v_5, v_4, v_3, v_2, v_1$ (and others)
Hamiltonian Path: $v_1, v_6, v_7, v_5, v_4, v_3, v_2$ (and others)
GRAPH 2:
Hamiltonian Circuit: None
Hamiltonian Path: $v_1, v_6, v_7, v_5, v_4, v_2, v_3$ (and others)

FIGURE 9.2 Examples of Hamiltonian circuits and paths.

A path is a **circuit** if the first and last nodes are the same. In G_1, (v_3, v_5, v_4, v_2, v_3) is a circuit. A **Hamiltonian circuit** (in a directed or undirected graph) is a circuit that visits every vertex in the graph exactly once. A **Hamiltonian path** is a path from some vertex v_i to a different vertex v_j that visits all other vertices en route. See Figure 9.2 for examples.

Note that both graphs in Figure 9.2 are connected; there can never be a Hamiltonian circuit or a Hamiltonian path unless the graph is connected.

We need to also consider the encoding of a graph $|G|$, since to show something is in **NP**, we will have to know the length of the encoding and find a certificate that is polynomial in this length $|G|$.

There are at least three different (but of course related) standard ways to encode G, whether directed or undirected:

1. An $n \times n$ adjacency matrix, where the ij^{th} entry is 1 if there is an edge from i to j, and 0 otherwise. For an undirected graph, the matrix is symmetric and there are no i, i entries, so only $n(n - 1)/2$ entries need to be stored. For a weighted graph, the ij^{th} entry is the weight on that edge (and either 0 or a very large number if there is no edge).

   Size of the encoding: $|G| = n^2$ for directed graphs (or $n^2 - n$ if there can be no arrow from a vertex back to itself), $(n^2 - n)/2$ for undirected graphs.

   Example: G_1 from Figure 9.2 represented as an adjacency matrix (note that it is symmetric):

0	1	0	0	0	1	0
1	0	1	1	0	0	0
0	1	0	1	1	0	0
0	1	1	0	1	0	0
0	0	1	1	0	1	1
1	0	0	0	1	0	1
0	0	0	0	1	1	0

2. An adjacency list: Each vertex v_i has an attached list of all vertices connected to it. In an undirected graph, each edge e_{ij} will have v_i appear in v_j's list and v_j appear in v_i's list. This is a duplication of information, but eliminating the duplication would make algorithms more complicated and would only halve the length of the representation of $|G|$. As elsewhere in the book, two encodings (or two algorithms) whose length only differs by a constant factor will be considered essentially the same size.

   Size of encoding: $|V| + 2|E|$ (each edge appears twice, but only one end is in the list at each appearance).

   Example: G_1 stored as adjacency lists:

   1: 2, 6

   2: 1, 3, 4

   3: 2, 5

   4: 2, 5

   5: 3, 4, 6, 7

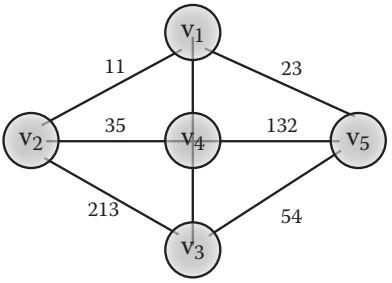
   6: 1, 5, 7

   7: 5, 6

3. A count of the vertices (n) followed by a list of the edges. Here, n would not be necessary if every vertex was connected to something, but that is not necessarily the case. In this representation:

   Size of encoding: $|G| = 1 + 2|E|$ (each edge appears once, but with two vertices).

   Example: See the edge list for G_1 in Figure 9.2.



Weighted Graph $G_3 = (V, E)$

1	**11**	**0**	**0**	**23**
11	**1**	**213**	**35**	**0**
0	**213**	**1**	**0**	**54**
0	**35**	**0**	**1**	**132**
23	**0**	**54**	**132**	**1**

OR (∞ = 32000 = very expensive travel i to j)

0	**11**	**∞**	**∞**	**23**
11	**0**	**213**	**35**	**∞**
∞	**213**	**0**	**∞**	**54**
∞	**35**	**∞**	**0**	**132**
23	**∞**	**54**	**132**	**0**

FIGURE 9.3 Weighted graph and two possible representations.

A subtle assumption is made here, and you should be questioning why one can encode $|V| = n$ in the above expression $1 + 2|E|$ in one character. You cannot, and a better (and very close) estimate would be the smallest integer greater than $\log_{10} |V|$. If, for example, the graph had 7285 nodes, and one were using decimal, it would take a length of 4 to encode $|V|$. But $\log_{10} n$ is much smaller than n, and is essentially treated as a small constant and ignored.

So, what is a good estimate for the size of $|G|$ in whatever encoding? Certainly, $0 \le |E| \le n(n - 1) < n^2$, where $|V| = n$. Any "interesting" graph has more than a few edges. Thus, we will generally take n^2 as the size of the encoding of G. To prove something is in **NP**-complete, we will have to show that there is a certificate that is polynomial in $\mathbf{O}(n^2)$—or, if we want to use the other characterization, that there is an NDTM of polynomial order in n^2. Note, though, that a function is polynomial in n^2 if and only if it is polynomial in n (a higher order in n, of course, but all we care about is a polynomial bound). Thus, we will often look for something polynomial in $n = |V|$, although that is not exactly the size of $|G|$.

An example weighted graph (with two possible matrix encodings) is shown in Figure 9.3.

9.4 VERTEX COVER: THE FIRST GRAPH PROBLEM

Definition 9.6

For $G = (V, E)$, a set $V' \subseteq V$ is a **vertex cover of size k = |V|** if every edge has at least one end in V'. ■

Clearly, the set $V' = V$ is a vertex cover of size $|V|$ for any graph G. Also clearly, no graph (except the graph with no edges) can have a vertex cover of size 0. For other numbers m between 1 and $|V| - 2$, a graph may or may not have a vertex cover of size m.

Finding vertex covers and determining whether they exist for a given m is easy for small graphs. However, imagine a graph with 100 vertices and thousands of edges. Does it have a vertex cover of, for example, 50? Perhaps yes, perhaps no, but there are a lot of possible vertex covers to try (100!/(50!50!) to be exact), and for each of these possible covers, a lot of edges need to be checked.

The Vertex Cover (VC) problem is the decision problem: Given a graph $G = (V, E)$ and a number $0 \leq k < |V|$, does G have a vertex cover of size k?

Theorem 9.7

The VC problem is **NP**-complete.

Proof

1. Vertex cover is in **NP**. The certificate c is the list of m vertices that form the vertex cover, that is, it is the set V'. It is important that we do not have to find a certificate: we are given the list, and all we have to do is to check or verify that it is indeed a cover.

   VERTEX-COVER-VERIFY (G, m, c) =

   Check that each element of c is an element of V in G. If not, reject.

   Check that each element of c is distinct. If not, reject.

   Check that $|c| < m$ (of course, if $|c| = m$, we can easily find the vertex cover of size m).

   Check that the vertices of c form a vertex cover for G. If yes, accept.

   The first three steps are simple to compute in a polynomial time.

   For each of the edges (and there are fewer than n^2 edges), the last step of the verifier goes through the list of $m < n$ vertices in the certificate, checking that one end of the edge is in c. Since there are at most n^2 to check and each requires walking a list of length $< n$, the last step is bounded by $O(n^3)$.
2. 3-CNF-SAT $\leq_p$ VC. This is another proof by component design, it is made more interesting since we are converting Boolean expressions

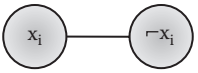


FIGURE 9.4 Example "bar" for a variable.

to a graph. Thus, we have to be ingenious and make a graph out of the pieces of the Boolean expression.

Note that a yes or no instance of VC consists of a graph and a number m. We construct the mapping by letting

$$|V'| = m = n + 2c$$

where n is the number of variables in the Boolean expression and c is the number of clauses in the expression.

A typical graph made from a Boolean expression will often have components made from the Boolean variables $x_1, \ldots, x_n$. A standard trick is to make a "bar" or "variable gadget" corresponding to each x_i; a bar for x_i is just the two vertices x_i and $\neg x_i$ connected by an edge, as in Figure 9.4.

For each of the clauses in 3-CNF-SAT, construct a triangle (clause gadget) labeled with the three terms from the clause. Some triangles may have multiple identical labels. Some labels are negated variables, others are not.

For example, the clause $(x_1 \vee x_1 \vee \neg x_4)$ would correspond to the triangle in Figure 9.5.

Finally, connect the bars to the triangles in the only natural way: connect each term in the triangle to the identical term in a bar. An example is shown in Figure 9.6.

Claims to be proved:

1. The component design is a polynomial transformation.
2. A yes instance of 3-CNF-SAT goes to a yes instance of VC.

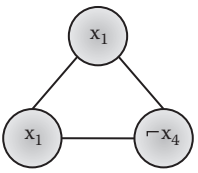


FIGURE 9.5 Triangle subgraph for the clause $x_1 \vee x_1 \vee \neg x_4$.

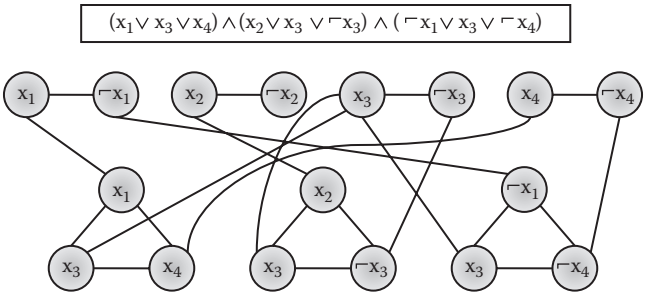


FIGURE 9.6 Example of a 3-CNF expression and its associated graph.

3. A yes instance of VC came from a yes instance of 3-CNF-SAT (which is equivalent to no goes to no).

1. The transformation is polynomial since its size is:

$|V| = 2 * (\#$ of variables in the Boolean expression$) + 3 * (\#$ of clauses$)$

$|E| = \#$ of variables $+ 6 * (\#$ of clauses$)$

2. Suppose the Boolean expression is satisfiable. Take the assignment of truth values to the x_i that makes all the clauses true so that at least one term is true in each clause.

   We claim that we can make a vertex cover of size $m = n + 2*|C|$ from this set of truth values.

x_1	x_2	x_3	x_4
T	F	T	F

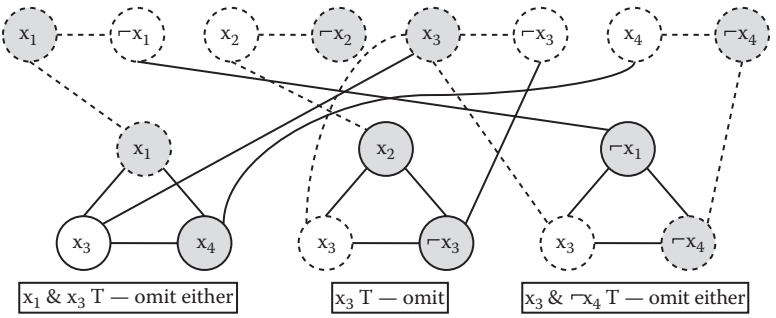


FIGURE 9.7 Vertex cover from a satisfying truth assignment.

Each bar: Look at the assignment of truth values that satisfy the equation. Put the vertex into V′ that corresponds to the end that is true. As shown in Figure 9.7, this adds n = 4 vertices (the shaded vertices in the top row) to V′.

Each triangle: There is (at least) one true variable in each clause. Put the other two triangle vertices into the vertex cover. This adds 2|C| vertices to the prospective cover set V′, shown in Figure 9.7 as the 2|C| = 6 shaded vertices in the triangles.

In short, for the vertex cover:

Bar: Choose TRUE end.

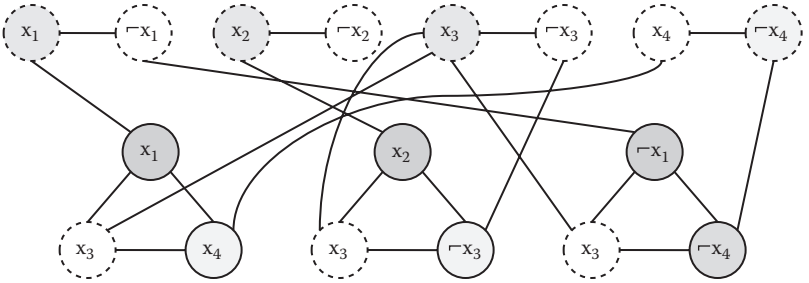
Triangle: Omit one TRUE vertex.

Now, this is a vertex cover. Each bar has an end (the true end) in V′. At least one of the vertices in each triangle is connected to the true end of a bar in order to satisfy the clause. As depicted in Figure 9.7, each triangle also has two vertices in V′, and any two vertices in a triangle cover the remaining edges in the triangle.

And, because a TRUE vertex was omitted from each triangle, the edge from this vertex to a bar is covered by the TRUE end of the bar. The other two edges from the triangle up to the bars are covered by the two vertices in the triangle.

3. To finish the proof, we need to show that if the graph—as constructed—has a vertex cover, then the Boolean expression was satisfiable (that is, yes comes from yes). This proof relies on the constraints on how the vertex cover can be chosen. While there are many ways to choose m = n + 2|C| vertices from G, the only possible way for them to be a vertex cover is to have a V′ that includes one vertex from each bar, and two vertices from each triangle (there is simply no other way to have a cover of a triangle!). If you can have a successful vertex cover, then the lines interconnecting the bars and the triangles must have one end in the vertex cover. Two of these interconnecting lines are covered by the two vertices chosen from each triangle, but the third must be covered by the vertex chosen from the bar—and that means setting that vertex to be TRUE will guarantee at least one true variable in each clause. ■

For example, in Figure 9.8, we find a vertex cover of size 4 + 6, which must have one vertex from each bar and two vertices from each triangle. Since it is a vertex cover, the edges between bars and triangles are also



Successful vertex cover—one end of each bar, two vertices from each triangle—covers all bars, all triangles, and all lines between bars and triangles. This means $x_1 = x_2 = x_3 = T$, $x_4 = F$ is a satisfying truth assignment.

FIGURE 9.8 Vertex cover (which implies a satisfying truth assignment).

covered. That means that one edge to each triangle is covered from the bars. Set the bar end of each inter-bar-triangle edge to be true; this is a satisfying assignment.

You should construct a no instance of 3-CNF-SAT and the corresponding graph. Convince yourself that there is no vertex cover of the correct size. See the Exercises for one simple example.

9.5 OTHER GRAPH PROBLEMS

The following graph definitions are used in many **NP**-complete problems. Some such problems are listed with their definitions.

Definition 9.8

An **independent set** of size m in G is a set of vertices V′, |V′| = m, such that no two vertices of V′ are connected by an edge. ■

Problem

Given G and m, does G contain an independent set (IS) of size m? ■

Definition 9.9

A **feedback vertex set** of size m in a directed graph G is a set of vertices V′ such that every circuit in G contains a vertex of V′. ■

Problem

Given a directed graph G and an integer m, does G contain a feedback vertex set (FVS) of size m? ■

Definition 9.10

A **k-clique** in a graph G is a complete graph on k vertices that is a subgraph of G. ■

Problem

Given a graph G and an integer k, does G contain a k-clique? ■

Definition 9.11

A **dominating set** of size m in a graph G is a set of vertices V′ such that every vertex of G is connected by an edge to some vertex in V′. ■

Aside: How is this definition different from a vertex cover? In a triangle graph, what is the smallest vertex cover? What is the smallest dominating set?

Problem

Given a graph G and an integer m, does G contain a dominating set of size m? ■

Definition 9.12

A graph G = (V, E) can be **colored** with m colors $c_1, c_2, \ldots, c_m$ ($1 \le m \le |V|$) if there is a way to assign the colors c_i to the vertices v_j such that no two adjacent vertices have the same color. Such an assignment of m colors to V is called an **m-coloring**. ■

Problem

1. Given a graph G, does G have a three-coloring?
2. Given a graph G and an integer k, does G have a k-coloring? ■

Definition 9.13

The **degree** of a vertex v in V is the number of edges connected to v. (See the Exercises at the end of the chapter for some problems using vertex degree.) ■

The proofs that these problems are **NP**-complete vary from easy to difficult. Also interesting is that some similar problems are not **NP**-complete. For example, there is a simple polynomial algorithm to determine whether a graph G has a two-coloring.

9.6 HAMILTONIAN CIRCUIT (HC)

Hamiltonian circuits (and the related traveling salesman problems) are old and interesting graph problems. Recall that HC asks if one can start at some vertex, visit all the other vertices in the graph, and return to the starting place? For many graphs, the answer is clearly no. There can be no Hamiltonian circuit if there are any isolated vertices, or if there are any vertices with only one edge, for example. For small graphs, it is easy to check by inspection whether or not there is a Hamiltonian circuit. For example, in Figure 9.9, the graph on the left is a yes instance; the graph on the right is a no instance.

The HC problem is the decision problem: Given a graph G = (V, E), does G have a Hamiltonian circuit?

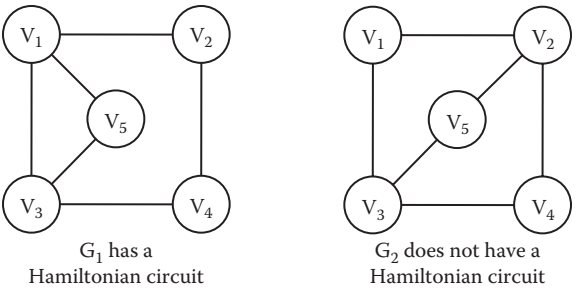


FIGURE 9.9 Yes-instance and No-instance of HC.

Theorem 9.14

HC is **NP**-complete.

The proof is in four parts:

1. HC is in **NP**
2. VC $\leq_p$ HC (construct transformation, show it is a polynomial)
3. VC $\leq_p$ HC (yes goes to yes)
4. VC $\leq_p$ HC (yes comes from yes)

1. HC is in **NP**.

   The certificate c is obviously the circuit. Without loss of generality, we can assume that any circuit starts and ends at the first vertex v_1, so there is no need to include v_1 in the certificate. Now, what do we need to check?

   HAMILTONIAN-CIRCUIT-VERIFY(G, c) =

   Check that each element of c is a vertex in G.
   Check that each element in c is distinct.
   Check that $|c| = n - 1$.
   Check that v_1 is not in the certificate.
   Check that $\{v_1, c_1\}$ is an edge in V.
   Check that $\{c_{n-1}, v_1\}$ is an edge in V.
   For each c_i and c_{i+1}in c, $1 \leq i \leq n - 2$.
   Check that $\{c_i, c_{i+1}\}$ is an edge in V.

   Clearly, each of the steps needed to verify the certificate is polynomially bounded.
2. VC $\leq_p$ HC.

   An instance of VC has a graph G = (V, E) and an integer m. We will use a (fairly complex) version of local replacement to build a new graph G′ = (V′, E′) so that G′ has a Hamiltonian circuit if and only if G has a vertex cover of size m.

   a. The integer m in VC becomes m vertices $a_1, \ldots, a_m$. There are no edges from any a_i to any a_j for I ≠ j).
   b. Each edge $\{v_i, v_j\}$ in G becomes a gadget, which is a set of 12 vertices (in two rows of 6, one row labeled $v_{i,j,i,1}$ to $v_{i,j,i,6}$, the other labeled $v_{i,j,j,1}$ to $v_{i,j,j,6}$) and 12 edges (down both sides, four edges

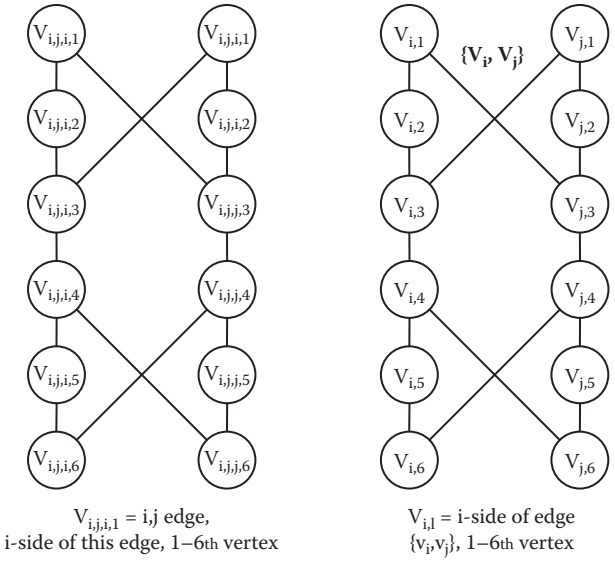


FIGURE 9.10 Subgraph (in full and abbreviated form) for edge $v_i - v_j$.

on each side, with four more edges making two X shapes), as shown in Figure 9.10 for the edge $\{v_i, v_j\}$.

Note: The first subscript i, j is only necessary to differentiate the vertices v from edge {i, j} from the six very similar vertices that come from edge {i, k}. We will drop that pair of subscripts in most of our diagrams. See the simpler figure on the right side of Figure 9.10.

We will call these complicated subgraphs "flags" since they have some resemblance to maritime signal flags.

The flags will be connected to other parts of the graph only at the four corners. This means that any path visiting the nodes in the flag enters at one corner, exits at another corner, and there are only two ways then to visit all the nodes in a flag. One way would be to only visit half of the vertices on this pass—come in at one corner, go straight up or down and out—and to visit the other side on a different pass (left side of Figure 9.11). Another path would be to come in at a corner (for example, upper left), go down, over and up, down, back across, and out the corner on the same side as the corner one came in on. Of course, this has a similar path coming in at upper right. These options are the second and third shown in Figure 9.11.

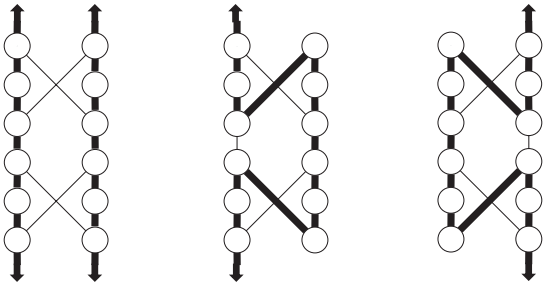


FIGURE 9.11 Three ways to visit all vertices of a "flag" subgraph.

There is no other way to visit all the nodes, either in one, two, or three or more passes. (Check this for yourself!)

c. Connecting the flags: Take each vertex v_i in the original graph G. There will be zero or more edges out of v_i. Order the edges out of v_i in any order (one way is to start with any one first, then order clockwise). Connect the v_i sides in this order (last of the first to first of the next; last of this one to first of the next, etc). The example in Figure 9.12 shows how the flags are connected for a vertex v_5 that is connected to three vertices v_1, v_2, and v_4 with the

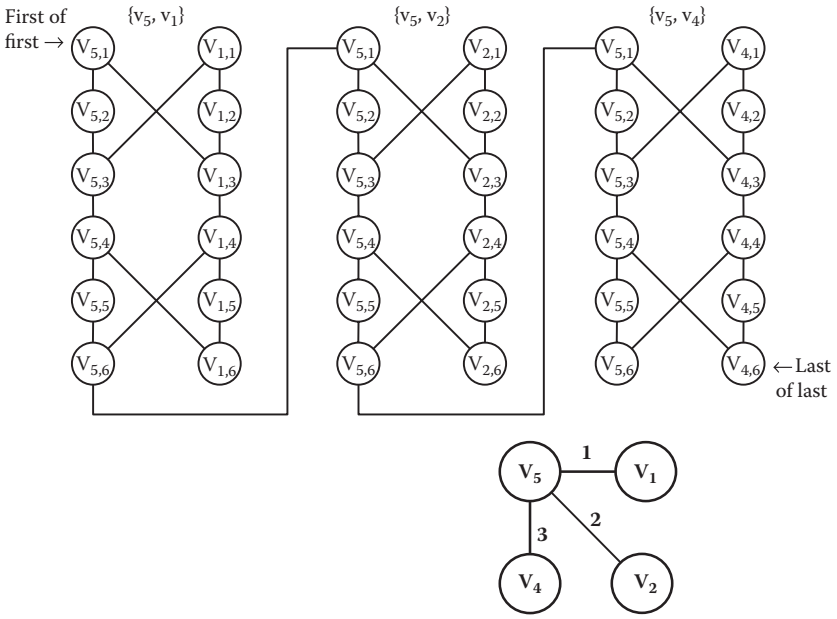


FIGURE 9.12 Interconnections of the edges to v_5 using the ordering shown.

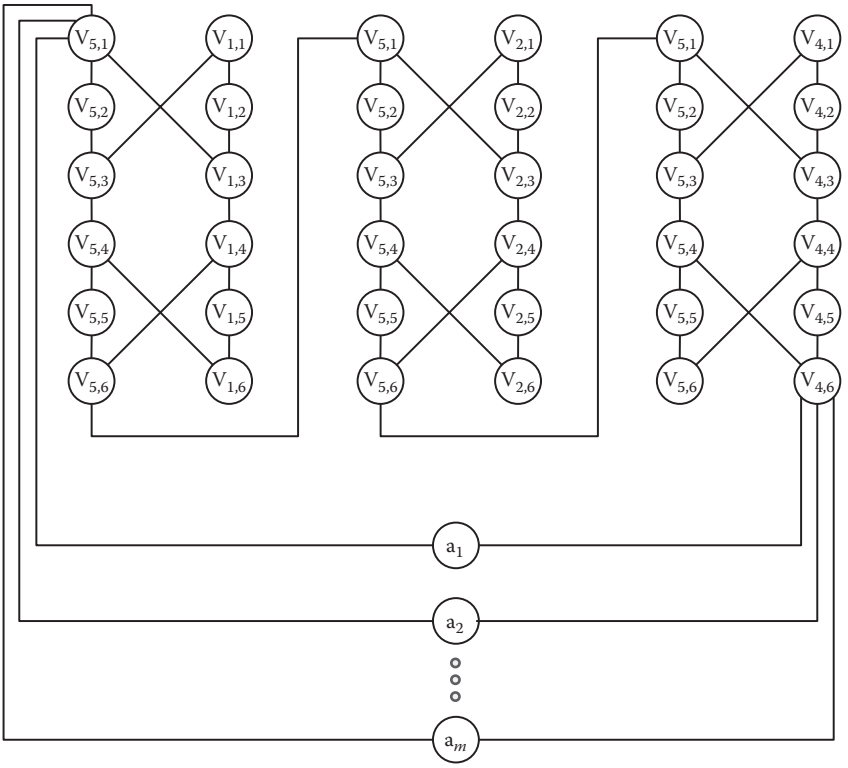


FIGURE 9.13 Connect v_5 edge subgraphs to all the a_i.

edge ordering $\{v_5, v_1\}$, $\{v_5, v_2\}$, $\{v_5, v_4\}$. Any other ordering would be allowed, but would lead to a different graph.

d. Connecting the flags to the a_i: For each vertex v_j, we have "strung" together the edges leaving v_i. Connect the first vertex of the first edge and the last vertex of the last edge to every one of the $a_1, \ldots, a_m$ vertices. Think of this as "first of the first and last of the last to each a_i." Figure 9.13 shows one set of flags connected to all of the a_i vertices.

   Claims to be proved:

   i. This example of component design is a polynomial transformation.
   ii. A yes instance of VC goes to a yes instance of HC.
   iii. A yes instance of HC comes from a yes instance of VC (note that this is the same as a no instance of VC goes to a no instance of HC).

3. Left to the reader (see the Exercises).

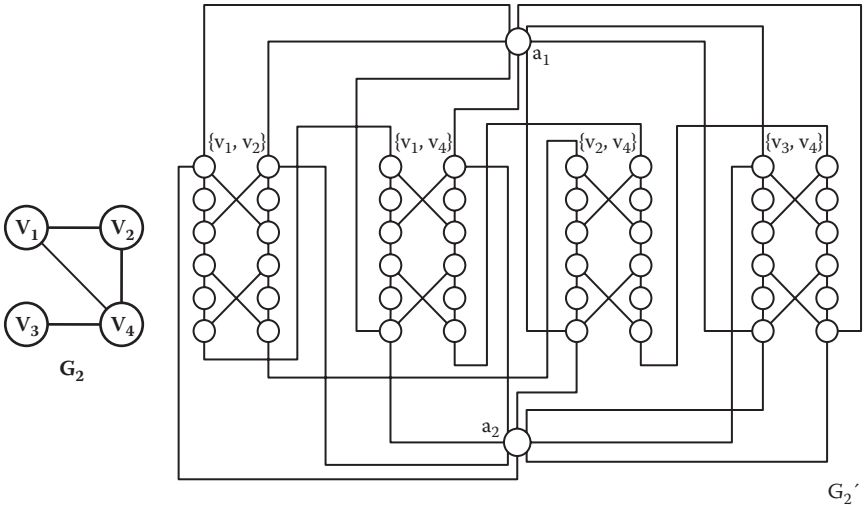


FIGURE 9.14 Final graph built from G for a vertex cover of size 2.

4. If the original graph G has a vertex cover of size m, then the constructed graph G′ has a Hamiltonian circuit. Consider the simple graph G_2 in Figure 9.14, which has a vertex cover $V' = \{v_2, v_4\}$ of size $m = 2$, shown with the graph G_2' built using the component design transformation just described.

   Note there are four gadgets, one for each of the edges in the original graph G_2, and two external vertices for $m = 2$, a yes instance of VC for G_2.

   The graph G_2' has a Hamiltonian circuit as shown in Figure 9.15. As there are only two vertices a_1 and a_2, the circuit is done in two loops, one pass from a_1 to a_2 and another from a_2 to a_1. As v_2 and v_4 in the VC graph G_2 are both adjacent to v_1, it is possible to make two passes through the gadget corresponding to either edge $\{v_1, v_2\}$ or $\{v_1, v_4\}$, whereas only a path from v_4 can reach v_3. This is shown in Figure 9.15.
5. Left to the reader (see the Exercises). ■

Corollary 9.15

Directed Hamiltonian Circuit (DirHC) is NP-complete. (This is the question: Does a directed graph G have a directed Hamiltonian circuit?) ■

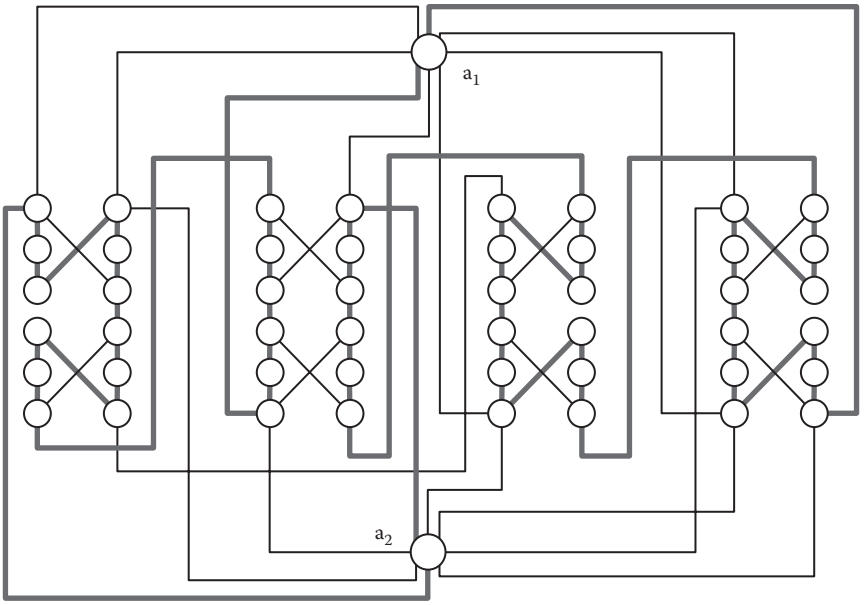


FIGURE 9.15 Successful Hamiltonian circuit in G′.

Proof

Left to the reader (see the Exercises). Hint: There are standard transformations from graphs to directed graphs and vice versa. The obvious transformation (one edge becomes two arrows) often works from a graph to a directed graph. For directed to undirected, one can try changing arrows into edges, but this generally loses too much information. A better choice is to replace each vertex by a trio of vertices, so v becomes three vertices $v_{in} - v_{mid} - v_{out}$ connected as shown, and the arrow (v, w) becomes the edge v_{out} to w_{in}.

Corollary 9.16 (K-TSP or Decision Problem TSP)

Decision problem TSP is **NP**-complete. (Given a weighted graph G with vertices $v_1\ v_2 \ldots v_n$ and edges e_{ij} with positive integer weight w_{ij} on each edge e_{ij}, and a bound K, does G have a path that starts at v_1, visits every vertex of G, and returns to v_1 such that the weighted length of the path is less than K?) ■

Corollary 9.17

The Traveling Salesman Problem (TSP) is **NP**-hard. [Given a weighted graph G with vertices v_1 v_2 … v_n and edge e_{ij} with positive integer weight w_{ij} between each pair of vertices v_iv_j, what is the length of the shortest path that starts at v_1, visits every vertex of G, and returns to v_1?] ■

Note: Since this problem is not a decision problem, it is not a candidate for the class of **NP**-complete problems, but if one could solve this, one would have a solution to the K-decision TSP.

9.7 EULERIAN CIRCUITS (AN INTERESTING PROBLEM IN **P**)

An interesting side note is the problem of Eulerian circuits. An Eulerian circuit is a path through the graph from v_1 back to v_1 such that every edge is included once in the path. Eulerian circuits are not required to be simple paths; they can visit a vertex more than once.

The famous Bridges of the Königsberg puzzle can be transformed into an Eulerian circuit problem. Can someone take a Sunday stroll in the city of Königsberg circa 1700 that would cross each bridge exactly once? The transformation is made by making vertices out of islands and mainlands, and connecting land masses if there is a bridge from one to the other. Figure 9.16 shows a map of 1700's Königsberg with the associated multigraph (a multigraph has more than one edge between some pairs of vertices).

It is easy to prove that an Eulerian circuit exists if and only if every vertex has an even degree. This test can obviously be done in polynomial time.

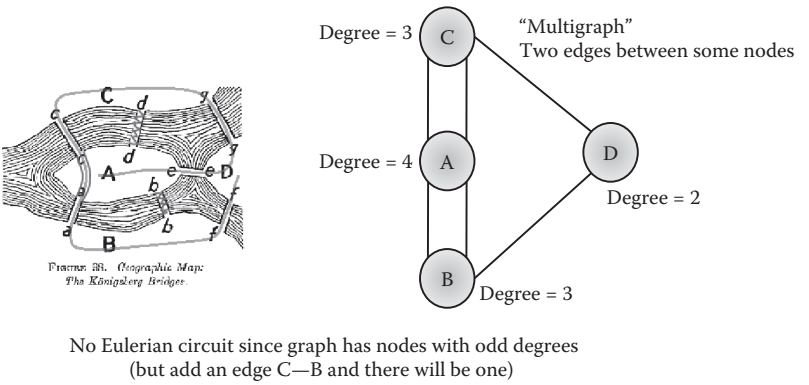


FIGURE 9.16 (Source: Kraitchik, M., *Mathematical Recreations*, New York: W.W. Norton, 1942, pp. 209–211, Section 8.4.1. Used with permission.)

9.8 THREE-DIMENSIONAL MATCHING (3DM)

The three-dimensional matching problem is best understood by first considering a simpler example, the two-dimensional matching (2DM) problem, which is often stated as the marriage problem. There is a relatively simple polynomial algorithm for solving 2DM, so 2DM is highly unlikely to be **NP**-complete.

Suppose you have two sets X and Y (perhaps women and men), of the same size, $|X| = |Y|$. Now you also have a third set P consisting of ordered pairs from X and Y, so that $P \subseteq X \times Y$. One can think that (x, y) in P means that x is willing to marry y, and vice versa. The idea is to find a suitable set of marriages so that everyone is paired with someone they are willing to marry. Thus, $|P|$ must be at least $|X|$ for a pairing to occur.

So, if

X = {Ann, Betty, Carol}

Y = {David, Ed, Frank}

P = {(Ann, Frank), (Ann, Ed), (Betty, Ed), (Carol, David), (Carol, Ed)}

then a matching is

{(Ann, Frank), (Betty, Ed), (Carol, David)}

The three-dimensional matching problem has three sets X, Y, Z and $P \subseteq X \times Y \times Z$. Perhaps we are on some planet where there are three genders, or perhaps X is the women, Y is the men, and Z is the pets, so that (Ann, David, Fido) in P means that Ann, David, and Fido are willing to live together.

The proof the 3DM is **NP**-complete is one of the more complicated component design proofs. The proof is due to Richard Karp (1972).

Many set problems can be proven **NP**-complete by reducing 3DM to the set problem in question. Often, you just need to drop the ordering so that triples are viewed as sets). An example is exact cover by three sets (X3C), where one is given a set X (whose size is a multiple of 3) and a collection C of three-element subsets of X, and asked if some subcollection of C covers X exactly (each element of X is in one, and only one, member of C).

Theorem 9.18

3DM is **NP**-complete.

1. 3DM is in **NP**. Left to the reader (see the Exercises).
2. 3SAT $\leq_P$ 3DM.

Beginning with an instance of 3SAT with variables $x_1 \ldots x_n$ and clauses $c_1 \ldots c_m$, we need to construct three sets X, Y, Z as well as P, the set of triples.

We start with the set X of 2nm elements $x_{i,j}$ and $\neg x_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$. Essentially, choosing $x_{i,j}$ will correspond to "x_i is true in clause j" and $\neg x_{i,j}$ will mean "x_i is false in clause j." Of course, now something must be done to ensure that:

a. Each x_i is either true or false.
b. If true in one clause, it is true in all other clauses (and likewise for false).

We begin Y and Z by putting m elements f_i, $1 \leq i \leq m$ into Y and g_j, $1 \leq j \leq m$ into Z. We begin the set of triples P by adding "satisfying" triples. The number of these triples will vary depending on the structure of the clauses. Each clause has length 3, but may contain a repeated literal (so will contain between one and three distinct literals). Thus, there will be between m and 3m such triples.

$(x_{i,j}, f_j, g_j)$ for $1 \leq j \leq m$ if the variable x_i occurs in clause j.
$(\neg x_{i,j}, f_j, g_j)$ for $1 \leq j \leq m$ if the variable $\neg x_i$ occurs in clause j.

The f and g will appear in no other triples, so to put them into a matching for each i, either $(x_{i,j}, f_j, g_j)$ or $(\neg x_{i,j}, f_j, g_j)$ must be chosen, as shown in Figure 9.17.

For the Boolean expression

$$(x_1 \vee x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

there are three variables in three clauses, so n = m = 3. X has 2*3*3 = 24 members $x_{1,1}$, $\neg x_{1,1}$, $x_{1,2}$, $\neg x_{1,2}$, …, $x_{3,3}$, $\neg x_{3,3}$, where $x_{i,j}$ can be read as "variable x_i in clause j."

Y so far has 3 elements $f_1 \ldots f_3$.
Z so far has 3 elements $g_1 \ldots g_3$.

P so far has eight triples ($3 \le 8 \le 9$) as indicated by the circles.

$(x_{1,1}, f_1, g_1)$, $(x_{3,1}, f_1, g_1)$ ← need one to match f_1, g_1.
$(x_{1,2}, f_2, g_2)$, $(x_{2,2}, f_2, g_2)$, $(\neg x_{3,2}, f_2, g_2)$ ← need one to match f_2, g_2.
$(\neg x_{1,3}, f_3, g_3)$, $(x_{2,3}, f_3, g_3)$, $(\neg x_{3,3}, f_3, g_3)$ ← need one to match f_3, g_3.

We will show that given a satisfying assignment of truth values, one can get a three-dimensional matching (and vice versa).

If we have a satisfying assignment, there will be one or more true literals in each clause. With x_1 true, x_2 true, and x_3 false, these triples are bold and italicized below. Arbitrarily, choose one italicized triple from each clause. The chosen triple is shown underlined.

$\underline{\boldsymbol{(x_{1,1}, f_1, g_1)}}$, $(x_{3,1}, f_1, g_1)$ ← Choose $(x_{1,1}, f_1, g_1)$

$\underline{\boldsymbol{(x_{1,2}, f_2, g_2)}}$, $\boldsymbol{(x_{2,2}, f_2, g_2)}$, $\boldsymbol{(x_{3,2}, f_2, g_2)}$ ← Choose $(x_{1,2}, f_2, g_2)$

$(\neg x_{1,3}, f_3, g_3)$, $\boldsymbol{(x_{2,3}, f_3, g_3)}$, $\underline{\boldsymbol{(\neg x_{3,3}, f_3, g_3)}}$ ← Choose $(\neg x_{3,3}, f_3, g_3)$

The clauses and choices are shown in Figure 9.17.

Choosing one will amount to letting x_i be true in clause j, but there is nothing (so far) to keep x_i from being true in one clause and false in another. For example, so far it is legal to have both $(x_{1,1}, f_1, g_1)$ and $(\neg x_{1,3}, f_3, g_3)$ in a matching, meaning x_1 is true in clause 1 and false in clause 3. So, we add triples to prevent this and to ensure consistency across all clauses.

As shown in Figure 9.18, if we add another nm elements into each of Y and Z as follows: Y gets $a_{i,j}$ and Z gets $b_{i,j}$ $1 \le i \le n$, $1 \le j \le m$, and we start P by putting these 2nm "consistency" triples into P:

$(x_{1,1}, a_{1,1}, b_{1,1})$ $(x_{1,2}, a_{1,2}, b_{1,2})$ … $(x_{1,m}, a_{1,m}, b_{1,m})$ (m triples)
$(\neg x_{1,1}, a_{1,2}, b_{1,1})$ $(\neg x_{1,2}, a_{1,3}, b_{1,2})$ … $(\neg x_{1,m-1}, a_{1,m}, b_{1,m-1})$ (m – 1 triples)

and one more triple,

$(\neg x_{1,m}, a_{1,1}, b_{1,m})$ (1 triple)

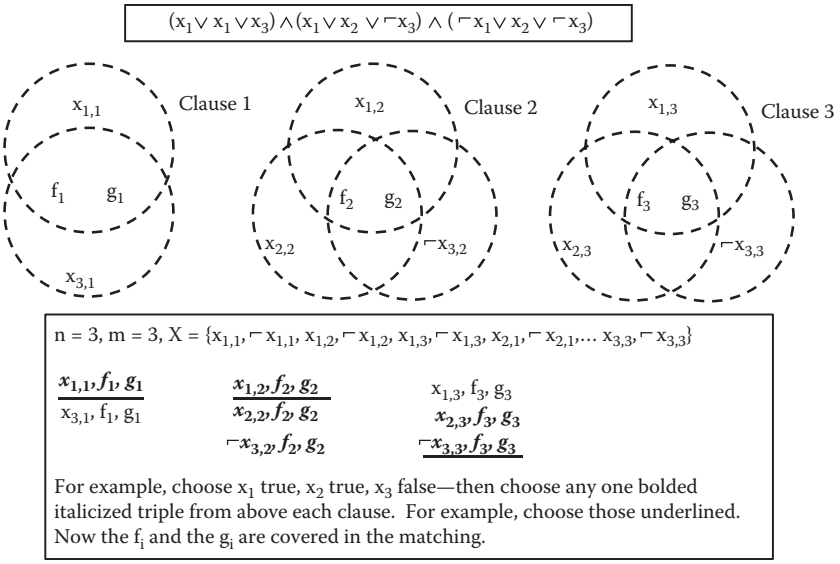


FIGURE 9.17 Triples for variable x_1 appearing in clauses 1, 2, 3.

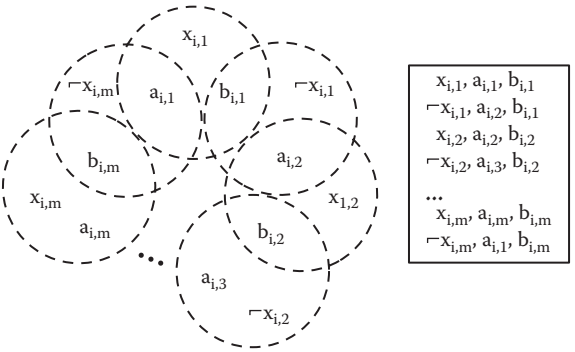


FIGURE 9.18 Building the $a_{i,j}$ and $b_{i,j}$ triples.

and so on, up to

$$(x_{n,1}, a_{n,1}, b_{n,1})\ (x_{n,2}, a_{n,2}, b_{n,2}) \ldots (x_{n,m}, a_{n,m}, b_{n,m})$$
$$(\neg x_{n,1}, a_{n,2}, b_{n,1})\ (\neg x_{n,2}, a_{n,3}, b_{n,2}) \ldots (\neg x_{n,m-1}, a_{n,m}, b_{n,m-1})$$
$$(\neg x_{n,m}, a_{n,1}, b_{n,m})$$

The $a_{i,j}$ and $b_{i,j}$ appear nowhere else—each appears in exactly two triples, so one of the two must be chosen in a matching. Each goes with either $x_{i,j}$ or $\neg x_{i,j}$. If $x_{i,1}$ is chosen to go with $a_{i,1}$ and $b_{i,1}$, there is no option but to

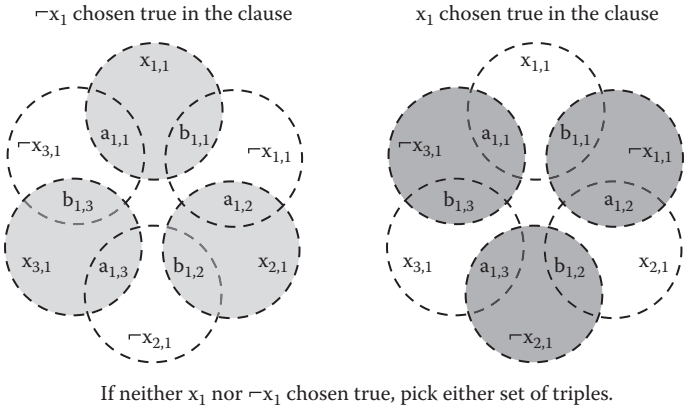


FIGURE 9.19 Choosing a subset of triples to cover $a_{1,1}$, $b_{1,1}$, $a_{1,2}$, $b_{1,2}$, $a_{1,3}$, $b_{1,3}$.

choose it with all the other $a_{i,j}$. In other words, half the triples (circles in the figure) must be chosen, and the circles cannot overlap—so the choice will include alternating circles/triples. This is shown for $|C| = 3$ clauses in Figure 9.19.

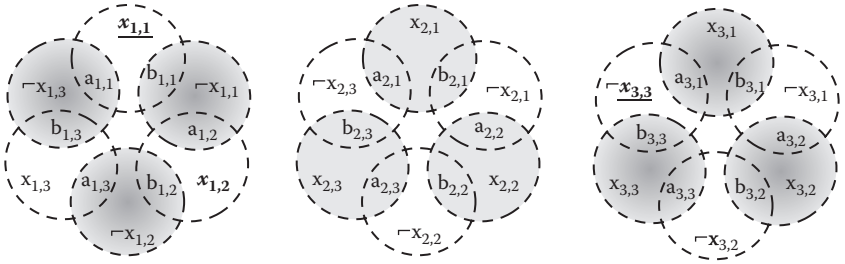
For our example $(x_1 \vee x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$, we chose x_1 true and x_3 false, and have already constructed a partial 3DM with $x_{1,1}$ and $\neg x_{3,3}$ matched. In Figure 9.20, the first and third groups have no choice—having already covered $x_{1,1}$ and $\neg x_{3,3}$, they cannot appear in a matching. In the second group of circles, either set of nonoverlapping three will work. Figure 9.20 demonstrates this.

The sets and triples above are the important ones, but no matching will be possible yet, as X has 2nm elements, but Y and Z each have nm + m elements. We simply add another nm – m elements to each set (g_r to Y, k_r to Z, $1 \le r \le nm - m$). These triples are listed below. They have no purpose except to bring Y and Z up to the size of X, and do nothing to enforce any restriction.

$(x_{i,j}, g_r, k_r)$ (nm)(nm – m) triples
$(\neg x_{i,j}, g_r, k_r)$ (nm)(nm – m) triples

Thus, we have three sets, each of size 2 nm, and we have a set of triples with $t + 2\,mn + 2\,nm(nm - m)$ elements, where $t = |f_i, g_i \text{ triples}|$, $m \le t \le 3m$.

We claim that the clauses are satisfiable if and only if the instance of 3DM has a matching. One completed matching for $(x_1 \vee x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ is shown in Figure 9.21.



Already chosen (underlined $x_{i,j}$ above):

$x_{1,1}, f_1, g_1$ $\quad$ $x_{1,2}, f_2, g_2$ $\quad$ $\neg x_{3,3}, f_3, g_3$

Cannot choose them, so choose "opposite" triples, shaded above:
(in center diagram, no constraint on which three are chosen if no overlap)

$x_{1,1}, a_{1,1}, b_{1,1}$	$x_{2,1}, a_{2,1}, b_{2,1}$	$x_{3,1}, a_{3,1}, b_{3,1}$
$\neg x_{1,1}, a_{1,2}, b_{1,1}$	$\neg x_{2,1}, a_{2,2}, b_{2,1}$	$\neg x_{3,1}, a_{3,2}, b_{3,1}$
$x_{1,2}, a_{1,2}, b_{1,2}$	$x_{2,2}, a_{2,2}, b_{2,2}$	$x_{3,2}, a_{3,2}, b_{3,2}$
$\neg x_{1,2}, a_{1,3}, b_{1,2}$	$\neg x_{2,2}, a_{2,3}, b_{2,2}$	$\neg x_{3,2}, a_{3,3}, b_{3,2}$
$x_{1,3}, a_{1,3}, b_{1,3}$	$x_{2,3}, a_{2,3}, b_{2,3}$	$x_{3,3}, a_{3,3}, b_{3,3}$
$\neg x_{1,3}, a_{1,1}, b_{1,3}$	$\neg x_{2,3}, a_{2,1}, b_{2,3}$	$\neg x_{3,3}, a_{3,1}, b_{3,3}$

FIGURE 9.20 Legal choices to cover the $a_{i,j}$ and $b_{i,j}$.

n=3, m=3,

$X = \{x_{1,1}, \neg x_{1,1}, x_{1,2}, \neg x_{1,2}, x_{1,3}, \neg x_{1,3}, x_{2,1}, \neg x_{2,1}, \ldots x_{3,3}, \neg x_{3,3}\}$

$Y = \{f_1, f_2, f_3, a_{1,1}, \ldots a_{3,3}, g_1, \ldots g_6\}$

$Z = \{g_1, ,g_2, g_3, b_{1,1}, \ldots b_{3,3}, k_1, \ldots k_6\}$

//nm-m=6 more elements added to Y and Z so sets equal size

// add 3*3*6 more "garbage" triples $x_{i,j}, g_r, k_r$ $1 \le i \le 3, 1 \le j \le 3, 1 \le r \le 6$

Choose one true variable in each clause (one possible way):

$x_{1,1}, f_1, g_1$ $\quad$ $x_{1,2}, f_2, g_2$ $\quad$ $\neg x_{3,3}, f_3, g_3$

Choose the OPPOSITE triples from the $a_{1,j}\, b_{1,s}$ and $a_{3,j}\, b_{3,s}$ triples. Choose any three compatible $a_{2,j}\, b_{2,s}$ triples

$\neg x_{1,1}, a_{1,2}, b_{1,1}$	$x_{2,1}, a_{2,2}, b_{2,1}$	$x_{3,1}, a_{3,1}, b_{3,1}$
$\neg x_{1,2}, a_{1,3}, b_{1,2}$	$x_{2,2}, a_{2,3}, b_{2,2}$	$x_{3,2}, a_{3,2}, b_{3,2}$
$\neg x_{1,3}, a_{1,1}, b_{1,3}$	$x_{2,3}, a_{2,1}, b_{2,3}$	$x_{3,3}, a_{3,3}, b_{3,3}$

Choose six of the 54 garbage triples as necessary to fill in other $x_{i,j}$

$x_{1,2}, g_1, k_1$	$\neg x_{2,1}, g_3, k_3$	$\neg x_{3,1}, g_5, k_5$
$x_{1,3}, g_2, k_2$	$\neg x_{2,3}, g_4, k_4$	$\neg x_{3,2}, g_6, k_6$

*These 2nm=2*3*3=18 triples are a matching*

FIGURE 9.21 Final complete matching.

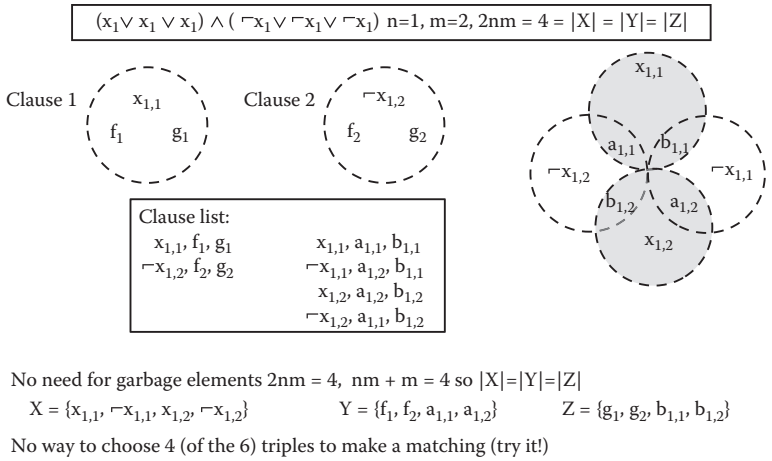


FIGURE 9.22 Example with no possible matching.

1. The triples were constructed in polynomial time—obvious from the size of the set of triples.
2. If the clauses were satisfiable, then we use the truth settings for those clauses to construct the matching as outlined above.
3. If the clauses as constructed have a matching, then there was at least one true variable in each clause. That is, a yes came from a yes.

To see why yes comes from yes, or no goes to no, consider the simplest no instance $(x_1 \vee x_1 \vee x_1) \wedge (x_1 \vee \neg x_1 \vee \neg x_1)$ (n = 1, m = 2). The construction gives three sets of size 4, and six triples, with no way to construct a matching from the six triples. As shown in Figure 9.22, there is no way to construct a matching. ■

9.9 SUBSET SUM

The last major decision problem that we address is Subset Sum. Here, we are given a multiset (which may have repeated elements) of n integers, $S = \{I_1 \ldots I_n\}$ and a target T. The question is, can one choose some subset of the integers $S' = \{I_{i_1} \ldots I_{i_j}\}$, $j \le n$, so that the sum of the integers in S′ is T?

For example, if S = {<u>32</u>, 47, 74, 22, <u>9</u>, <u>50</u>, 15, 18, <u>9</u>} and T = 100, then this is a yes instance since the sum of the underlined integers is 100.

Theorem 9.19

Subset Sum is **NP**-complete. ■

Proof

1. SUBSET-SUM is in **NP**.

   SUBSET-SUM-VERIFY(S, T, c) =

   Check that each element of c corresponds to one distinct element of S.
   Check that the elements of c add to the target T.

The above verifier is quite simple. Note that the addition of two numbers is proportional to the length of the longer integer, so adding the | S′ | elements of the certificate is bounded polynomially.

3-CNF-SAT $\leq_p$ Subset Sum

This is an interesting proof, since it is a conversion between Boolean expressions (variables and clauses) and integers.

An instance of 3-CNF-SAT has k variables $x_1 \ldots x_k$ and m clauses $C_1 \ldots C_m$. Figure 9.23 shows the basic construction for an example Boolean expression. Each of the 2k + 2m rows represents one number in the set S. The maximum length of each integer is k + m; the minimum length is 1.

Quadrants

I	**II**
---	**III**

I: One 1 per row
two 1's per column

II: Any number of 1's per row
one–three 1's per column

III: One 1 per row
two 1's per column

T: 1111..133....3

							Clause 1	Clause 2	Clause 3	⋮	Clause m
x_1	1	0	0	0	...	0	1	0	0	...	
$\neg x_1$	1	0	0	0	...	0	0	0	1	...	
x_2		1	0	0	...	0	0	1	0	...	
$\neg x_2$		1	0	0	...	0	0	0	0	...	
x_3			1	...	...	0	...	...	...	...	
			...	...	...	0					
x_n						1					
$\neg x_n$						1				...	
							1	0	0		
							1	0	0		
								1	0		
								1	0		
									...	...	...
											1
											1
T	1	1	1	1	...	1	3	3	3	...	3

FIGURE 9.23 General scheme for constructing the integers S and target T.

Note that although only 0 and 1 are used as digits when constructing these numbers, this is not binary representation! Each row in Figure 9.23 is one integer, base 10.

As shown in Figure 9.23, we begin by constructing two numbers for each variable $x_1 \ldots x_k$—one for the positive literal x_i and one for its negation $\neg x_i$. As indicated in the upper left quadrant (I) in Figure 9.23, each pair of numbers corresponding to variable x_i has a 1 in column i of the table, with 0's in the following k–i columns to complete this quadrant.

The upper right quadrant (II) is constructed by cross-indexing each variable with each clause. If the positive literal x_i appears in clause C_j, we place a 1 in the corresponding (k + j)th column for x_i; otherwise, the digit is 0. If the negative literal $\neg x_i$ appears in clause C_j, we place a 1 in the corresponding (k + j)th column for $\neg x_i$; otherwise, the value is 0. Thus, each partial row in II may have any number of 1's since x_i or $\neg x_i$ can appear in any number of clauses. Each partial column in II can have one 1 (if the three-clause is $x_i \vee x_i \vee x_i$), two 1's (one repeated literal), or three 1's (three distinct literals).

For example, the literal x_1 occurs in C_1 in the Boolean expression above, so the column for clause 1 in the upper right quadrant is set to 1; x_1 does not appear in C_2 or C_3, and so a 0 is placed in columns 2 and 3. The literal $\neg x_1$ occurs in C_3, so the column for clause 3 in the upper right quadrant is set to 1; $\neg x_1$ does not appear in C_1 or C_2, and so a 0 is placed in the columns for clauses 1 and 2. This is shown in Figure 9.24.

The lower right quadrant (III) adds a pair of numbers 10^{3-j} for each clause C_j. These numbers serve as a reserve, so that we have the option of adding these numbers to reach the target of 3. If one literal is true in a clause, we add both reserves; if two, we add one of the extra numbers; and if all three literals are true, we ignore these numbers completely).

Note that if we do not satisfy C_j, we cannot reach 3 even if we add both of these reserve numbers!

The target number T for the list of numbers is 1^k3^m; that is, the target has k + m digits, the first k of which are 1, the remaining m of which are 3—T has the digit 3 in its final m positions, since our reduction is from 3-CNF-SAT, meaning that each clause has exactly three literals. The upper right quadrant shows what variable assignment "hits" which clause. It is possible to satisfy a clause with all three literals true. For example, choosing $\neg x_1$, x_3, and $\neg x_4$ would hit clause C_3 three times; choosing the numbers corresponding to these literals would add to 3 in the last column. However, it is also possible to satisfy a clause with two literals true (and

$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

					Clause 1	Clause 2	Clause 3
x_1	1	0	0	0	1	0	0
$\neg x_1$	1	0	0	0	0	0	1
x_2		1	0	0	0	1	0
$\neg x_2$		1	0	0	0	0	0
x_3			1	0	1	1	1
$\neg x_3$			1	0	0	1	0
x_4				1	1	0	0
$\neg x_4$				1	0	0	1
					1	0	0
					1	0	0
						1	0
						1	0
							1
							1
	1	1	1	1	3	3	3

FIGURE 9.24 Boolean expression and the associated 14 S integers and T =1111333.

add to 2) or with one literal true (and add to 1). For example, x_1, x_3, and x_4 would hit only literal x_3 in C_3. Thus, one or two of the "dummy" 1's in quadrant III would be needed.

And why does T have 1 in the first i positions? In order to add to 1 in the ith column of the table, we must select one of the literals for variable x_i—choosing either x_i or $\neg x_i$. Thus, an assignment must be made for every variable in the Boolean expression.

Claims to be proved:

1. The component design is a polynomial transformation.
2. A yes instance of 3-CNF-SAT goes to a yes instance of the Subset Sum.
3. A no instance of 3-CNF-SAT goes to a no instance of the Subset Sum—or as usual, that a yes instance of the Subset Sum came from a yes instance of 3-CNF-SAT.

The transformation is polynomial since its size is easily estimated by examining the table:

Number of rows: $2k + 2m$
Number of columns: $k + m$

This is clearly polynomial.

x_1	1	0	0	0	1	0	0
$\neg x_1$	1	0	0	0	0	0	1
x_2		1	0	0	0	1	0
$\neg x_2$		1	0	0	0	0	0
x_3			1	0	1	1	1
$\neg x_3$			1	0	0	1	0
x_4				1	1	0	0
$\neg x_4$				1	0	0	1
					1	0	0
					1	0	0
						1	0
						1	0
							1
							1
	1	1	1	1	3	3	3

x_1	x_2	x_3	x_4
T	F	T	F

For a satisfying assignment, a subset of numbers can be selected that will add up to the target number

$$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$$

FIGURE 9.25 Solution to Subset Sum taken from satisfying assignment.

If the Boolean expression is satisfiable, then each individual clause must also be satisfied. Choosing the number corresponding to the true (x_i) or false ($\neg x_i$) assignment of each variable, and adding up to two possible reserve numbers for each clause, the target T can always be reached. The solution for our example is shown in Figure 9.25.

If the target sum were reached, the columns with a 3 in them could only be reached by choosing at least one number from the top section of the schema—at least one came from the ones from variables in clauses. Setting that variable to be true will give a truth assignment with at least one true variable per clause.

For instance, the crucial piece of finding a solution to the Subset Sum problem is to have a partial sum of $1 \ldots 1X_1 \ldots X_m$ at the bottom of quadrants I and II, where the X_i are digits $1 \leq X_i \leq 3$ (since then the two 1's in quadrant III can be used to hit $1 \ldots 13 \ldots 3$). As long as you can do this, you will have found a way to satisfy the Boolean expression. See Figure 9.26. ■

9.10 SUMMARY AND REPRISE

We have seen a variety of **NP**-complete problems from areas of logic, graph theory, and number theory. There seems to be little way of making blanket categorizations, except that **NP**-complete problems involve finding a solution from an exponential number of possibilities, and for these problems, no good shortcuts have been found.

x_1	1	0	0	0	1	0	0	*
$\neg x_1$	1	0	0	0	0	0	1	
x_2		1	0	0	0	1	0	*
$\neg x_2$		1	0	0	0	0	0	
x_3			1	0	1	1	1	
$\neg x_3$			1	0	0	1	0	*
x_4				1	1	0	0	
$\neg x_4$				1	0	0	1	*
					1	0	0	
					1	0	0	
						1	0	
						1	0	
							1	
							1	
	1	1	1	1	3	3	3	

$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$

x_1	x_2	x_3	x_4
T	T	F	F

If we choose the starred rows, partial sum at this point is 1 1 1 1 1 2 1

FIGURE 9.26 Partial sum that can be completed to be a yes for Subset Sum.

There are other standard **NP**-complete problems that look different superficially and may (or may not) actually be similar to some that have been discussed. Multiprocessor Scheduling is one; it is actually very similar to Subset Sum.

There are many known **NP**-complete problems. Garey and Johnson (1979) list hundreds of problems known to be **NP**-complete at that time. It would be possible to take any—or even all—of the problems listed in that text and assign them as exercises. Some are easy and some are very difficult. The well-educated computer scientist does not need to be able to identify **NP**-complete problems and prove their **NP**-completeness. He or she does need to have some experience with these problems, be able to guess if a problem may be **NP**-complete, and have some idea what that might predict for solutions.

EXERCISES

1. Prove that the following problems are in **P**:

   a. Two-coloring for a graph G

   b. Two-satisfiability for a Boolean expression in conjunctive normal form

   c. Two-dimensional matching between sets X and Y with pairs $W \subseteq X \times Y$

2. For each pair of problems below, decide if the second is a correct generalization of the first.

a. Graph G: G has a three-coloring.

   Graph G: G has a four-coloring.

b. K-clique: Given a graph G and a number K, does G contain a k-clique?

   Subgraph isomorphism: Given graphs G and G′, does G contain a subgraph isomorphic to G′?

c. Directed Hamiltonian circuit in a directed graph G.

   Directed Hamiltonian path in a directed graph G.

d. 3-SAT.

   4-SAT.

e. Partition (integer): Given a finite set of integers $S = \{I_1 \ldots I_n\}$, can S be divided into two subsets (A and S-A) so that the sum of the integers in A is equal to the sum of the integers in S-A?

   Knapsack: Given a set of items $S = \{u_1 \ldots u_n\}$, where each item u has an integer size s_i and an integer value v_i, and two positive integers B (the capacity of knapsack) and K (the desired quantity to pack), is there a subset S′ of S such that the sum of the sizes of the items in S′ is ≤B and the sum of the values in S′ is ≥K?

f. 3-CNF-SAT: Boolean expression in CNF satisifiable, three terms per clause.

   3-DISTINCT-CNF_SAT: Boolean expression in CNF satisfiable with three distinct terms per clause.

3. The k-SAT decision problems are variations of 3-SAT, for each $k \geq 1$:

   Instance: Boolean expression in CNF, each clause contains exactly k distinct literals.

   Question: Is the expression satisfiable?

   Problem: Prove that 4-SAT is **NP**-complete.

4. MONOTONE_SAT, or PURE_SAT, is a restriction of SAT; each clause in the expression must contain either all negated literals or all unnegated literals. Show that MONOTONE_SAT is **NP**-complete.

5. Suppose that one has N files, of size $s_1, s_2, \ldots, s_N$ (in megabytes), to store on disk, and suppose that there are two K-megabyte disks available for storage.

   Show that the problem of deciding whether M files, for $1 \leq M \leq N$, can be stored on the two disks is **NP**-hard.

6. A degree-constrained spanning tree of degree K in graph G is a spanning tree in which no vertex has degree larger than K. The "degree constrained spanning tree problem" is:

   Instance: Graph G (V, E), positive integer K <= |V|.

   Question: Is there a spanning tree for G in which no vertex has degree larger than K?

   a. Give (clearly and concisely) a yes instance.

   b. Give a no instance.

   c. Prove: This problem is **NP**-complete.

7. Consider the problem TRUE-SAT = {Boolean expressions E in conjunctive normal form that satisfy both:

   i. are true when all variables are set true, AND ALSO

   ii. have some other truth assignment that makes E true}

   Instance: Set of clauses $C_1, C_2, \ldots, C_k$ over the Boolean variables $u_1, u_2, \ldots, u_m$.

   Questions:

   i. Are all C_i true when $u_1, u_2, \ldots, u_m$ are set to true?

   ii. Is there another satisfying truth assignment (not all variables set to true)?

   Problems:

   a. Show that TRUE-SAT is in **NP**.

   b. State clearly but briefly what one must show to prove that there is a polynomial time reduction (polynomial transformation) from problem X to problem Y.

   c. The following is proposed as a reduction from SAT: Given a set of k clauses over variables $u_1, u_2, \ldots, u_m$, construct k new clauses

by adding a new variable z to each clause. Prove or disprove: This shows SAT $\leq_p$ TRUE-SAT.

8. Consider the following decision problem.

   Instance: A polynomial $p(x_1, \ldots, x_n)$ of degree k in n variables having integer coefficients.

   Question: Do there exist integers $a_1, \ldots, a_n$ such that $p(a_1, \ldots, a_n) = 0$?

   a. Why does the following argument *not* prove that this problem is in **NP**?

      "Guess values for $a_1, \ldots, a_n$ and carry out the evaluation. Since the number of multiplications and additions required will be bounded by the size of the polynomial, this can be carried out in polynomial time."

   b. This problem is undecidable. What does that say about it being **NP**-complete?

9. Prove that the version of satisfiability in which every variable x_i and $-x_i$ appears in no more than two clauses is **NP**-complete. (Hint: Reduction from standard CNF-SAT.)

10. Prove that the k-link path problem is **NP**-complete.

    Instance: Graph G = (V, E), integer k > 0, vertices v and w.

    Question: Does G have a simple path from v to w with exactly k edges?

11. Prove that HALF_CLIQUE is **NP**-complete.

    Instance: Graph G = (V, E), |V| is even.

    Question: Does G contain a complete subgraph of size |V|/2?

12. Prove that Dir-HC (Hamiltonian circuit in a directed graph) is **NP**-complete.

13. Prove that ≠–3_SAT is **NP**-complete.

    Instance: A Boolean expression in CNF with three literals per clause.

    Question: Does there exist an assignment of truth values that makes the expression true, but in which no clause contains three literals that are all true? (Each clause contains two literals with opposite truth values.)

14. Consider 3-DISTINCT-CNF-SAT.

    Instance: A Boolean expression in conjunctive normal form with three distinct literals per clause.

    Question: Does there exist an assignment of truth values that makes the expression true?

    a. Give yes and no instances of 3-DISTINCT-CNF-SAT.

    b. Prove 3-DISTINCT-CNF-SAT is **NP**-complete.

15. Find a polynomial reduction from Bipartite_Graph to 2-Graph_Color.

    Definitions to Use:

    Bipartite Graph: The vertices of graph G = (V, E) can be separated into two subsets V_1 and V_2 so that every edge in E joins a vertex in V_1 to a vertex in V_2.

    Two-Graph Color: The vertices of G = (V, E) can be colored with two colors so that no two vertices connected by an edge are assigned the same color.

16.

    a. Map the no instance $(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$ of 3-CNF-SAT to an instance of VERTEX COVER. Show that it is a no instance of VERTEX COVER.

    b. Map the no instance $(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$ of 3-CNF-SAT to an instance of Subset Sum. Show that it is a no instance of Subset Sum.

17. Finish the proofs in the text.

    a. Prove that the transformation given from VC to HC is polynomial.

    b. Prove that in the transformation given, VC to HC, yes goes to yes (or no comes from no).

    c. Prove that 3DM is in the class **NP**.

18. Prove X3C (exact cover by 3 sets, see p. 209) is **NP**-complete.

CHAPTER 10

Other Interesting Questions and Classes

10.1 INTRODUCTION

In this chapter, we will look at interesting questions and classes that come out of the ideas in the previous chapters. Are there other complexity classes? The answer is yes, that there are entire hierarchies of classes, although most are not as naturally defined as **P** and **NP**. We will consider some of the more important of these classes.

How do these relate to what has already been discussed? In many cases, we will be able to show some interesting relationships. But are there any subset relations known to be proper subsets? If so, which? What natural questions are still unanswered?

Are there distinctions between or among the members of the class of **NP**-complete problems? Are there any subclasses of **NP** except the classes **P** and **NP**-complete? Are all **NP**-complete problems of the same difficulty? In one sense, yes, since of course any **NP**-complete problem is polynomially reducible to any other **NP**-complete problem. However, some might be considered to be harder than others. A few owe their difficulty to aspects of the problem that disappear when certain limits are set on the input.

10.2 NUMBER PROBLEMS

Consider the following simple algorithm to decide whether an integer p is prime:

```
is_prime = ! (p mod 2 == 0);
div = 3;
while (div²<= p) and is_prime {
  if (p mod div == 0) is_prime = false;
  div += 2;}
```

This algorithm is bounded by $1/2\sqrt{p}$ (and will take that many iterations when p is prime). Therefore, it would be tempting to say that the algorithm is better than polynomial, since it runs in $O(p^{1/2})$ iterations. But there is a fallacy in this thinking. Yes, it runs in $p^{1/2}$, but this is *not* polynomial in the length of the input, as is required in the definition of the class **P**. The length of the input is the number of digits in p, or, in base 10, the ceiling of log p $\lceil \log_{10} p \rceil$, that is,

$$p^{1/2} = 10^{.5 \log p} = 10^{constant * sizeofinput}$$

Thus, it is really exponential in the size of the input. This distinction—an algorithm that is polynomial in some *part* of the input, but not the *size* or length of the input—appears in several **NP**-complete problems. A polynomial algorithm for PRIME has been found (Agrawal et al., 2004), but it is much more complex than the above algorithm.

Another interesting problem is Subset Sum, which we have shown to be **NP**-complete. Recall that this problem consists of a multiset (a set with possibly repeated elements) of n integers, $S = \{I_1, \ldots, I_n\}$ and a target T. The question is, can one choose some subset of the integers $S' = \{I_{k1}, \ldots, I_{kj}\}$ so that the sum of the integers in S′ is T?

This problem has an interesting dynamic programming algorithm. Dynamic programming algorithms are those that are based on saving solutions to parts of the problem and building on these, combining them into a whole solution.

One dynamic programming algorithm for Subset Sum is based on the construction of an $n \times (T + 1)$ table. The rows are labeled by the $I_1, \ldots, I_n$, the columns by the integers from 0 to T. Each table entry table[i, j] will be TRUE if the jth number can be made from a sum of the first i of the $I_1, \ldots, I_n$. All entries are initialized to FALSE.

For example, suppose than n = 6, T = 21, and $I_1 = 2$, $I_2 = 3$, $I_3 = 6$, $I_4 = 3$, $I_5 = 7$, $I_6 = 5$ (see Figure 10.1).

The algorithm fills the table row by row. Entries in the first row are set to true for 0 and for I_1 (so here, the two true entries are beneath the 0 and the

Columns = integers from 0 to T

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2	TRUE		TRUE																			
3	TRUE		TRUE	TRUE		TRUE																
6	TRUE		TRUE	TRUE		TRUE	TRUE		TRUE	TRUE		TRUE										
3	TRUE		TRUE	TRUE		TRUE	TRUE		TRUE	TRUE		TRUE	TRUE		TRUE							
7	TRUE		TRUE	TRUE		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE		TRUE
5	TRUE		TRUE	TRUE		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Rows = integers $I_1=2$, $I_2=3$, $I_3=6$, $I_4=3$, $I_5=7$, $I_6=5$

FIGURE 10.1 Successful targets with {2, 3, 6, 3, 7, 5} table.

2). To fill in row i, table[i,j] is true if either (1) the entry above it is true or (2) the entry in the row above that is I_i to the left is true; that is, table[i – 1, j – I_i] = true. By hand, the easiest way to proceed is to (1) copy all T's from the previous row down to the current row, and (2) add I_i to each entry that has a true in the row above, and if not already true in this row, make it so.

To demonstrate, consider filling in the third row (labeled 6 since $I_3 = 6$). Copy the four true's from the previous row (under 0, 2, 3, 5). Then 6 = 0 + 6, 8 = 2 + 6, 9 = 3 + 6, and 11 = 5 + 6 also become true.

To fill the entire table, there are only n·(T + 1) entries, and each requires only one addition, so the algorithm is polynomial in nT.

The input consists of the n numbers $I_1 \ldots I_n$ and the one number T. At first glance, the size of the input is (n + 1) and the algorithm is polynomial. However, the reader should be suspicious, since this problem is **NP**-complete.

What is going on? The answer is that the size of the input depends on the *size* of the numbers $I_1 \ldots I_n$, T. Each number I_i requires about $\log_{10}(I_i)$ to represent it in decimal, T requires about $\log_{10}$ T. We can assume that T is larger than any of the $I_1 \ldots I_n$. If it is not, any of the I_j that are larger than T can be discarded as they can play no part in the Subset Sum. Thus, a good bound for the size of the input is (n $\log_{10}$ T), and nT is *not* polynomial in this quantity.

We now try to capture what is going on in the easy algorithm for PRIME and the dynamic programming algorithm for Subset Sum. These algorithms are only very slow if the size of the integers in the input is large. If all input numbers were bounded to fit in a long-length integer on a standard computer, the problem would be much easier. The idea is to capture the difficulty arising because there are unboundedly large numbers in the problem, and it is the *size* of these integers that makes the problem difficult.

Definition 10.1

A decision problem is a **number problem** if there is no polynomial bound on the largest number that is part of the problem input, in terms of the length of the encoding of the input. ■

Let Max(input) be the size of the largest number in an instance I of a problem A (if A has no numbers, Max(input) = 0). The size of the input I

is, as usual, $|I|$. If there is no polynomial $p(n)$ such that $\text{Max(input)} \leq p(|I|)$, then A is a number problem.

In the above example, we can assume that the target $T = \text{Max(input)}$; and no polynomial p exists so that $T \leq p(n \log_{10} T)$, where $n \log_{10} T$ is a measure of the size of the input.

Note that problems such as CNF-SAT and HC have no numbers at all. They are not number problems. A problem such as vertex cover, where the size of the cover is limited by the number of vertices, is also not a number problem. Wait, you may say, there *are* numbers in SAT and in any graph problem—we have numbered the Boolean variables and we have numbered the vertices in the graph. Well yes, but these numbers are just names, and not really numbers. And, the numbers are small compared to the size of the problem—the number of Boolean variables or the integer names of the vertices in the graph are not what makes the problem large.

PRIME is a number problem. It has only one number in its input, and that number is not polynomial in its own log, which is the length of the input.

Definition 10.2

An algorithm is **pseudopolynomial** if it is polynomial in the two variables (1) the length of the input and (2) the size of the largest number in the input. ■

For problems with no numbers in the input, we take condition 2 to be 0 [Max(input) = 0], so an algorithm for a problem with no numbers is polynomial if and only if it is pseudopolynomial.

Since filling in the table for Subset Sum is polynomial in n and T, and we can assume that T is the largest number in the input, this is a pseudopolynomial algorithm. The standard simple algorithm for PRIME is also pseudopolynomial.

Definition 10.3

An **NP**-complete problem is **strongly NP-complete** if it is (1) **NP**-complete and (2) remains **NP**-complete even if it is restricted so that the largest number appearing in the problem is bounded by a polynomial in the length of the input. ■

Subset Sum is (almost certainly) not strongly **NP**-complete, since if T is bounded by a polynomial in the length of the input, T ≤ p (length of the input), then we have an algorithm bounded by nT, and nT is polynomial in the length of the input. It can only remain **NP**-complete with bounded T if **P** = **NP**.

For the **NP**-complete problems that do not have numbers (e.g., SAT), there is no difference between polynomial and pseudo-polynomial. They cannot have polynomial or pseudo-polynomial algorithms unless **P** = **NP**.

There are other problems that have numbers, but the numbers are small numbers compared to the size of the input. For instance, VC has a number K, the size of the vertex cover. K must be smaller than |V|. Any pseudo-polynomial algorithm in K would be a polynomial in |V|. Such problems cannot have pseudo-polynomial algorithms unless **P** = **NP**.

The decision version of the Traveling Salesman Problem (TSP) is a number problem, since the weights on the edges can be arbitrarily large. However, even if the weights are bounded by a small integer (even 1 or 2), it can be shown to remain **NP**-complete (see the Exercises). Thus, it is an example of a number problem that is strongly **NP**-complete.

Multiprocessor Scheduling (defined in the Exercises) is another number problem, and is also strongly **NP**-complete.

10.3 COMPLEMENT CLASSES

A natural operation on sets is taking the complement; a natural operation on decision problems is to consider the no instances. For example, a yes instance of CNF-SAT is a Boolean expression in conjunctive normal form that is true for some assignment of true and false to its variables. A no instance is a Boolean expression, again in conjunctive normal form, that is *never* true for any assignment of truth values to its variables. In other words, it is an expression whose truth table has a final column that is all FALSE.

Definition 10.4

Let X be any class of decision problems. Then co-X = {A | A is a decision problem, and $A^c \in X$}. ■

Fact

$$\mathbf{P} = \text{co-}\mathbf{P}$$

This is true since a polynomial solver for a decision problem also solves the complement of that problem. ■

What does a problem in co-**NP** look like? The easiest way to construct an example is simply to take the complement of a problem in **NP**. For example, take $\overline{\text{CNF-SAT}}$ = {Boolean expressions with no assignment of truth values to make the expression true}. There is a straightforward exponential algorithm to show that an expression is in: $\overline{\text{CNF-SAT}}$—construct the truth table and show that all rows evaluate to false. There is no obvious way to make a shorter certificate. Unlike showing that one row is true, there seems to be no good short way to show that all rows are false without actually constructing all the rows.

Thus, by definition, $\overline{\text{CNF-SAT}}$ is in co-**NP**. But it seems not to be in **NP**. Notice, though, that proving a problem is not in **NP** is nontrivial. One would have to prove that no polynomial certificate exists, and nonexistence proofs are often difficult (usually assuming existence and getting a contradiction).

Or, consider the complement of the Hamiltonian circuit, $\overline{\text{HC}}$ = {G | there is no way to make a Hamiltonian circuit through G}. One way to check that a graph is in $\overline{\text{HC}}$ would be to try all possible circuits; no easier way is readily apparent. $\overline{\text{HC}}$ is also in co-**NP**, but almost certainly is not in **NP**.

10.4 OPEN QUESTIONS

We now have the classes **P**, co-**P**, **NP**, co-**NP**, **NP**-complete, and co-**NP**-complete. What is known? Certainly **P** = co-**P**, and of course, **P** $\subseteq$ **NP**, **NP**-complete $\subseteq$ **NP**, and co-**NP**-Complete $\subseteq$ co-**NP.** What questions are open? We start with the most famous of the open questions:

1. Does **P** = **NP**?

   We have seen this question before, but it is so important that it should be restated. No polynomial algorithm has been found for any **NP**-complete problem, but neither has it been shown that such algorithms do not exist. If a polynomial algorithm for any one of the **NP**-complete problems were to be found, then **P** = **NP** (and much of this text, and many others, would be uninteresting).

   As noted before, a nondeterministic Turing machine (TM) can be transformed to an equivalent deterministic TM, but t(n) steps

become $O(2^{t(n)})$ by the proof given. If some means of doing the transformation could be found that did not introduce an exponent, we would have **P** = **NP**.

2. Is **NP** = co-**NP**?

   This is another open question. Notice that if **NP** = **P**, then **NP** = co-**NP**. However, the reverse is not true. It is possible that **NP** = co-**NP** but still **NP** ≠ **P**.

   Can any problem in co-**NP** also be **NP**-complete? It is possible that problems such as $\overline{\text{SAT}}$ might also be **NP**-complete (although it is not clear that such problems are even in **NP**). However, if there are any problems in co-**NP** ∩ **NP**-complete, then **NP** = co-**NP**.

Theorem 10.5

If any **NP**-complete problem is also in co-**NP**, then **NP** = co-**NP.**

Proof

Assume that some **NP**-complete problem Z is also in co-**NP**. This means (1) for any X in **NP**, $X \leq_P Z$, and (2) $\overline{Z}$ is also in **NP**.

First, to show **NP** ⊆ co-**NP**, assume that X is in **NP**. But, $X \leq_P Z$ so $\overline{X} \leq_P \overline{Z}$ (why?). Now $\overline{X}$ is polynomially reducible to an element of **NP**, so $\overline{X}$ is also in **NP**. That means, of course, X is in co-**NP**.

One must also show that co-**NP** ⊆ **NP**; the proof is very similar and is left as an exercise. ■

10.5 ARE THERE ANY PROBLEMS IN **NP-P** BUT NOT **NP**-COMPLETE?

Essentially all the decision problems that we have examined so far have been in either **P** or **NP**-complete. It would seem that there should be problems that are harder than those in **P** but not as hard as those in **NP**-complete. We might call such problems **NP**-intermediate. The next theorem says that as long as **P** ≠ **NP**, such problems must exist. However, it (so far) is not possible to describe a good example problem in this class.

Theorem 10.6

If **P** ≠ **NP**, then there exist languages in **NP** that are not **NP**-complete but also not in **P**.

Proof

This is a diagonalization argument (Ladner, 1975), but a complicated one. In previous diagonalizations, there has only been one diagonal. In this case, must diagonalize against both the members of **P** and the members of **NP**-complete. This is done by diagonalizing with the polynomial reductions from SAT; every member of **NP**-complete has one of these. Thus, we enumerate all the polynomially bounded Turing machines—and construct a language not decided by one of these, and so not in **P**. Then we also enumerate all the polynomial reductions from SAT—and construct a language not reduced by one of these, and so not in **NP**-complete.

Let M_{SAT} be a TM that decides SAT. An easy exponential algorithm exists to construct and by checking the whole truth table; it is not known if a polynomial time algorithm exists.

All (the countably many) Turing machines can be enumerated as M_1, M_2, …, M_i, ….

To enumerate the polynomially bounded deciding Turing machines (TMs): M_{P1}, M_{P2}, …, M_{Pi}, …, do the following:

Enumerate the TMs: M_1, M_2, …, M_i, ….
Enumerate the polynomials: n, n^2, …, n^k, ….
Enumerate the pairs (M_i, n^j). We can make this pair into a polynomially bounded TM (by running M_i until time = n^j; if it has not yet accepted, stop and reject).

This is now an enumeration of the polynomial-bounded TMs since (1) each is a polynomially bounded TM, and (2) every polynomially bounded TM will appear in the enumeration (why?).

Similarly, enumerate all the polynomial reductions: R_1, R_2, …, R_i, …, as follows:

First, enumerate all the computing Turing machines M_{R1}, M_{R2}, …, M_{Ri}, ….

Enumerate the polynomials: $n, n^2, \ldots, n^k, \ldots$.

Enumerate the pairs (M_{Ri}, n^j). Each pair describes a polynomially bounded reduction. We can run M_{Ri} for time = n^j; if the computation has finished, return the value; else return 0. This now enumerates all polynomial TM reductions.

Of course, we can easily enumerate the strings over Σ^* in lexicographic order, $z_0 = \varepsilon, z_1, z_2, z_i, \ldots$.

We will need one more function f: $\mathbf{N} \rightarrow \mathbf{N}$ and the Turing machine M_f that computes it; it is defined below.

Using M_{SAT} and M_f, we define M, a TM that takes a Boolean expression in conjunctive normal form, as follows. Its language L(M) will be (1) not in **P**, (2) in **NP**, but (3) not **NP**-complete.

For input w, run M_{SAT} on w:

If M_{SAT} rejects w, so does M (so $L(M) \subseteq SAT$).

If M_{SAT} accepts w, compute M_f on $|w|$ to compute $f(|w|)$.

If $f(|w|)$ is an even integer, accept; else reject.

Thus $L(M) = \{w \mid w \text{ is in SAT and } f(|w|) \text{ is even}\}$ (where f and M_f are defined below).

The easy part (once f is defined) is that L(M) is in **NP**—all that is required is a certificate that the string is in SAT, and a computation that $f(|w|)$ is even. This last part will be true since $f(|w|)$ will only take $O(|w|)$ steps to compute.

Now define f, a nondecreasing, but slow-growing function on the positive integers to the integers, and TM M_f that computes f. Begin by setting $f(0) = f(1) = 2$.

Input to M_f: n, encoded as 1^n (as a string of n 1's):

Stage 1: Repeat until the blank cell at the end of 1^n is reached.

Compute the first step of f(0) (easy, f(0) = 2) and move right one cell.

If not finished with f(0), compute its next step else compute first step of f(1) (again easy, f(1) = 2) and move right one cell.

…

If not finished with f(i) compute its next step, else compute first step of f(i + 1); move right.

Since each step moves right, there are only n iterations.

Stage 2: Look at the last (and largest) value of f fully computed in stage 1, f(i) = k. This number k is an integer, so it is either an even or an odd number. The value of f will be either k or k + 1; this stage will decide:

If k is even, k = 2i. Find machine M_{Pi} in the enumeration. Now look at the enumerated z_i. As in stage 1 above, repeat for n steps of $M_{Pi}(z_i)$.

If not done with z_i:

Compute the next step of $M_{SAT}(z_i)$—result 1.

Compute the next step of $M_{Pi}(z_i)$—result 2.

Compute the next step of $M_f(|z_i|)$—result 3.

Else move to z_{i+1}.

If at the end of this time, some string z has been found such that

$$M(z) \neq M_{Pi}(z)$$

then return (k+1); else return (k).

When is $M(z) \neq M_{Pi}(z)$, there are several cases:

1. M_{Pi} (z) accepts and $M_{SAT}(z)$ rejects.
2. M_{Pi} (z) accepts and $M_f(|z|)$ is odd.
3. M_{Pi} (z) rejects and $M_{SAT}(z)$ accepts and $M_f(|z|)$ is even.

If k is odd, k = 2i – 1. Exactly the same, except replace $M_{Pi}(z)$ by $M_{Ri}(z)$ and do the computations below, stopping after n steps:

Compute the next step of $M_{Ri}(z_i)$.

Compute the next step of $M_{SAT}(z_i)$.

Compute the next step of $M_{SAT}(M_{Ri}(z_i))$.

Compute the next step of $M_f|M_{Ri}(z_i)|$.

Return (k + 1) if we have found a z such that $M(M_{Ri}(z)) \neq M_{SAT}(z)$. This happens if:

4. $M_{SAT}(z)$ accepts and $M_{SAT}(M_{Ri}(z))$ rejects.
5. $M_{SAT}(z)$ accepts and $M_f\,|M_{Ri}(z_i)|$ is odd.
6. $M_{SAT}(z)$ rejects, $M_{SAT}(M_{Ri}(z_i))$ accepts, and $M_f\,|M_{Ri}(z_i)|$ is even.

Now M_f is (1) well defined, (2) computes a nondecreasing integer function f, and (3) computes f in $O(n)$ time.

Now let L be the language decided by M. We claim that (1) L is in **NP** (easy, done above), (2) L is not in **P**, and (3) L is not **NP**-complete.

L is not in **P**: This is the first diagonalization. If it were in **P**, it would be accepted by one of the polynomial deciding machines M_{Pi} in the enumeration.

$$L = L(M) = L(M_{Pi})$$

Thus, we never find the string z so that $M(z) \neq M_{Pi}(z)$.

Now consider the even number 2i. None of the conditions 1 to 3 can be satisfied and f(n) = 2i from this point on. That means there are only a finite number of strings for which f(|w|) is odd, and L only differs from SAT in a finite number of strings. But then L in **P** would imply that SAT is also in **P**, which contradicts the assumption **P** ≠ **NP**.

L is not **NP**-complete: Suppose it were, then there is a polynomial reduction (which must be computed by M_{Ri} in the enumeration of all polynomial reductions) from CNF-SAT to L, that is, $M(M_{Ri}(z_k)) = M_{SAT}(z_k)$ for all z_k. So, after 2i – 1, none of conditions 4 to 6 are satisfied and f(n) = 2i – 1 for all but a finite number of n. Then f(|w|) is only even for a finite number of strings, L(M) is a finite language, and L(M) is in **P**. But since we assumed L is **NP**-complete, this again contradicts the assumption **P** ≠ **NP**. ■

However, since this is not a proof that constructs a language that is easy to describe, this does not yield a nice useful example. Are there any **NP**-intermediate languages that arise from natural problems? Somewhat surprisingly, no good examples are known. There are few candidate problems including the following.

10.5.1 Linear Programming

Until 1979, when Linear Programming was shown to be in P, it was considered a possible candidate for **NP**-intermediate.

Given m integer vectors ($V_1 \ldots V_m$) of length n, vectors D of length m ($d_1 \ldots d_m$), and C of length n ($c_1 \ldots c_n$), and integer target T, is there a rational vector X, X = ($x_1 \ldots x_n$) such that

$$V_i \cdot X \leq d_i \text{ for all i, } 1 \leq i \leq m$$

$$C \cdot X \geq T$$

The operator is the dot product of two vectors, = $(x_1 \ldots x_n)\cdot(y_1 \ldots y_n) = x_1y_1 + x_2y_2 + \ldots + x_ny_n$.

In **NP**: The certificate is the vector X.

In **P**: The polynomial algorithm is found (Khachian, 1979).

This problem has the interesting extra property that the simplex algorithm has long been used to solve it. The simplex algorithm is not polynomial—its worst case running time is exponential. It could not be used to show membership in **P**. However in practice, its usual running time has been fast.

One lesson here is that exponential algorithms should not be simply discarded as too slow. Even though an algorithm is exponential, it may run quickly enough in practice, on average cases, to be useful.

10.5.2 PRIME

Given a positive integer N, is N prime?

In **NP**: The certificate is certainly not obvious—the certificate for COMPOSITE is any pair of factors. However, there is also a certificate for PRIME.

In **P**: The polynomial algorithm is found (Agrawal et al., 2004), so this cannot be our **NP**-intermediate problem.

10.5.3 Graph Isomorphism

Given two graphs $G_1 = (V, E)$ and $G_2 = (V, E_2)$, is there a way to match the vertices of G_1 to those of G_2 so that there is an edge between the two vertices of G_1 iff there is an edge between the corresponding vertices of G_2? (Note that we assume that the two graphs have the same number of vertices, $v_1, \ldots, v_n$, otherwise, there would be no hope of an isomorphism.)

In **NP**: The certificate is the vertex correspondence between the two graphs.

It is still unknown whether in **P** or **NP**-complete: Open question. It has been proven in **P** for some special cases of graphs.

10.5.4 Other Examples?

Here, there are very few examples of possible problems to be in **NP**, but not **NP**-complete. Almost any problem that has a polynomial certificate has turned out to be in either **P** or **NP**-complete. Again, if **P** = **NP**, there can be no **NP**-intermediate problems.

10.6 PSPACE

In discussing the possible meanings of efficiency for an algorithm, both time efficiency and space efficiency were mentioned. But, in all of the

previous analyses of algorithms, only the number of steps (time) has been counted.

What about space? Space is of course important, and algorithms that require too much space are impractical. How should one count space? What about an algorithm that requires no additional space beyond what the input occupies? This last question is interesting, and sometimes it is interesting to count only the space beyond that occupied by the input.

We will assume that every algorithm uses at least the space occupied by the input. Thus, for s(n) = number of Turing machine tape squares required to solve the worst case input of size n,

$$s(n) = \max(n, \text{tape cells besides the input required by the algorithm})$$

In the relationships between classes, there is one obvious fact: in t(n) time steps, no Turing machine can access more than t(n) cells on the tape.

This easily implies the facts below:

Fact

Any algorithm that runs in time t(n) must also run in space t(n). ■

Fact

$$\mathbf{P} \subseteq \mathbf{PSPACE}$$ ■

Somewhat less obvious, but still easy to prove, is the following:

Theorem 10.7

$$\mathbf{NP} \subseteq \mathbf{PSPACE}$$

Proof

Any problem in **NP** has a certificate that can be checked in polynomial time and so must not take up more than polynomial space. Thus, we can

write a **PSPACE** algorithm by listing the possible certificates, one at a time, reusing the same space each time, and verifying whether or not this is the certificate. Or, to put it briefly: we can try all possible certificates; if one works, report yes; otherwise report no.

There are likely to be an exponential number of possible certificates, but here there is no concern with the running time of the algorithm. It is only necessary to cycle through the possible certificates, one at a time. ■

It is also obvious that any machine can spend limitless time on very limited space. It would be possible to cycle forever between two adjacent tape squares. But, what is less obvious is that if space is limited, time can be limited—there is a time bound so that once this is reached, a Turing machine that has not halted yet will never halt (and thus never accept). For example, a Turing machine that cycles between adjacent tape cells must be going back and forth between two configurations that are the same—something like q_{35}, a $\rightarrow$ q_{349}, a, R and q_{349}, a $\rightarrow$ q_{35}, a, L. Since it is in exactly the same configuration as before, it must be in an infinite loop—and we can detect this loop. If it was here before, in the same state, at the same place, with the same tape contents, and since this is a deterministic Turing machine, well then, it must be headed off doing the same thing it did before, and it will be returning to this configuration again and again and again.

Thus, there is a limit to the amount of space a Turing machine can use, and if we keep track of all possible Turing machine configurations—there may be a very large number, but finite—then if we see any configuration repeated, we must be in an infinite loop. Furthermore, if this possible number of configurations is K and the Turing machine runs for K + 1 steps, then we know (by the pigeonhole principle*) that it must have repeated some configuration and be in an infinite loop. Thus, there is no reason to let this Turing machine run for arbitrarily long time periods. After (K + 1) steps, it has either accepted or it will never accept.

This means that it is not necessary to keep track of all those TM configurations to check for a repeated configuration. After the time bound (K + 1), it must be in a repeated configuration, and must be in an infinite

* The pigeonhole principle is a standard fact of mathematics, and it says that if m objects are placed in n slots (pigeonholes), with m, n finite, and m > n, then at least one slot contains two objects. Think of a mailman placing 10 letters in 9 mailboxes—it is obvious that at least one mailbox must get at least two letters.

loop. Thus, a space-bounded Turing machine can always be converted to an equivalent deciding, always-halting Turing machine.

This can be stated as a theorem:

Theorem 10.8

If a problem is in SPACE s(n) for any function s(n), then the problem is decidable. In particular, for s(n)≥n, it has a Turing machine bounded by time $O(2^{ks(n)})$.

Proof

In previous paragraphs. ■

10.7 REACHABLE CONFIGURATIONS

In a graph, vertex v_i is reachable from vertex v_j if there is a path between the vertices. There is a simple polynomial algorithm to check this (see Chapter 4).

This has a nice application to Turing machines. We consider a graph that has nodes for every possible configuration of the Turing machine operating on a string w using no more than K cells on the tape, with edges between nodes for moves of the transition function δ. There are a lot of these configurations, but only a finite number (see Exercises). There is, of course, one starting configuration (q_0, $w_1w_2 \ldots w_n$). There are many accepting configurations. Any configuration with state q_{accept}—as there is no concern as to what is on the tape—is an accepting configuration.

The question about whether a Turing machine accepts string w can be transformed to the graph question: Is there a path from the starting configuration to any of the accepting configurations?

What is the size of this graph, where every configuration is a node? For a Turing machine with Q states, tape alphabet Γ using S tape cells, the number of configurations is bounded by

$$|G| = |Q| \times |\Gamma|^S \times S$$

Since the Turing machine is one of |Q| possible states, each of the S cells contains one of |Γ| symbols, and the head is on one of the S cells.

Thus, for a space-bounded problem, one can find whether the machine reaches an accept state by using the standard algorithm for reachable on this graph.

10.8 **NP**SPACE = **PSPACE**

There is a surprising relationship between nondeterministic space bounds and deterministic space. Recall that a major recurring theme has been the fact that it is unknown whether **P** = **NP.** However, for space, the answer to the analogous question is known.

We could define **NP**SPACE as the set of problems that can be accepted by a nondeterministic Turing machine in polynomial space; that is, those that have at least one path to q_{accept} that requires no more than a polynomial amount of space (for strings in the language).

Theorem 10.9 (Savitch, 1969)

If a decision problem X can be recognized by a nondeterministic Turing machine in space bounded by s(n), where n is the length of the input, then X can be recognized by an equivalent deterministic Turing machine in space bounded by $s^2(n)$.

Proof

This is an application of reachability. For configurations on the Turing machine (the vertices in the graph of all configurations) let:

REACHABLE (c_i, c_j, k) = true if configuration c_j is reachable from c_i in k steps

Note the following:

REACHABLE (c_i, c_j, 0) = true if configuration c_j equals c_i
REACHABLE (c_i, c_j, 1) = true if $\delta(c_i)$ includes c_j as one of the possibilities

As we have noted, the total number of configurations is finite, and for some K,

$$2^K > \text{total number of configurations}$$

If c_j is reachable from c_i in any number of steps, it can reached in 2^K (or fewer) steps.

$$\text{REACHABLE } (c_i, c_j, k) \text{ for any } k \Leftrightarrow \text{REACHABLE } (c_i, c_j, 2^K)$$

For a Turing machine that uses s(n) cells, the total number of configurations is

$$\begin{aligned} |Q| \times |\Gamma|^{s(n)} \times s(n) &= \text{constant} \times 2^{\log |Q| + \log s(n) + s(n) \log |\Gamma|} \\ &= \text{constant} \times 2^{\text{constant x } s(n)} \end{aligned}$$

That is, $K = K's(n)$.

Now for each of the accepting configurations c_{accept}, we will determine REACHABLE $(c_0, c_{accept}, 2^K)$. This will be true if, for some configuration c_m:

$$\text{REACHABLE } (c_0, c_m, 2^{K-1}) \text{ and REACHABLE } (c_m, c_{accept}, 2^{K-1})$$

Thus, our algorithm for REACHABLE $(c_i, c_j, 2^K)$ will be:

```
REACHABLE (ci, cj, 2^K) =
      for all configurations c
            if REACHABLE (ci, c, 2^(K-1))
            if REACHABLE (c, cj, 2^(K-1))
                  return true
      return false
```

This is a recursive algorithm. There are no more than $K = K's(n)$ calls to REACHABLE active at once. For each call, we store the configuration of the caller and K. Each configuration requires $O(s(n))$ space to store—essentially, the tape (bounded by s(n), with the head position encoded) and the state number, $\leq |Q|$.

Thus, the total space used is $K's(n)s(n) = O(s^2(n))$. ■

Corollary 10.10

NPSPACE = PSPACE

Proof

The square of a polynomial is a polynomial. ■

10.9 A **PSPACE** COMPLETE PROBLEM

What kinds of problems are in **PSPACE**? Of course, any problem in **P** or in **NP** is in **PSPACE**. Any problem in co-**NP** is in **PSPACE**.

But all of these problems are (probably) too easy to catch the full power of what can be computed in polynomial space, **PSPACE**. We should look for one of the hardest problems in this category.

Definition 10.11

A problem is **PSPACE**-complete if:

1. Y is in the class **PSPACE**.
2. For any problem X in **PSPACE**, $X \leq_P Y$. ■

Definition 10.12

A problem Y is **PSPACE**-hard if for any problem X in **PSPACE**, $X \leq_P Y$. ■

The reduction used is still polynomial *time*, though it is applied to problems in polynomial *space*.

To find a good example, it is only necessary to return to Boolean expressions. A Boolean expression can contain the quantifiers ∀ (for all) and ∃ (there exists, for some) in addition to the connectors.

For example, in the universe of cats,

$$\exists \times \exists \text{ y same_color (x, y)}$$

means for some cats x and y (not necessarily distinct), x and y are the same color. This statement is obviously true. Any one cat from the universe will demonstrate its truth, since it is the same color as itself.

$$\forall \times \exists \text{ y same_color (x, y)}$$

means for any cat x, there is a cat the same color as x. This is also true, since letting y = x, x is the same color as x.

$$\forall \times \exists \text{ y same_color (x, y)} \wedge \neq \text{(x, y)}$$

means for any cat x, there is a different cat that is the same color as x. This is probably not true; the existence of any uniquely colored cat would make it false.

$$\forall \times \forall \text{ y same_color (x, y)}$$

is patently false, since this says that all cats are the same color.

If x, y are Boolean variables, their universe is much smaller than the universe of all cats—the only possible values for x and y are false and true, 0 and 1.

$$\forall \times \exists y\ (x \wedge y) \vee (\neg x \wedge \neg y)$$

This expression can be read as "For any x, either true or false, there is a truth value for y which will make the expression true." So we would need to check it for both x = true and x = false.

True: when x = 0, y = 1 makes the expression true
when x = 1, y = 0 makes it true

$$\forall \times \forall\ y\ (x \wedge y) \vee (x \wedge \neg\ y)$$

This expression says that for both x true and x false, and for both y true and y false, the expression is true. That is not correct.

False: if x = 0, expression is false when y = 0
if x = 1, expression is false when y = 0

The above expressions are in *prenex* form—that is, all the quantifiers are at the head of the expression. That is not necessary. However, any quantified Boolean expression can be rewritten in prenex form.

A variable is *bound* if it occurs within the scope of a quantifier; otherwise it is *free*. All variables are bound in the above expressions.

Example

x_1, x_2, x_3 are bound in the formula below; x_4 and x_5 are free:

$$\forall x_1 \exists x_2 \forall x_3 \quad x_1 \wedge \neg x_2 \wedge x_5 \Rightarrow \neg x_4 \vee x_2 \vee x_3$$

10.9.1 Quantified Boolean Formulas (QBFs)

A quantified Boolean formula (QBF) is a Boolean expression in prenex form with no free variables. Thus, it has the form

$$Q_1 x_1\ Q_2 x_2 \ldots Q_n x_n\ E$$

where E is a Boolean expression in x_1, …, x_n, and each Q_i is either the quantifier $\forall$ or $\exists$.

The decision problem is whether this formula is true or not.

To prove that QBF is **PSPACE**-complete, we need to know two things: (1) that the problem is in **PSPACE**, and (2) that every problem in **PSPACE** is polynomially reducible to QBF. Neither of these properties is immediately obvious.

Theorem 10.13

QBF is in **PSPACE**.

Proof

This is another application of recursion, similar to REACHABILITY. We will use recursion to remove the quantifiers, one at a time. The recursive algorithm requires only polynomial space.

Let E be a Boolean expression in prenex form, possibly with quantifiers:

1. If E contains no quantifiers, then it contains no variables and consists of the Boolean constants 0 and 1 with connectors; simply evaluate it.

2. If E begins with $\exists x_i$, then it only needs to be true for either 0 or 1.
   So, first evaluate the expression recursively with the quantifier removed and 0 substituted for x_i.
   If true, accept (no need for another recursive call).
   Otherwise, reevaluate with 1 for x_i, if true accept.
3. If E begins with $\forall x_i$, then it must be true for both 0 and 1.
   Again, substitute 0 for x_i and call recursively; if false, there is no need to call recursively.
   If true with $x_i = 0$, return result of call with $x_i = 1$.

The depth of the recursion is the number of variables, and at each stage, it need only store the value of the variable. Thus, the space used is linear in the number of variables. ■

Theorem 10.14

QBF is **PSPACE**-hard.

Proof

What we need to show is that any problem X solved by a TM M in polynomial space **P**(n), with n the length of the input, can be polynomially time reduced to QBF. As in Cook–Levin, we can encode Turing machine configurations with Boolean variables.

If we write all the Boolean expressions as in Cook–Levin, we have expressions such as S[i, j, t] for cell i holds symbol j at time t, where i is bounded by polynomial space, j is bounded by the constant |Q|, but t is only bounded by $2^{Kp(n)}$. That will not do.

So, we will use a version of reachability to construct a reduction.

Suppose c_1 and c_2 are two configurations of a polynomially space-bounded TM.

Reach (c_1, c_2, t) will be a quantified Boolean formula that is true exactly when c_2 is reachable from c_1 in t steps.

$$\text{Reach } (c_1, c_2, 0) \Leftrightarrow c_1 = c_2$$

$$\text{Reach } (c_1, c_2, 1) \Leftrightarrow c_2 \text{ is reachable from } c_1 \text{ by one move}$$

$$\text{Reach}(c_1, c_2, t) \Leftrightarrow \exists\, c\, [\text{Reach}(c_1, c, t') \wedge \text{Reach}(c, c_2, t'')]$$

where $t' = \lceil t/2 \rceil$ = greatest integer in t/2, $t'' = t - t'$.

For $t = 2^N$, Reach $(c_1, c_2, 2^N) \Leftrightarrow \exists\, c\, [\text{Reach}(c_1, c, 2^{N-1}) \wedge \text{Reach}(c, c_2, 2^{N-1})]$*.

However, this results in a QBF that is too long—the two occurrences of reach are the problem. An ingenious way to rewrite * is presented below. It is not in conjunctive normal form, but can be transformed into CNF, though one must remove both the if/then and the equality operator.

$$\exists c \forall c' \forall c''[(c' = c_1) \wedge (c'' = c)] \vee [(c' = c) \wedge (c'' = c_2)]) \Rightarrow \text{Reach}(c', c'', t')$$

Now, we want to know if Reach (start, accept, t) = true with $t = 2^{Kp(n)}$ so $t' = t/2 = 2^{Kp(n)-1}$.

To do this, recursively solve the QBF above. At each level of the recursion, we have a formula that is some constant times the size of one configuration; each configuration takes at most **P**(n) space. And, the depth of the recursion is also bounded by **P**(n). Thus, the entire recursion is bounded by $O(p^2(n))$. ■

10.10 OTHER **PSPACE**-COMPLETE PROBLEMS

The kinds of problems that turn out to be **PSPACE**-complete are interesting. Many of them come from the theory of two-person games. That may be natural, since a position is a winning position in n moves for the first player (that is, player 1 can force a win in n moves) if:

```
∃ move that player_one can make so that
∀ moves that player_two can make
∃ answering move that player_one can make
      }depth = n
∀ moves player_two has to choose from
…
Player-one has won the game
```

Standard chess will not do as an example, although only for the reason that there are a finite (up to 32) number of pieces on an 8×8 board. Thus, the size of the input cannot grow past a fixed bound. Theoretically, there are only a finite number of possible chess positions, and theoretically, we

could determine for each one what the best move would be, and then play chess by a simple table lookup. This has not happened—and is unlikely to happen—since the finite number is so large as to be unmanageable.

Games such as chess, checkers, and Go can be generalized to n x n board, easily for checkers and Go with their pieces that are all the same, and less easily for chess with its pawns, rooks, knights, and so forth. Under such generalizations, it is known that the games of Geography and Hex are **PSPACE**-complete, while the games of checkers and Go are **PSPACE**-hard.

10.11 THE CLASS EXP

Another interesting class of problems is the set of decision problems that are solvable in exponential time; that is, have time bounded by $c^{p(n)}$ for some constant c, and polynomial p, with n the size of the input. This class is naturally called **EXPTIME**, or simply **EXP**.

Facts

NP $\subseteq$ **EXP** ■

This was proved above; use transformation from NDTM to TM.

PSPACE $\subseteq$ **EXP** ■

This follows directly as corollary to Theorem 10.8.

To reiterate the important point, time and space are related.

P (poly time) $\subseteq$ **PSPACE** (poly space) $\subseteq$ **EXP** (exp time) ■

There is no easy way to find a problem in **EXP** but proved not in **NP** or **PSPACE**. Most problems are produced by diagonalization arguments, so the problem produced is not natural and is not easy to describe. The proof that a particular problem is not in either of these classes is really difficult: one has to prove that polynomial certificates do not exist, or that no algorithm can ever be found that will run in polynomial space. There have been problems (such as combinatorial game problems) that have been proved complete for **EXP**, and thus are the hardest problems in **EXP**. These

are not likely to be in **NP** or in **PSPACE** (as an **NP**-complete problem is not likely to be in **P**), but proofs have not yet been found.

Of course, one can continue making other natural classes. For example, **NEXP** (problems with a nondeterministic exponential solver) and **EXPSPACE** (problems solvable in exponential space).

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{EXPSPACE}$$

The open question remains as to which of these set inclusions are proper. It is known that

$$\mathbf{PSPACE} \subsetneq \mathbf{EXPSPACE}$$

and of course, this implies that

$$\mathbf{P} \subsetneq \mathbf{EXPSPACE}$$

but very little is known about what other set inclusions are proper.

10.12 SPACE RESTRICTIONS

In the other direction, there are interesting classes where one restricts space to be *less* than the size of the input. At first glance, this may not make sense, since one has to read the input, how can one use less space for an instance I than its size |I|? However, for many problems, one does not need to store the input—it can reside on external storage, and be read in increments. For example, one can check that an array is sorted by bringing in the elements of the array two at a time, and checking that for each successive pair, a[i] ≤ a[i + 1], so the only space needed is 2 $\log_2$ (the maximum element of a[]).

This model gives rise to deciding Turing machines that have a read-only input tape and a working tape of limited size—generally log |I|, where I is the input. If the Turing machine is a deciding TM, that is all it needs; if it is a transducer to compute f(I), it will have a third tape for writing its output.

Definition 10.15

The class LOGSPACE is the set of decision problems decided by a Turing machine with read-only input, and a working space bounded by the log of the size of the input. ■

And of course, one can analogously define NLOGSPACE.

Theorem 10.16

$$\text{LOGSPACE} \subseteq \mathbf{P}$$

Proof

Easy. ■

Examples of Problems in LOGSPACE

{A, max | a ≤ max for all a in array A} is decidable in space $\log_2$ max.
{w | w is a palindrome over {0.1}*} is decidable in space log |w|.
{G | G is a graph with no isolated nodes} is in LOGSPACE.

10.13 APPROACHES TO HARD PROBLEMS IN PRACTICE

So, what do you do if you need a solution to a problem that is **NP**-complete, or worse, **PSPACE**-complete? Or even worse, undecidable?

The bottom line is be wary, but the situation is not hopeless. For one thing, if the problem is **NP**-hard, it may only be exponential in the worst case. In practice, these hard cases may almost never arise. One approach is to use an exponential algorithm, with the expectation (hope?) that its running time for the usual sort of problem encountered will be reasonable.

In some cases, there may be an algorithm for identifying the worse (exponential) cases. Then you can run that algorithm first, and your solution second.

If an approximation to the best solution is good enough (as it may well be for Traveling Salesman), many **NP**-complete problems are known to have polynomial approximation algorithms. There are also probabilistic algorithms, which make random choices in their executions (which is different from nondeterminism). Some decision problems have good probabilistic algorithms, which will answer: "Yes, with probability 0.X." If this probability is close enough to 1.0, that may be good enough.

Even an undecidable problem may have a solution for the cases you are interested in. Do your research, find what is known, and write a program to solve the problem most of the time.

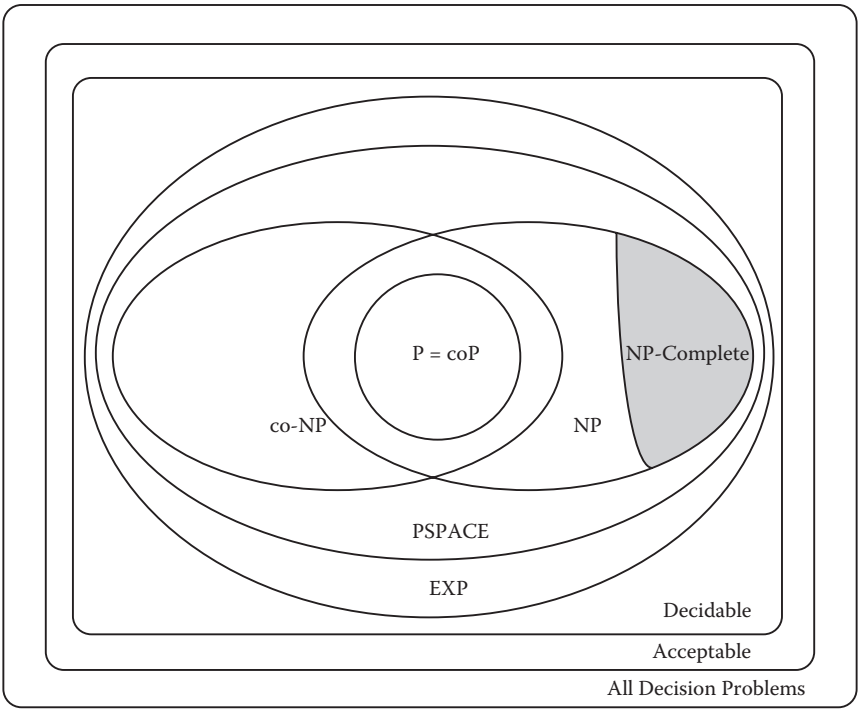


FIGURE 10.2 Relationships among language classes (assuming **P** ≠ **NP** and **NP** ≠ co-**NP**).

There is no one answer here. But, armed with what you know, you can find much more information on your particular problem. As has been said elsewhere, forewarned is forearmed.

10.14 SUMMARY

The most important classes for us have been the undecidable and intractable classes. This is because it is crucial that anyone in computer science must realize that not all problems, even not all easily stated problems, have any solution or any reasonable solution.

There are many other interesting classes of problems.

The entire text could be summarized in a diagram of the sets involved. Note that in Figure 10.2, it is assumed that both (1) **P** ≠ **NP** and (2) **NP** ≠ co-**NP.** A good exercise would be to redraw the diagram assuming that (a) **P** ≠ **NP** but **NP** = co-**NP,** and or that (b) **P** = **NP.**

EXERCISES

1. Prove that TSP-2, Decision Traveling Salesman, K-TSP but with the weights on each edge constrained to be bounded by 2, is **NP**-complete. (So it is a number problem that is strongly **NP**-complete.)

2. Prove: MULTIPROCESSOR_SCHEDULING is strongly **NP**-complete.

3. Prove: LOGSPACE $\subseteq$ **P**.

4. Answer the following:

   a. Consider some two-person game that you are familiar with (trivial example: tic-tac-toe). Write a possible generalization of this game to larger dimensions.

   b. Find by consulting the literature if the generalized version of your game is known to be **PSPACE**-complete.

5. Prove each of the following is in LOGSPACE:

   {A, max | a ≤ max for all a in array A} is decidable in space $\log_2(\max)$.

   {w | w is a palindrome over $\{0.1\}^*$} is decidable in space $\log_b |w|$, for some b.

   {G | G is a graph with no isolated nodes} is in LOGSPACE.

6. Fill in the details to complete the proof of Theorem 10.5 (if any **NP**-complete problem is also in co-**NP**, then **NP** = co-**NP**).

7. For a Turing machine with states Q and tape alphabet Γ, how many different configurations are possible using only K tape cells?

Bibliography

Bovet, D., and P. Crescenzi. 1994. *Introduction to the Theory of Complexity.* Hertfordshire, UK: Prentice Hall International.

Brainerd, W.S., and L.H. Landweber. 1974. *Theory of Computation.* New York: Wiley.

Cohen, D.I.A. 1996. *Introduction to Computer Theory*. New York: Wiley.

Cook, S. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of ACM Symposium on Theory of Computing*. ACM Press, 151–158.

Cook, S. 1983. An Overview of Computational Complexity. *Communications of the ACM*, 26, 401–408.

Cook, W.J. 2011. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation.* Princeton, NJ: Princeton University Press.

Devlin, K. 2002. *The Millennium Problems*. New York: Basic Books, 105–130.

Floyd, R.W. 1962. On Ambiguity in Phrase Structure Languages. *Communications of the ACM*, 5, 8, 526–534.

Garey, M.R., and D.S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman.

Harrison, M.A. 1978. *Introduction to Formal Language Theory*. Boston: Addison-Wesley.

Hartmanis, J., and J.E. Hopcroft. 1971. An Overview of the Theory of Computational Complexity. *Journal of the ACM*, 18, 444–475.

Hofstadter, D. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books.

Homer, S., and A.L. Selman. 2001. *Computability and Complexity Theory*. New York: Springer.

Hopcroft, J.E., Motwani, and J.D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston: Addison-Wesley.

Karp, R. 1972. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Tacher. New York: Plenum Press, 85–103.

Khachian, L. 1979. A Polynomial Algorithm in Linear Programming. *Soviet Math. Doklady* 20, 1, 191–194.

Knuth, D. 1968. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Boston: Addison-Wesley.

Ladner, R.E. 1975. On the Structure of Polynomial Time Reducibility. *Journal of ACM*, 22, 155–171.

Levin, L.A. 1973. Universal Sequential Search Problems. *Probl. Peredachi Inf.*, 9, 3, 115–116; (trans.: *Problems of Information Transmission*, 9, 3, 265–266).

Levin, L.A. 1974. *Computational Complexity of Algorithms*, ed. Kosmiadiadi and Maslov. Boston University Report BUCS-TR-85-005, 174–185.

Lewis, H.R., and C.H. Papadimitriou. 1997. *Elements of the Theory of Computation*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall.

Papadimitriou, C.H. 1994. *Computational Complexity*. Boston: Addison-Wesley.

Penrose, R. 1989. *The Emperor's New Mind: Concerning Computers, Minds and the Laws of Physics*. Oxford, UK: Oxford University Press.

Pratt, V. 1975. Every Prime Has a Succinct Certificate. *SIAM Journal of Computation*, 4, 214–220.

Savitch, W.J. 1969. Deterministic Simulation of Non-Deterministic Turing Machines [detailed abstract]. In *STOC '69: 1st Annual ACM Symposium on Theory of Computing* (New York, NY). ACM, 247–248.

Sipser, M. 2006. *Introduction to the Theory of Computation*, 2nd ed. Boston: Thomson Course Technology.

Turing, A.M. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of London Mathematical Society Series* 2, 42, 230–265.

Computer Science

Limits of Computation: An Introduction to the Undecidable and the Intractable offers a gentle introduction to the theory of computational complexity. It explains the difficulties of computation, addressing problems that have no algorithm at all and problems that cannot be solved efficiently. The book enables readers to understand:

- *What does it mean for a problem to be unsolvable or to be NP-complete?*
- *What is meant by a computation and what is a general model of a computer?*
- *What does it mean for an algorithm to exist and what kinds of problems have no algorithm?*
- *What problems have algorithms but the algorithm may take centuries to finish?*

Developed from the authors' course on computational complexity theory, the text is suitable for advanced undergraduate and beginning graduate students without a strong background in theoretical computer science. Each chapter presents the fundamentals, examples, complete proofs of theorems, and a wide range of exercises.

Features

- Helps readers understand the nature of computation
- Focuses on the two main questions of theoretical computer science
- Presents complete proofs of many NP-complete problems
- Provides in-depth discussions of difficult topics, such as different sizes of infinity, reducibility, and self-reference
- Describes Turing machines from state diagrams to encodings
- Assumes no prior knowledge of automata, advanced algorithmic analysis, and formal languages

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K13878
ISBN: 978-1-4398-8206-1
90000
9 781439 882061