*This page intentionally left blank*

Acquisitions Editor: *Matt Goldstein*

Project Editor: *Maite Suarez-Rivas*

Production Supervisor: *Marilyn Lloyd*

Marketing Manager: *Michelle Brown*

Marketing Coordinator: *Jake Zavracky*

Project Management: *Windfall Software*

Composition: *Windfall Software, using ZzTEX*

Copyeditor: *Carol Leyba*

Technical Illustration: *Dartmouth Publishing*

Proofreader: *Jennifer McClain*

Indexer: *Ted Laux*

Cover Design: *Joyce Cosentino Wells*

Cover Photo: *© 2005 Tim Laman / National Geographic. A pair of weaverbirds work together on their nest in Africa.*

Prepress and Manufacturing: *Caroline Fell*

Printer: *Courier Westford*

Access the latest information about Addison-Wesley titles from our World Wide Web site: http://www.aw-bc.com/computing

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Kleinberg, Jon.

Algorithm design / Jon Kleinberg, Eva Tardos.—1st ed. ´

p. cm.

Includes bibliographical references and index.

ISBN 0-321-29535-8 (alk. paper)

1. Computer algorithms. 2. Data structures (Computer science) I. Tardos, Eva. ´ II. Title.

QA76.9.A43K54 2005

005.1—dc22 2005000401 Copyright © 2006 by Pearson Education, Inc.

For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contract Department, 75 Arlington Street, Suite 300, Boston, MA 02116 or fax your request to (617) 848-7047.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or any toher media embodiments now known or hereafter to become known, without the prior written permission of the publisher. Printed in the United States of America.

ISBN 0-321-29535-8

12345678910-CRW-08 07 06 05

***About the Authors***

Jon Kleinberg is a professor of Computer Science at 

Cornell University. He received his Ph.D. from M.I.T.

in 1996. He is the recipient of an NSF Career Award,

an ONR Young Investigator Award, an IBM Outstand

ing Innovation Award, the National Academy of Sci

ences Award for Initiatives in Research, research fel

lowships from the Packard and Sloan Foundations,

and teaching awards from the Cornell Engineering

College and Computer Science Department.

Kleinberg’s research is centered around algorithms, particularly those con cerned with the structure of networks and information, and with applications to information science, optimization, data mining, and computational biol ogy. His work on network analysis using hubs and authorities helped form the foundation for the current generation of Internet search engines.

Eva Tardos is a professor of Computer Science at Cor- ´ 

nell University. She received her Ph.D. from Eotv ¨ os¨

University in Budapest, Hungary in 1984. She is a

member of the American Academy of Arts and Sci

ences, and an ACM Fellow; she is the recipient of an

NSF Presidential Young Investigator Award, the Fulk

erson Prize, research fellowships from the Guggen

heim, Packard, and Sloan Foundations, and teach

ing awards from the Cornell Engineering College and

Computer Science Department.

Tardos’s research interests are focused on the design and analysis of algorithms for problems on graphs or networks. She is most known for her work on network-flow algorithms and approximation algorithms for network problems. Her recent work focuses on algorithmic game theory, an emerging area concerned with designing systems and algorithms for selfish users.

*This page intentionally left blank*

***Contents***

***About the Authors*** v ***Preface*** xiii

***1 Introduction: Some Representative Problems* 1** 1.1 A First Problem: Stable Matching 1

1.2 Five Representative Problems 12

Solved Exercises 19

Exercises 22

Notes and Further Reading 28

***2 Basics of Algorithm Analysis* 29** 2.1 Computational Tractability 29

2.2 Asymptotic Order of Growth 35

2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays 42

2.4 A Survey of Common Running Times 47

2.5 A More Complex Data Structure: Priority Queues 57 Solved Exercises 65

Exercises 67

Notes and Further Reading 70

***3 Graphs* 73** 3.1 Basic Definitions and Applications 73

3.2 Graph Connectivity and Graph Traversal 78

3.3 Implementing Graph Traversal Using Queues and Stacks 87 3.4 Testing Bipartiteness: An Application of Breadth-First Search 94 3.5 Connectivity in Directed Graphs 97

**viii** Contents

3.6 Directed Acyclic Graphs and Topological Ordering 99

Solved Exercises 104

Exercises 107

Notes and Further Reading 112

***4 Greedy Algorithms* 115**

4.1 Interval Scheduling: The Greedy Algorithm Stays Ahead 116

4.2 Scheduling to Minimize Lateness: An Exchange Argument 125

4.3 Optimal Caching: A More Complex Exchange Argument 131

4.4 Shortest Paths in a Graph 137

4.5 The Minimum Spanning Tree Problem 142

4.6 Implementing Kruskal’s Algorithm: The Union-Find Data

Structure 151

4.7 Clustering 157

4.8 Huffman Codes and Data Compression 161

∗ 4.9 Minimum-Cost Arborescences: A Multi-Phase Greedy

Algorithm 177

Solved Exercises 183

Exercises 188

Notes and Further Reading 205

***5 Divide and Conquer* 209**

5.1 A First Recurrence: The Mergesort Algorithm 210

5.2 Further Recurrence Relations 214

5.3 Counting Inversions 221

5.4 Finding the Closest Pair of Points 225

5.5 Integer Multiplication 231

5.6 Convolutions and the Fast Fourier Transform 234

Solved Exercises 242

Exercises 246

Notes and Further Reading 249

***6 Dynamic Programming* 251**

6.1 Weighted Interval Scheduling: A Recursive Procedure 252

6.2 Principles of Dynamic Programming: Memoization or Iteration

over Subproblems 258

6.3 Segmented Least Squares: Multi-way Choices 261

∗ The star indicates an optional section. (See the Preface for more information about the relationships

among the chapters and sections.)

Contents **ix**

6.4 Subset Sums and Knapsacks: Adding a Variable 266

6.5 RNA Secondary Structure: Dynamic Programming over

Intervals 272

6.6 Sequence Alignment 278

6.7 Sequence Alignment in Linear Space via Divide and

Conquer 284

6.8 Shortest Paths in a Graph 290

6.9 Shortest Paths and Distance Vector Protocols 297

∗ 6.10 Negative Cycles in a Graph 301

Solved Exercises 307

Exercises 312

Notes and Further Reading 335

***7 Network Flow* 337**

7.1 The Maximum-Flow Problem and the Ford-Fulkerson

Algorithm 338

7.2 Maximum Flows and Minimum Cuts in a Network 346

7.3 Choosing Good Augmenting Paths 352

∗ 7.4 The Preflow-Push Maximum-Flow Algorithm 357

7.5 A First Application: The Bipartite Matching Problem 367

7.6 Disjoint Paths in Directed and Undirected Graphs 373

7.7 Extensions to the Maximum-Flow Problem 378

7.8 Survey Design 384

7.9 Airline Scheduling 387

7.10 Image Segmentation 391

7.11 Project Selection 396

7.12 Baseball Elimination 400

∗ 7.13 A Further Direction: Adding Costs to the Matching Problem 404

Solved Exercises 411

Exercises 415

Notes and Further Reading 448

***8 NP and Computational Intractability* 451**

8.1 Polynomial-Time Reductions 452

8.2 Reductions via “Gadgets”: The Satisfiability Problem 459

8.3 Efficient Certification and the Definition of NP 463

8.4 NP-Complete Problems 466

8.5 Sequencing Problems 473

8.6 Partitioning Problems 481

8.7 Graph Coloring 485

**x** Contents

8.8 Numerical Problems 490

8.9 Co-NP and the Asymmetry of NP 495

8.10 A Partial Taxonomy of Hard Problems 497

Solved Exercises 500

Exercises 505

Notes and Further Reading 529

***9 PSPACE: A Class of Problems beyond NP* 531**

9.1 PSPACE 531

9.2 Some Hard Problems in PSPACE 533

9.3 Solving Quantified Problems and Games in Polynomial

Space 536

9.4 Solving the Planning Problem in Polynomial Space 538

9.5 Proving Problems PSPACE-Complete 543

Solved Exercises 547

Exercises 550

Notes and Further Reading 551

***10 Extending the Limits of Tractability* 553**

10.1 Finding Small Vertex Covers 554

10.2 Solving NP-Hard Problems on Trees 558

10.3 Coloring a Set of Circular Arcs 563

∗ 10.4 Tree Decompositions of Graphs 572

∗ 10.5 Constructing a Tree Decomposition 584

Solved Exercises 591

Exercises 594

Notes and Further Reading 598

***11 Approximation Algorithms* 599**

11.1 Greedy Algorithms and Bounds on the Optimum: A Load

Balancing Problem 600

11.2 The Center Selection Problem 606

11.3 Set Cover: A General Greedy Heuristic 612

11.4 The Pricing Method: Vertex Cover 618

11.5 Maximization via the Pricing Method: The Disjoint Paths

Problem 624

11.6 Linear Programming and Rounding: An Application to Vertex

Cover 630

∗ 11.7 Load Balancing Revisited: A More Advanced LP Application 637

Contents **xi**

11.8 Arbitrarily Good Approximations: The Knapsack Problem 644

Solved Exercises 649

Exercises 651

Notes and Further Reading 659

***12 Local Search* 661**

12.1 The Landscape of an Optimization Problem 662

12.2 The Metropolis Algorithm and Simulated Annealing 666

12.3 An Application of Local Search to Hopfield Neural Networks 671

12.4 Maximum-Cut Approximation via Local Search 676

12.5 Choosing a Neighbor Relation 679

∗ 12.6 Classification via Local Search 681

12.7 Best-Response Dynamics and Nash Equilibria 690

Solved Exercises 700

Exercises 702

Notes and Further Reading 705

***13 Randomized Algorithms* 707**

13.1 A First Application: Contention Resolution 708

13.2 Finding the Global Minimum Cut 714

13.3 Random Variables and Their Expectations 719

13.4 A Randomized Approximation Algorithm for MAX 3-SAT 724

13.5 Randomized Divide and Conquer: Median-Finding and

Quicksort 727

13.6 Hashing: A Randomized Implementation of Dictionaries 734

13.7 Finding the Closest Pair of Points: A Randomized Approach 741

13.8 Randomized Caching 750

13.9 Chernoff Bounds 758

13.10 Load Balancing 760

13.11 Packet Routing 762

13.12 Background: Some Basic Probability Definitions 769

Solved Exercises 776

Exercises 782

Notes and Further Reading 793

***Epilogue: Algorithms That Run Forever*** 795

***References*** 805

***Index*** 815

*This page intentionally left blank*

***Preface***

Algorithmic ideas are pervasive, and their reach is apparent in examples both within computer science and beyond. Some of the major shifts in Internet routing standards can be viewed as debates over the deficiencies of one shortest-path algorithm and the relative advantages of another. The basic notions used by biologists to express similarities among genes and genomes have algorithmic definitions. The concerns voiced by economists over the feasibility of combinatorial auctions in practice are rooted partly in the fact that these auctions contain computationally intractable search problems as special cases. And algorithmic notions aren’t just restricted to well-known and long standing problems; one sees the reflections of these ideas on a regular basis, in novel issues arising across a wide range of areas. The scientist from Yahoo! who told us over lunch one day about their system for serving ads to users was describing a set of issues that, deep down, could be modeled as a network flow problem. So was the former student, now a management consultant working on staffing protocols for large hospitals, whom we happened to meet on a trip to New York City.

The point is not simply that algorithms have many applications. The deeper issue is that the subject of algorithms is a powerful lens through which to view the field of computer science in general. Algorithmic problems form the heart of computer science, but they rarely arrive as cleanly packaged, mathematically precise questions. Rather, they tend to come bundled together with lots of messy, application-specific detail, some of it essential, some of it extraneous. As a result, the algorithmic enterprise consists of two fundamental components: the task of getting to the mathematically clean core of a problem, and then the task of identifying the appropriate algorithm design techniques, based on the structure of the problem. These two components interact: the more comfortable one is with the full array of possible design techniques, the more one starts to recognize the clean formulations that lie within messy

**xiv** Preface

problems out in the world. At their most effective, then, algorithmic ideas do

not just provide solutions to well-posed problems; they form the language that

lets you cleanly express the underlying questions.

The goal of our book is to convey this approach to algorithms, as a design

process that begins with problems arising across the full range of computing

applications, builds on an understanding of algorithm design techniques, and

results in the development of efficient solutions to these problems. We seek

to explore the role of algorithmic ideas in computer science generally, and

relate these ideas to the range of precisely formulated problems for which we

can design and analyze algorithms. In other words, what are the underlying

issues that motivate these problems, and how did we choose these particular

ways of formulating them? How did we recognize which design principles were

appropriate in different situations?

In keeping with this, our goal is to offer advice on how to identify clean

algorithmic problem formulations in complex issues from different areas of

computing and, from this, how to design efficient algorithms for the resulting

problems. Sophisticated algorithms are often best understood by reconstruct

ing the sequence of ideas—including false starts and dead ends—that led from

simpler initial approaches to the eventual solution. The result is a style of ex

position that does not take the most direct route from problem statement to

algorithm, but we feel it better reflects the way that we and our colleagues

genuinely think about these questions.

**Overview**

The book is intended for students who have completed a programming

based two-semester introductory computer science sequence (the standard

“CS1/CS2” courses) in which they have written programs that implement

basic algorithms, manipulate discrete structures such as trees and graphs, and

apply basic data structures such as arrays, lists, queues, and stacks. Since

the interface between CS1/CS2 and a first algorithms course is not entirely

standard, we begin the book with self-contained coverage of topics that at

some institutions are familiar to students from CS1/CS2, but which at other

institutions are included in the syllabi of the first algorithms course. This

material can thus be treated either as a review or as new material; by including

it, we hope the book can be used in a broader array of courses, and with more

flexibility in the prerequisite knowledge that is assumed.

In keeping with the approach outlined above, we develop the basic algo

rithm design techniques by drawing on problems from across many areas of

computer science and related fields. To mention a few representative examples

here, we include fairly detailed discussions of applications from systems and

networks (caching, switching, interdomain routing on the Internet), artificial

Preface **xv**

intelligence (planning, game playing, Hopfield networks), computer vision

(image segmentation), data mining (change-point detection, clustering), op

erations research (airline scheduling), and computational biology (sequence

alignment, RNA secondary structure).

The notion of computational intractability, and NP-completeness in par

ticular, plays a large role in the book. This is consistent with how we think

about the overall process of algorithm design. Some of the time, an interest

ing problem arising in an application area will be amenable to an efficient

solution, and some of the time it will be provably NP-complete; in order to

fully address a new algorithmic problem, one should be able to explore both

of these options with equal familiarity. Since so many natural problems in

computer science are NP-complete, the development of methods to deal with

intractable problems has become a crucial issue in the study of algorithms,

and our book heavily reflects this theme. The discovery that a problem is NP

complete should not be taken as the end of the story, but as an invitation to

begin looking for approximation algorithms, heuristic local search techniques,

or tractable special cases. We include extensive coverage of each of these three

approaches.

**Problems and Solved Exercises**

An important feature of the book is the collection of problems. Across all

chapters, the book includes over 200 problems, almost all of them developed

and class-tested in homework or exams as part of our teaching of the course

at Cornell. We view the problems as a crucial component of the book, and

they are structured in keeping with our overall approach to the material. Most

of them consist of extended verbal descriptions of a problem arising in an

application area in computer science or elsewhere out in the world, and part of

the problem is to practice what we discuss in the text: setting up the necessary

notation and formalization, designing an algorithm, and then analyzing it and

proving it correct. (We view a complete answer to one of these problems as

consisting of all these components: a fully explained algorithm, an analysis of

the running time, and a proof of correctness.) The ideas for these problems

come in large part from discussions we have had over the years with people

working in different areas, and in some cases they serve the dual purpose of

recording an interesting (though manageable) application of algorithms that

we haven’t seen written down anywhere else.

To help with the process of working on these problems, we include in

each chapter a section entitled “Solved Exercises,” where we take one or more

problems and describe how to go about formulating a solution. The discussion

devoted to each solved exercise is therefore significantly longer than what

would be needed simply to write a complete, correct solution (in other words,

**xvi** Preface

significantly longer than what it would take to receive full credit if these were

being assigned as homework problems). Rather, as with the rest of the text,

the discussions in these sections should be viewed as trying to give a sense

of the larger process by which one might think about problems of this type,

culminating in the specification of a precise solution.

It is worth mentioning two points concerning the use of these problems

as homework in a course. First, the problems are sequenced roughly in order

of increasing difficulty, but this is only an approximate guide and we advise

against placing too much weight on it: since the bulk of the problems were

designed as homework for our undergraduate class, large subsets of the

problems in each chapter are really closely comparable in terms of difficulty.

Second, aside from the lowest-numbered ones, the problems are designed to

involve some investment of time, both to relate the problem description to the

algorithmic techniques in the chapter, and then to actually design the necessary

algorithm. In our undergraduate class, we have tended to assign roughly three

of these problems per week.

**Pedagogical Features and Supplements**

In addition to the problems and solved exercises, the book has a number of

further pedagogical features, as well as additional supplements to facilitate its

use for teaching.

As noted earlier, a large number of the sections in the book are devoted

to the formulation of an algorithmic problem—including its background and

underlying motivation—and the design and analysis of an algorithm for this

problem. To reflect this style, these sections are consistently structured around

a sequence of subsections: “The Problem,” where the problem is described

and a precise formulation is worked out; “Designing the Algorithm,” where

the appropriate design technique is employed to develop an algorithm; and

“Analyzing the Algorithm,” which proves properties of the algorithm and

analyzes its efficiency. These subsections are highlighted in the text with an

icon depicting a feather. In cases where extensions to the problem or further

analysis of the algorithm is pursued, there are additional subsections devoted

to these issues. The goal of this structure is to offer a relatively uniform style

of presentation that moves from the initial discussion of a problem arising in a

computing application through to the detailed analysis of a method to solve it.

A number of supplements are available in support of the book itself. An

instructor’s manual works through all the problems, providing full solutions to

each. A set of lecture slides, developed by Kevin Wayne of Princeton University,

is also available; these slides follow the order of the book’s sections and can

thus be used as the foundation for lectures in a course based on the book. These

files are available at *www.aw.com*. For instructions on obtaining a professor

Preface **xvii**

login and password, search the site for either “Kleinberg” or “Tardos” or

contact your local Addison-Wesley representative.

Finally, we would appreciate receiving feedback on the book. In particular,

as in any book of this length, there are undoubtedly errors that have remained

in the final version. Comments and reports of errors can be sent to us by e-mail,

at the address algbook@cs.cornell.edu; please include the word “feedback”

in the subject line of the message.

**Chapter-by-Chapter Synopsis**

Chapter 1 starts by introducing some representative algorithmic problems. We

begin immediately with the Stable Matching Problem, since we feel it sets

up the basic issues in algorithm design more concretely and more elegantly

than any abstract discussion could: stable matching is motivated by a natural

though complex real-world issue, from which one can abstract an interesting

problem statement and a surprisingly effective algorithm to solve this problem.

The remainder of Chapter 1 discusses a list of five “representative problems”

that foreshadow topics from the remainder of the course. These five problems

are interrelated in the sense that they are all variations and/or special cases

of the Independent Set Problem; but one is solvable by a greedy algorithm,

one by dynamic programming, one by network flow, one (the Independent

Set Problem itself) is NP-complete, and one is PSPACE-complete. The fact that

closely related problems can vary greatly in complexity is an important theme

of the book, and these five problems serve as milestones that reappear as the

book progresses.

Chapters 2 and 3 cover the interface to the CS1/CS2 course sequence

mentioned earlier. Chapter 2 introduces the key mathematical definitions and

notations used for analyzing algorithms, as well as the motivating principles

behind them. It begins with an informal overview of what it means for a prob

lem to be computationally tractable, together with the concept of polynomial

time as a formal notion of efficiency. It then discusses growth rates of func

tions and asymptotic analysis more formally, and offers a guide to commonly

occurring functions in algorithm analysis, together with standard applications

in which they arise. Chapter 3 covers the basic definitions and algorithmic

primitives needed for working with graphs, which are central to so many of

the problems in the book. A number of basic graph algorithms are often im

plemented by students late in the CS1/CS2 course sequence, but it is valuable

to present the material here in a broader algorithm design context. In par

ticular, we discuss basic graph definitions, graph traversal techniques such

as breadth-first search and depth-first search, and directed graph concepts

including strong connectivity and topological ordering.

**xviii** Preface

Chapters 2 and 3 also present many of the basic data structures that will

be used for implementing algorithms throughout the book; more advanced

data structures are presented in subsequent chapters. Our approach to data

structures is to introduce them as they are needed for the implementation of

the algorithms being developed in the book. Thus, although many of the data

structures covered here will be familiar to students from the CS1/CS2 sequence,

our focus is on these data structures in the broader context of algorithm design

and analysis.

Chapters 4 through 7 cover four major algorithm design techniques: greedy

algorithms, divide and conquer, dynamic programming, and network flow.

With greedy algorithms, the challenge is to recognize when they work and

when they don’t; our coverage of this topic is centered around a way of clas

sifying the kinds of arguments used to prove greedy algorithms correct. This

chapter concludes with some of the main applications of greedy algorithms,

for shortest paths, undirected and directed spanning trees, clustering, and

compression. For divide and conquer, we begin with a discussion of strategies

for solving recurrence relations as bounds on running times; we then show

how familiarity with these recurrences can guide the design of algorithms that

improve over straightforward approaches to a number of basic problems, in

cluding the comparison of rankings, the computation of closest pairs of points

in the plane, and the Fast Fourier Transform. Next we develop dynamic pro

gramming by starting with the recursive intuition behind it, and subsequently

building up more and more expressive recurrence formulations through appli

cations in which they naturally arise. This chapter concludes with extended

discussions of the dynamic programming approach to two fundamental prob

lems: sequence alignment, with applications in computational biology; and

shortest paths in graphs, with connections to Internet routing protocols. Fi

nally, we cover algorithms for network flow problems, devoting much of our

focus in this chapter to discussing a large array of different flow applications.

To the extent that network flow is covered in algorithms courses, students are

often left without an appreciation for the wide range of problems to which it

can be applied; we try to do justice to its versatility by presenting applications

to load balancing, scheduling, image segmentation, and a number of other

problems.

Chapters 8 and 9 cover computational intractability. We devote most of

our attention to NP-completeness, organizing the basic NP-complete problems

thematically to help students recognize candidates for reductions when they

encounter new problems. We build up to some fairly complex proofs of NP

completeness, with guidance on how one goes about constructing a difficult

reduction. We also consider types of computational hardness beyond NP

completeness, particularly through the topic of PSPACE-completeness. We

Preface **xix**

find this is a valuable way to emphasize that intractability doesn’t end at

NP-completeness, and PSPACE-completeness also forms the underpinning for

some central notions from artificial intelligence—planning and game playing—

that would otherwise not find a place in the algorithmic landscape we are

surveying.

Chapters 10 through 12 cover three major techniques for dealing with com

putationally intractable problems: identification of structured special cases,

approximation algorithms, and local search heuristics. Our chapter on tractable

special cases emphasizes that instances of NP-complete problems arising in

practice may not be nearly as hard as worst-case instances, because they often

contain some structure that can be exploited in the design of an efficient algo

rithm. We illustrate how NP-complete problems are often efficiently solvable

when restricted to tree-structured inputs, and we conclude with an extended

discussion of tree decompositions of graphs. While this topic is more suit

able for a graduate course than for an undergraduate one, it is a technique

with considerable practical utility for which it is hard to find an existing

accessible reference for students. Our chapter on approximation algorithms

discusses both the process of designing effective algorithms and the task of

understanding the optimal solution well enough to obtain good bounds on it.

As design techniques for approximation algorithms, we focus on greedy algo

rithms, linear programming, and a third method we refer to as “pricing,” which

incorporates ideas from each of the first two. Finally, we discuss local search

heuristics, including the Metropolis algorithm and simulated annealing. This

topic is often missing from undergraduate algorithms courses, because very

little is known in the way of provable guarantees for these algorithms; how

ever, given their widespread use in practice, we feel it is valuable for students

to know something about them, and we also include some cases in which

guarantees can be proved.

Chapter 13 covers the use of randomization in the design of algorithms.

This is a topic on which several nice graduate-level books have been written.

Our goal here is to provide a more compact introduction to some of the

ways in which students can apply randomized techniques using the kind of

background in probability one typically gains from an undergraduate discrete

math course.

**Use of the Book**

The book is primarily designed for use in a first undergraduate course on

algorithms, but it can also be used as the basis for an introductory graduate

course.

When we use the book at the undergraduate level, we spend roughly

one lecture per numbered section; in cases where there is more than one

**xx** Preface

lecture’s worth of material in a section (for example, when a section provides

further applications as additional examples), we treat this extra material as a

supplement that students can read about outside of lecture. We skip the starred

sections; while these sections contain important topics, they are less central

to the development of the subject, and in some cases they are harder as well.

We also tend to skip one or two other sections per chapter in the first half of

the book (for example, we tend to skip Sections 4.3, 4.7–4.8, 5.5–5.6, 6.5, 7.6,

and 7.11). We cover roughly half of each of Chapters 11–13.

This last point is worth emphasizing: rather than viewing the later chapters

as “advanced,” and hence off-limits to undergraduate algorithms courses, we

have designed them with the goal that the first few sections of each should

be accessible to an undergraduate audience. Our own undergraduate course

involves material from all these chapters, as we feel that all of these topics

have an important place at the undergraduate level.

Finally, we treat Chapters 2 and 3 primarily as a review of material from

earlier courses; but, as discussed above, the use of these two chapters depends

heavily on the relationship of each specific course to its prerequisites.

The resulting syllabus looks roughly as follows: Chapter 1; Chapters 4–8

(excluding 4.3, 4.7–4.9, 5.5–5.6, 6.5, 6.10, 7.4, 7.6, 7.11, and 7.13); Chapter 9

(briefly); Chapter 10, Sections.10.1 and 10.2; Chapter 11, Sections 11.1, 11.2,

11.6, and 11.8; Chapter 12, Sections 12.1–12.3; and Chapter 13, Sections 13.1–

13.5.

The book also naturally supports an introductory graduate course on

algorithms. Our view of such a course is that it should introduce students

destined for research in all different areas to the important current themes in

algorithm design. Here we find the emphasis on formulating problems to be

useful as well, since students will soon be trying to define their own research

problems in many different subfields. For this type of course, we cover the

later topics in Chapters 4 and 6 (Sections 4.5–4.9 and 6.5–6.10), cover all of

Chapter 7 (moving more rapidly through the early sections), quickly cover NP

completeness in Chapter 8 (since many beginning graduate students will have

seen this topic as undergraduates), and then spend the remainder of the time

on Chapters 10–13. Although our focus in an introductory graduate course is

on the more advanced sections, we find it useful for the students to have the

full book to consult for reviewing or filling in background knowledge, given

the range of different undergraduate backgrounds among the students in such

a course.

Finally, the book can be used to support self-study by graduate students,

researchers, or computer professionals who want to get a sense for how they

Preface **xxi**

might be able to use particular algorithm design techniques in the context of

their own work. A number of graduate students and colleagues have used

portions of the book in this way.

**Acknowledgments**

This book grew out of the sequence of algorithms courses that we have taught

at Cornell. These courses have grown, as the field has grown, over a number of

years, and they reflect the influence of the Cornell faculty who helped to shape

them during this time, including Juris Hartmanis, Monika Henzinger, John

Hopcroft, Dexter Kozen, Ronitt Rubinfeld, and Sam Toueg. More generally, we

would like to thank all our colleagues at Cornell for countless discussions both

on the material here and on broader issues about the nature of the field.

The course staffs we’ve had in teaching the subject have been tremen

dously helpful in the formulation of this material. We thank our undergradu

ate and graduate teaching assistants, Siddharth Alexander, Rie Ando, Elliot

Anshelevich, Lars Backstrom, Steve Baker, Ralph Benzinger, John Bicket,

Doug Burdick, Mike Connor, Vladimir Dizhoor, Shaddin Doghmi, Alexan

der Druyan, Bowei Du, Sasha Evfimievski, Ariful Gani, Vadim Grinshpun,

Ara Hayrapetyan, Chris Jeuell, Igor Kats, Omar Khan, Mikhail Kobyakov,

Alexei Kopylov, Brian Kulis, Amit Kumar, Yeongwee Lee, Henry Lin, Ash

win Machanavajjhala, Ayan Mandal, Bill McCloskey, Leonid Meyerguz, Evan

Moran, Niranjan Nagarajan, Tina Nolte, Travis Ortogero, Martin Pal, Jon ´

Peress, Matt Piotrowski, Joe Polastre, Mike Priscott, Xin Qi, Venu Ramasubra

manian, Aditya Rao, David Richardson, Brian Sabino, Rachit Siamwalla, Se

bastian Silgardo, Alex Slivkins, Chaitanya Swamy, Perry Tam, Nadya Travinin,

Sergei Vassilvitskii, Matthew Wachs, Tom Wexler, Shan-Leung Maverick Woo,

Justin Yang, and Misha Zatsman. Many of them have provided valuable in

sights, suggestions, and comments on the text. We also thank all the students

in these classes who have provided comments and feedback on early drafts of

the book over the years.

For the past several years, the development of the book has benefited

greatly from the feedback and advice of colleagues who have used prepubli

cation drafts for teaching. Anna Karlin fearlessly adopted a draft as her course

textbook at the University of Washington when it was still in an early stage of

development; she was followed by a number of people who have used it either

as a course textbook or as a resource for teaching: Paul Beame, Allan Borodin,

Devdatt Dubhashi, David Kempe, Gene Kleinberg, Dexter Kozen, Amit Kumar,

Mike Molloy, Yuval Rabani, Tim Roughgarden, Alexa Sharp, Shanghua Teng,

Aravind Srinivasan, Dieter van Melkebeek, Kevin Wayne, Tom Wexler, and

**xxii** Preface

Sue Whitesides. We deeply appreciate their input and advice, which has in

formed many of our revisions to the content. We would like to additionally

thank Kevin Wayne for producing supplementary material associated with the

book, which promises to greatly extend its utility to future instructors.

In a number of other cases, our approach to particular topics in the book

reflects the infuence of specific colleagues. Many of these contributions have

undoubtedly escaped our notice, but we especially thank Yuri Boykov, Ron

Elber, Dan Huttenlocher, Bobby Kleinberg, Evie Kleinberg, Lillian Lee, David

McAllester, Mark Newman, Prabhakar Raghavan, Bart Selman, David Shmoys,

Steve Strogatz, Olga Veksler, Duncan Watts, and Ramin Zabih.

It has been a pleasure working with Addison Wesley over the past year.

First and foremost, we thank Matt Goldstein for all his advice and guidance in

this process, and for helping us to synthesize a vast amount of review material

into a concrete plan that improved the book. Our early conversations about

the book with Susan Hartman were extremely valuable as well. We thank Matt

and Susan, together with Michelle Brown, Marilyn Lloyd, Patty Mahtani, and

Maite Suarez-Rivas at Addison Wesley, and Paul Anagnostopoulos and Jacqui

Scarlott at Windfall Software, for all their work on the editing, production, and

management of the project. We further thank Paul and Jacqui for their expert

composition of the book. We thank Joyce Wells for the cover design, Nancy

Murphy of Dartmouth Publishing for her work on the figures, Ted Laux for

the indexing, and Carol Leyba and Jennifer McClain for the copyediting and

proofreading.

We thank Anselm Blumer (Tufts University), Richard Chang (University of

Maryland, Baltimore County), Kevin Compton (University of Michigan), Diane

Cook (University of Texas, Arlington), Sariel Har-Peled (University of Illinois,

Urbana-Champaign), Sanjeev Khanna (University of Pennsylvania), Philip

Klein (Brown University), David Matthias (Ohio State University), Adam Mey

erson (UCLA), Michael Mitzenmacher (Harvard University), Stephan Olariu

(Old Dominion University), Mohan Paturi (UC San Diego), Edgar Ramos (Uni

versity of Illinois, Urbana-Champaign), Sanjay Ranka (University of Florida,

Gainesville), Leon Reznik (Rochester Institute of Technology), Subhash Suri

(UC Santa Barbara), Dieter van Melkebeek (University of Wisconsin, Madi

son), and Bulent Yener (Rensselaer Polytechnic Institute) who generously

contributed their time to provide detailed and thoughtful reviews of the man

uscript; their comments led to numerous improvements, both large and small,

in the final version of the text.

Finally, we thank our families—Lillian and Alice, and David, Rebecca, and

Amy. We appreciate their support, patience, and many other contributions

more than we can express in any acknowledgments here.

Preface **xxiii**

This book was begun amid the irrational exuberance of the late nineties,

when the arc of computing technology seemed, to many of us, briefly to pass

through a place traditionally occupied by celebrities and other inhabitants of

the pop-cultural firmament. (It was probably just in our imaginations.) Now,

several years after the hype and stock prices have come back to earth, one can

appreciate that in some ways computer science was forever changed by this

period, and in other ways it has remained the same: the driving excitement

that has characterized the field since its early days is as strong and enticing as

ever, the public’s fascination with information technology is still vibrant, and

the reach of computing continues to extend into new disciplines. And so to

all students of the subject, drawn to it for so many different reasons, we hope

you find this book an enjoyable and useful guide wherever your computational

pursuits may take you.

Jon Kleinberg

Eva Tardos ´

Ithaca, 2005

*This page intentionally left blank*

***Chapter 1***

***Introduction: Some***

***Representative Problems***

1.1 A First Problem: Stable Matching

As an opening topic, we look at an algorithmic problem that nicely illustrates many of the themes we will be emphasizing. It is motivated by some very natural and practical concerns, and from these we formulate a clean and simple statement of a problem. The algorithm to solve the problem is very clean as well, and most of our work will be spent in proving that it is correct and giving an acceptable bound on the amount of time it takes to terminate with an answer. The problem itself—the *Stable Matching Problem*—has several origins.

**The Problem**

The Stable Matching Problem originated, in part, in 1962, when David Gale and Lloyd Shapley, two mathematical economists, asked the question: Could one design a college admissions process, or a job recruiting process, that was *self-enforcing*? What did they mean by this?

To set up the question, let’s first think informally about the kind of situation that might arise as a group of friends, all juniors in college majoring in computer science, begin applying to companies for summer internships. The crux of the application process is the interplay between two different types of parties: companies (the employers) and students (the applicants). Each applicant has a preference ordering on companies, and each company—once the applications come in—forms a preference ordering on its applicants. Based on these preferences, companies extend offers to some of their applicants, applicants choose which of their offers to accept, and people begin heading off to their summer internships.

**2** Chapter 1 Introduction: Some Representative Problems

Gale and Shapley considered the sorts of things that could start going

wrong with this process, in the absence of any mechanism to enforce the status

quo. Suppose, for example, that your friend Raj has just accepted a summer job

at the large telecommunications company CluNet. A few days later, the small

start-up company WebExodus, which had been dragging its feet on making a

few final decisions, calls up Raj and offers him a summer job as well. Now, Raj

actually prefers WebExodus to CluNet—won over perhaps by the laid-back,

anything-can-happen atmosphere—and so this new development may well

cause him to retract his acceptance of the CluNet offer and go to WebExodus

instead. Suddenly down one summer intern, CluNet offers a job to one of its

wait-listed applicants, who promptly retracts his previous acceptance of an

offer from the software giant Babelsoft, and the situation begins to spiral out

of control.

Things look just as bad, if not worse, from the other direction. Suppose

that Raj’s friend Chelsea, destined to go to Babelsoft but having just heard Raj’s

story, calls up the people at WebExodus and says, “You know, I’d really rather

spend the summer with you guys than at Babelsoft.” They find this very easy

to believe; and furthermore, on looking at Chelsea’s application, they realize

that they would have rather hired her than some other student who actually

*is* scheduled to spend the summer at WebExodus. In this case, if WebExodus

were a slightly less scrupulous company, it might well find some way to retract

its offer to this other student and hire Chelsea instead.

Situations like this can rapidly generate a lot of chaos, and many people—

both applicants and employers—can end up unhappy with the process as well

as the outcome. What has gone wrong? One basic problem is that the process

is not self-enforcing—if people are allowed to act in their self-interest, then it

risks breaking down.

We might well prefer the following, more stable situation, in which self

interest itself prevents offers from being retracted and redirected. Consider

another student, who has arranged to spend the summer at CluNet but calls

up WebExodus and reveals that he, too, would rather work for them. But in

this case, based on the offers already accepted, they are able to reply, “No, it

turns out that we prefer each of the students we’ve accepted to you, so we’re

afraid there’s nothing we can do.” Or consider an employer, earnestly following

up with its top applicants who went elsewhere, being told by each of them,

“No, I’m happy where I am.” In such a case, all the outcomes are stable—there

are no further outside deals that can be made.

So this is the question Gale and Shapley asked: Given a set of preferences

among employers and applicants, can we assign applicants to employers so

that for every employer *E*, and every applicant *A* who is not scheduled to work

for *E*, at least one of the following two things is the case?

1.1 A First Problem: Stable Matching **3**

(i) *E* prefers every one of its accepted applicants to *A*; or

(ii) *A* prefers her current situation over working for employer *E*.

If this holds, the outcome is stable: individual self-interest will prevent any

applicant/employer deal from being made behind the scenes.

Gale and Shapley proceeded to develop a striking algorithmic solution to

this problem, which we will discuss presently. Before doing this, let’s note that

this is not the only origin of the Stable Matching Problem. It turns out that for

a decade before the work of Gale and Shapley, unbeknownst to them, the

National Resident Matching Program had been using a very similar procedure,

with the same underlying motivation, to match residents to hospitals. Indeed,

this system, with relatively little change, is still in use today.

This is one testament to the problem’s fundamental appeal. And from the

point of view of this book, it provides us with a nice first domain in which

to reason about some basic combinatorial definitions and the algorithms that

build on them.

***Formulating the Problem*** To get at the essence of this concept, it helps to

make the problem as clean as possible. The world of companies and applicants

contains some distracting asymmetries. Each applicant is looking for a single

company, but each company is looking for many applicants; moreover, there

may be more (or, as is sometimes the case, fewer) applicants than there are

available slots for summer jobs. Finally, each applicant does not typically apply

to every company.

It is useful, at least initially, to eliminate these complications and arrive at a

more “bare-bones” version of the problem: each of *n* applicants applies to each

of *n* companies, and each company wants to accept a *single* applicant. We will

see that doing this preserves the fundamental issues inherent in the problem;

in particular, our solution to this simplified version will extend directly to the

more general case as well.

Following Gale and Shapley, we observe that this special case can be

viewed as the problem of devising a system by which each of *n* men and

*n* women can end up getting married: our problem naturally has the analogue

of two “genders”—the applicants and the companies—and in the case we are

considering, everyone is seeking to be paired with exactly one individual of

the opposite gender.1

1 Gale and Shapley considered the same-sex Stable Matching Problem as well, where there is only a

single gender. This is motivated by related applications, but it turns out to be fairly different at a

technical level. Given the applicant-employer application we’re considering here, we’ll be focusing

on the version with two genders.

**4** Chapter 1 Introduction: Some Representative Problems

So consider a set *M* = {*m*1,..., *mn*} of *n* men, and a set *W* = {*w*1,..., *wn*}

of *n* women. Let *M* × *W* denote the set of all possible ordered pairs of the form

*(m*, *w)*, where *m* ∈ *M* and *w* ∈ *W*. A *matching S* is a *set* of ordered pairs, each

from *M* × *W*, with the property that each member of *M* and each member of

*W* appears in at most one pair in *S*. A *perfect matching S*is a matching with

the property that each member of *M* and each member of *W* appears in *exactly*

one pair in *S*.

Matchings and perfect matchings are objects that will recur frequently

An instability: *m* and *w* each prefer the other to their current partners.

throughout the book; they arise naturally in modeling a wide range of algo rithmic problems. In the present situation, a perfect matching corresponds simply to a way of pairing off the men with the women, in such a way that everyone ends up married to somebody, and nobody is married to more than one person—there is neither singlehood nor polygamy.

Now we can add the notion of *preferences* to this setting. Each man *m* ∈ *M ranks* all the women; we will say that *m prefers w to w*if *m* ranks *w* higher

*m*

than *w*. We will refer to the ordered ranking of *m* as his *preference list*. We will *~~w~~*

*m**~~w~~*

Figure 1.1 Perfect matching *S* with instability *(m*, *w**)*.

not allow ties in the ranking. Each woman, analogously, ranks all the men. Given a perfect matching *S*, what can go wrong? Guided by our initial motivation in terms of employers and applicants, we should be worried about the following situation: There are two pairs *(m*, *w)* and *(m*, *w**)* in *S* (as depicted in Figure 1.1) with the property that *m* prefers *w*to *w*, and *w*prefers *m* to *m*. In this case, there’s nothing to stop *m* and *w*from abandoning their current partners and heading off together; the set of marriages is not self enforcing. We’ll say that such a pair *(m*, *w**)* is an *instability* with respect to *S*: *(m*, *w**)* does not belong to *S*, but each of *m* and *w*prefers the other to their partner in *S*.

Our goal, then, is a set of marriages with no instabilities. We’ll say that a matching *S* is *stable* if (i) it is perfect, and (ii) there is no instability with respect to *S*. Two questions spring immediately to mind:

. Does there exist a stable matching for every set of preference lists? . Given a set of preference lists, can we efficiently construct a stable matching if there is one?

***Some Examples*** To illustrate these definitions, consider the following two very simple instances of the Stable Matching Problem.

First, suppose we have a set of two men, {*m*, *m*}, and a set of two women, {*w*, *w*}. The preference lists are as follows:

*m* prefers *w* to *w*.

*m*prefers *w* to *w*.

1.1 A First Problem: Stable Matching **5**

*w* prefers *m* to *m*.

*w*prefers *m* to *m*.

If we think about this set of preference lists intuitively, it represents complete

agreement: the men agree on the order of the women, and the women agree

on the order of the men. There is a unique stable matching here, consisting

of the pairs *(m*, *w)* and *(m*, *w**)*. The other perfect matching, consisting of the

pairs *(m*, *w)* and *(m*, *w**)*, would not be a stable matching, because the pair

*(m*, *w)* would form an instability with respect to this matching. (Both *m* and

*w* would want to leave their respective partners and pair up.)

Next, here’s an example where things are a bit more intricate. Suppose

the preferences are

*m* prefers *w* to *w*.

*m*prefers *w*to *w*.

*w* prefers *m*to *m*.

*w*prefers *m* to *m*.

What’s going on in this case? The two men’s preferences mesh perfectly with

each other (they rank different women first), and the two women’s preferences

likewise mesh perfectly with each other. But the men’s preferences clash

completely with the women’s preferences.

In this second example, there are two different stable matchings. The

matching consisting of the pairs *(m*, *w)* and *(m*, *w**)* is stable, because both

men are as happy as possible, so neither would leave their matched partner.

But the matching consisting of the pairs *(m*, *w)* and *(m*, *w**)* is also stable, for

the complementary reason that both women are as happy as possible. This is

an important point to remember as we go forward—it’s possible for an instance

to have more than one stable matching.

**Designing the Algorithm**

We now show that there exists a stable matching for every set of preference

lists among the men and women. Moreover, our means of showing this will

also answer the second question that we asked above: we will give an efficient

algorithm that takes the preference lists and constructs a stable matching.

Let us consider some of the basic ideas that motivate the algorithm.

. Initially, everyone is unmarried. Suppose an unmarried man *m* chooses

the woman *w* who ranks highest on his preference list and *proposes* to

her. Can we declare immediately that*(m*, *w)* will be one of the pairs in our

final stable matching? Not necessarily: at some point in the future, a man

*m*whom *w* prefers may propose to her. On the other hand, it would be

**6** Chapter 1 Introduction: Some Representative Problems dangerous for *w* to reject *m* right away; she may never receive a proposal

Woman *w* will become engaged to *m* if she prefers him to *m* .

from someone she ranks as highly as *m*. So a natural idea would be to have the pair *(m*, *w)* enter an intermediate state—*engagement*. . Suppose we are now at a state in which some men and women are *free*— not engaged—and some are engaged. The next step could look like this. An arbitrary free man *m* chooses the highest-ranked woman *w* to whom he has not yet proposed, and he proposes to her. If *w* is also free, then *m* and *w* become engaged. Otherwise, *w* is already engaged to some other man *m*. In this case, she determines which of *m* or *m*ranks higher on her preference list; this man becomes engaged to *w* and the other becomes free.

*m m*

*~~w~~*

. Finally, the algorithm will terminate when no one is free; at this moment, all engagements are declared final, and the resulting perfect matching is returned.

Here is a concrete description of the *Gale-Shapley algorithm*, with Fig

Figure 1.2 An intermediate state of the G-S algorithm when a free man *m* is propos ing to a woman *w*.

ure 1.2 depicting a state of the algorithm.

Initially all *m* ∈ *M* and *w* ∈ *W* are free

While there is a man *m* who is free and hasn’t proposed to every woman

Choose such a man *m*

Let *w* be the highest-ranked woman in *m*’s preference list to whom *m* has not yet proposed

If *w* is free then

*(m*, *w)* become engaged

Else *w* is currently engaged to *m*

If *w* prefers *m*to *m* then

*m* remains free

Else *w* prefers *m* to *m*

*(m*, *w)* become engaged

*m*becomes free

Endif

Endif

Endwhile

Return the set *S* of engaged pairs

An intriguing thing is that, although the G-S algorithm is quite simple to state, it is not immediately obvious that it returns a stable matching, or even a perfect matching. We proceed to prove this now, through a sequence of intermediate facts.

1.1 A First Problem: Stable Matching **7**

**Analyzing the Algorithm**

First consider the view of a woman *w* during the execution of the algorithm.

For a while, no one has proposed to her, and she is free. Then a man *m* may

propose to her, and she becomes engaged. As time goes on, she may receive

additional proposals, accepting those that increase the rank of her partner. So

we discover the following.

(1.1) *w remains engaged from the point at which she receives her first*

*proposal; and the sequence of partners to which she is engaged gets better and*

*better (in terms of her preference list).*

The view of a man *m* during the execution of the algorithm is rather

different. He is free until he proposes to the highest-ranked woman on his

list; at this point he may or may not become engaged. As time goes on, he

may alternate between being free and being engaged; however, the following

property does hold.

(1.2) *The sequence of women to whom m proposes gets worse and worse (in*

*terms of his preference list).*

Now we show that the algorithm terminates, and give a bound on the

maximum number of iterations needed for termination.

(1.3) *The G-S algorithm terminates after at most n*2 *iterations of the* While

*loop.*

Proof. A useful strategy for upper-bounding the running time of an algorithm,

as we are trying to do here, is to find a measure of *progress*. Namely, we seek

some precise way of saying that each step taken by the algorithm brings it

closer to termination.

In the case of the present algorithm, each iteration consists of some man

proposing (for the only time) to a woman he has never proposed to before. So

if we let P*(t)* denote the set of pairs *(m*, *w)* such that *m* has proposed to *w* by

the end of iteration *t*, we see that for all *t*, the size of P*(t* + 1*)* is strictly greater

than the size of P*(t)*. But there are only *n*2 possible pairs of men and women

in total, so the value of P*(*·*)* can increase at most *n*2 times over the course of

the algorithm. It follows that there can be at most *n*2 iterations.

Two points are worth noting about the previous fact and its proof. First,

there are executions of the algorithm (with certain preference lists) that can

involve close to *n*2 iterations, so this analysis is not far from the best possible.

Second, there are many quantities that would not have worked well as a

*progress measure* for the algorithm, since they need not strictly increase in each

**8** Chapter 1 Introduction: Some Representative Problems

iteration. For example, the number of free individuals could remain constant

from one iteration to the next, as could the number of engaged pairs. Thus,

these quantities could not be used directly in giving an upper bound on the

maximum possible number of iterations, in the style of the previous paragraph.

Let us now establish that the set *S* returned at the termination of the

algorithm is in fact a perfect matching. Why is this not immediately obvious?

Essentially, we have to show that no man can “fall off” the end of his preference

list; the only way for the While loop to exit is for there to be no free man. In

this case, the set of engaged couples would indeed be a perfect matching.

So the main thing we need to show is the following.

(1.4) *If m is free at some point in the execution of the algorithm, then there*

*is a woman to whom he has not yet proposed.*

Proof. Suppose there comes a point when *m* is free but has already proposed

to every woman. Then by (1.1), each of the *n* women is engaged at this point

in time. Since the set of engaged pairs forms a matching, there must also be

*n* engaged men at this point in time. But there are only *n* men total, and *m* is

not engaged, so this is a contradiction.

(1.5) *The set S returned at termination is a perfect matching.*

Proof. The set of engaged pairs always forms a matching. Let us suppose that

the algorithm terminates with a free man *m*. At termination, it must be the

case that *m* had already proposed to every woman, for otherwise the While

loop would not have exited. But this contradicts (1.4), which says that there

cannot be a free man who has proposed to every woman.

Finally, we prove the main property of the algorithm—namely, that it

results in a stable matching.

(1.6) *Consider an execution of the G-S algorithm that returns a set of pairs*

*S. The set S is a stable matching.*

Proof. We have already seen, in (1.5), that *S* is a perfect matching. Thus, to

prove *S* is a stable matching, we will assume that there is an instability with

respect to *S* and obtain a contradiction. As defined earlier, such an instability

would involve two pairs, *(m*, *w)* and *(m*, *w**)*, in *S* with the properties that

. *m* prefers *w*to *w*, and

. *w*prefers *m* to *m*.

In the execution of the algorithm that produced *S*, *m*’s last proposal was, by

definition, to *w*. Now we ask: Did *m* propose to *w*at some earlier point in

1.1 A First Problem: Stable Matching **9**

this execution? If he didn’t, then *w* must occur higher on *m*’s preference list

than *w*, contradicting our assumption that *m* prefers *w*to *w*. If he did, then

he was rejected by *w*in favor of some other man *m*, whom *w*prefers to *m*.

*m*is the final partner of *w*, so either *m*= *m*or, by (1.1), *w*prefers her final

partner *m*to *m*; either way this contradicts our assumption that *w*prefers

*m* to *m*.

It follows that *S* is a stable matching.

**Extensions**

We began by defining the notion of a stable matching; we have just proven

that the G-S algorithm actually constructs one. We now consider some further

questions about the behavior of the G-S algorithm and its relation to the

properties of different stable matchings.

To begin with, recall that we saw an example earlier in which there could

be multiple stable matchings. To recap, the preference lists in this example

were as follows:

*m* prefers *w* to *w*.

*m*prefers *w*to *w*.

*w* prefers *m*to *m*.

*w*prefers *m* to *m*.

Now, in any execution of the Gale-Shapley algorithm, *m* will become engaged

to *w*, *m*will become engaged to *w*(perhaps in the other order), and things

will stop there. Thus, the *other* stable matching, consisting of the pairs *(m*, *w)*

and *(m*, *w**)*, is not attainable from an execution of the G-S algorithm in which

the men propose. On the other hand, it would be reached if we ran a version of

the algorithm in which the women propose. And in larger examples, with more

than two people on each side, we can have an even larger collection of possible

stable matchings, many of them not achievable by any natural algorithm.

This example shows a certain “unfairness” in the G-S algorithm, favoring

men. If the men’s preferences mesh perfectly (they all list different women as

their first choice), then in all runs of the G-S algorithm all men end up matched

with their first choice, independent of the preferences of the women. If the

women’s preferences clash completely with the men’s preferences (as was the

case in this example), then the resulting stable matching is as bad as possible

for the women. So this simple set of preference lists compactly summarizes a

world in which *someone* is destined to end up unhappy: women are unhappy

if men propose, and men are unhappy if women propose.

Let’s now analyze the G-S algorithm in more detail and try to understand

how general this “unfairness” phenomenon is.

**10** Chapter 1 Introduction: Some Representative Problems

To begin with, our example reinforces the point that the G-S algorithm

is actually underspecified: as long as there is a free man, we are allowed to

choose *any* free man to make the next proposal. Different choices specify

different executions of the algorithm; this is why, to be careful, we stated (1.6)

as “Consider an execution of the G-S algorithm that returns a set of pairs *S*,”

instead of “Consider the set *S* returned by the G-S algorithm.”

Thus, we encounter another very natural question: Do all executions of

the G-S algorithm yield the same matching? This is a genre of question that

arises in many settings in computer science: we have an algorithm that runs

*asynchronously*, with different independent components performing actions

that can be interleaved in complex ways, and we want to know how much

variability this asynchrony causes in the final outcome. To consider a very

different kind of example, the independent components may not be men and

women but electronic components activating parts of an airplane wing; the

effect of asynchrony in their behavior can be a big deal.

In the present context, we will see that the answer to our question is

surprisingly clean: all executions of the G-S algorithm yield the same matching.

We proceed to prove this now.

***All Executions Yield the Same Matching*** There are a number of possible

ways to prove a statement such as this, many of which would result in quite

complicated arguments. It turns out that the easiest and most informative ap

proach for us will be to uniquely *characterize* the matching that is obtained and

then show that all executions result in the matching with this characterization.

What is the characterization? We’ll show that each man ends up with the

“best possible partner” in a concrete sense. (Recall that this is true if all men

prefer different women.) First, we will say that a woman *w* is a *valid partner*

of a man *m* if there is a stable matching that contains the pair *(m*, *w)*. We will

say that *w* is the *best valid partner* of *m* if *w* is a valid partner of *m*, and no

woman whom *m* ranks higher than *w* is a valid partner of his. We will use

*best(m)* to denote the best valid partner of *m*.

Now, let *S*∗ denote the set of pairs {*(m*, *best(m))* : *m* ∈ *M*}. We will prove

the following fact.

(1.7) *Every execution of the G-S algorithm results in the set S*∗*.*

This statement is surprising at a number of levels. First of all, as defined,

there is no reason to believe that *S*∗ is a matching at all, let alone a stable

matching. After all, why couldn’t it happen that two men have the same best

valid partner? Second, the result shows that the G-S algorithm gives the best

possible outcome for every man simultaneously; there is no stable matching

in which any of the men could have hoped to do better. And finally, it answers

1.1 A First Problem: Stable Matching **11**

our question above by showing that the order of proposals in the G-S algorithm

has absolutely no effect on the final outcome.

Despite all this, the proof is not so difficult.

Proof. Let us suppose, by way of contradiction, that some execution E of the

G-S algorithm results in a matching *S* in which some man is paired with a

woman who is not his best valid partner. Since men propose in decreasing

order of preference, this means that some man is rejected by a valid partner

during the execution E of the algorithm. So consider the first moment during

the execution E in which some man, say *m*, is rejected by a valid partner *w*.

Again, since men propose in decreasing order of preference, and since this is

the first time such a rejection has occurred, it must be that *w* is *m*’s best valid

partner *best(m)*.

The rejection of *m* by *w* may have happened either because *m* proposed

and was turned down in favor of *w*’s existing engagement, or because *w* broke

her engagement to *m* in favor of a better proposal. But either way, at this

moment *w* forms or continues an engagement with a man *m*whom she prefers

to *m*.

Since *w* is a valid partner of *m*, there exists a stable matching *S*containing

the pair *(m*, *w)*. Now we ask: Who is *m*paired with in this matching? Suppose

it is a woman *w* = *w*.

Since the rejection of *m* by *w* was the first rejection of a man by a valid

partner in the execution E, it must be that *m*had not been rejected by any valid

partner at the point in E when he became engaged to *w*. Since he proposed in

decreasing order of preference, and since *w*is clearly a valid partner of *m*, it

must be that *m*prefers *w* to *w*. But we have already seen that *w* prefers *m*

to *m*, for in execution E she rejected *m* in favor of *m*. Since *(m*, *w)*  ∈ *S*, it

follows that *(m*, *w)* is an instability in *S*.

This contradicts our claim that *S*is stable and hence contradicts our initial

assumption.

So for the men, the G-S algorithm is ideal. Unfortunately, the same cannot

be said for the women. For a woman *w*, we say that *m* is a valid partner if

there is a stable matching that contains the pair *(m*, *w)*. We say that *m* is the

*worst valid partner* of *w* if *m* is a valid partner of *w*, and no man whom *w*

ranks lower than *m* is a valid partner of hers.

(1.8) *In the stable matching S*∗*, each woman is paired with her worst valid*

*partner.*

Proof. Suppose there were a pair *(m*, *w)* in *S*∗ such that *m* is not the worst

valid partner of *w*. Then there is a stable matching *S*in which *w* is paired

**12** Chapter 1 Introduction: Some Representative Problems

with a man *m*whom she likes less than *m*. In *S*, *m* is paired with a woman

*w* = *w*; since *w* is the best valid partner of *m*, and *w*is a valid partner of *m*,

we see that *m* prefers *w* to *w*.

But from this it follows that *(m*, *w)* is an instability in *S*, contradicting the

claim that *S*is stable and hence contradicting our initial assumption.

Thus, we find that our simple example above, in which the men’s pref

erences clashed with the women’s, hinted at a very general phenomenon: for

any input, the side that does the proposing in the G-S algorithm ends up with

the best possible stable matching (from their perspective), while the side that

does not do the proposing correspondingly ends up with the worst possible

stable matching.

1.2 Five Representative Problems

The Stable Matching Problem provides us with a rich example of the process of

algorithm design. For many problems, this process involves a few significant

steps: formulating the problem with enough mathematical precision that we

can ask a concrete question and start thinking about algorithms to solve

it; designing an algorithm for the problem; and analyzing the algorithm by

proving it is correct and giving a bound on the running time so as to establish

the algorithm’s efficiency.

This high-level strategy is carried out in practice with the help of a few

fundamental design techniques, which are very useful in assessing the inherent

complexity of a problem and in formulating an algorithm to solve it. As in any

area, becoming familiar with these design techniques is a gradual process; but

with experience one can start recognizing problems as belonging to identifiable

genres and appreciating how subtle changes in the statement of a problem can

have an enormous effect on its computational difficulty.

To get this discussion started, then, it helps to pick out a few representa

tive milestones that we’ll be encountering in our study of algorithms: cleanly

formulated problems, all resembling one another at a general level, but differ

ing greatly in their difficulty and in the kinds of approaches that one brings

to bear on them. The first three will be solvable efficiently by a sequence of

increasingly subtle algorithmic techniques; the fourth marks a major turning

point in our discussion, serving as an example of a problem believed to be un

solvable by any efficient algorithm; and the fifth hints at a class of problems

believed to be harder still.

The problems are self-contained and are all motivated by computing

applications. To talk about some of them, though, it will help to use the

terminology of *graphs*. While graphs are a common topic in earlier computer

1.2 Five Representative Problems **13**

science courses, we’ll be introducing them in a fair amount of depth in

Chapter 3; due to their enormous expressive power, we’ll also be using them

extensively throughout the book. For the discussion here, it’s enough to think

of a graph *G* as simply a way of encoding pairwise relationships among a set

of objects. Thus, *G* consists of a pair of sets *(V*, *E)*—a collection *V* of *nodes*

and a collection *E* of *edges*, each of which “joins” two of the nodes. We thus

represent an edge *e* ∈ *E* as a two-element subset of *V*: *e* = {*u*, *v*} for some *u*, *v* ∈ *V*, where we call *u* and *v* the *ends* of *e*. We typically draw graphs as in Figure 1.3, with each node as a small circle and each edge as a line segment joining its two ends.

Let’s now turn to a discussion of the five representative problems.

**Interval Scheduling**

Consider the following very simple scheduling problem. You have a resource— it may be a lecture room, a supercomputer, or an electron microscope—and many people request to use the resource for periods of time. A *request* takes the form: Can I reserve the resource starting at time *s*, until time *f*? We will assume that the resource can be used by at most one person at a time. A scheduler wants to accept a subset of these requests, rejecting all others, so that the accepted requests do not overlap in time. The goal is to maximize the number of requests accepted.

More formally, there will be *n* requests labeled 1, . . . , *n*, with each request *i* specifying a start time *si* and a finish time *fi*. Naturally, we have *si < fi* for all *i*. Two requests *i* and *j* are *compatible* if the requested intervals do not overlap: that is, either request *i* is for an earlier time interval than request *j* (*fi* ≤ *sj*), or request *i* is for a later time than request *j* (*fj* ≤ *si*). We’ll say more generally that a subset *A* of requests is compatible if all pairs of requests *i*, *j* ∈ *A*, *i*  = *j* are compatible. The goal is to select a compatible subset of requests of maximum possible size.

We illustrate an instance of this *Interval Scheduling Problem* in Figure 1.4. Note that there is a single compatible set of size 4, and this is the largest compatible set.

Figure 1.4 An instance of the Interval Scheduling Problem.

**(a)**

**(b)**

Figure 1.3 Each of (a) and (b) depicts a graph on four nodes.

**14** Chapter 1 Introduction: Some Representative Problems

We will see shortly that this problem can be solved by a very natural

algorithm that orders the set of requests according to a certain heuristic and

then “greedily” processes them in one pass, selecting as large a compatible

subset as it can. This will be typical of a class of *greedy algorithms* that we

will consider for various problems—myopic rules that process the input one

piece at a time with no apparent look-ahead. When a greedy algorithm can be

shown to find an optimal solution for all instances of a problem, it’s often fairly

surprising. We typically learn something about the structure of the underlying

problem from the fact that such a simple approach can be optimal.

**Weighted Interval Scheduling**

In the Interval Scheduling Problem, we sought to maximize the *number* of

requests that could be accommodated simultaneously. Now, suppose more

generally that each request interval *i* has an associated *value*, or *weight*,

*vi >* 0; we could picture this as the amount of money we will make from

the *i*th individual if we schedule his or her request. Our goal will be to find a

compatible subset of intervals of maximum total value.

The case in which *vi* = 1 for each *i* is simply the basic Interval Scheduling

Problem; but the appearance of arbitrary values changes the nature of the

maximization problem quite a bit. Consider, for example, that if *v*1 exceeds

the sum of all other *vi*, then the optimal solution must include interval 1

regardless of the configuration of the full set of intervals. So any algorithm

for this problem must be very sensitive to the values, and yet degenerate to a

method for solving (unweighted) interval scheduling when all the values are

equal to 1.

There appears to be no simple greedy rule that walks through the intervals

one at a time, making the correct decision in the presence of arbitrary values.

Instead, we employ a technique, *dynamic programming*, that builds up the

optimal value over all possible solutions in a compact, tabular way that leads

to a very efficient algorithm.

**Bipartite Matching**

When we considered the Stable Matching Problem, we defined a *matching* to

be a set of ordered pairs of men and women with the property that each man

and each woman belong to at most one of the ordered pairs. We then defined

a *perfect matching* to be a matching in which every man and every woman

belong to some pair.

We can express these concepts more generally in terms of graphs, and in

order to do this it is useful to define the notion of a *bipartite graph*. We say that

a graph *G* = *(V*, *E)* is *bipartite* if its node set *V* can be partitioned into sets *X*

1.2 Five Representative Problems **15**

and *Y* in such a way that every edge has one end in *X* and the other end in *Y*. A bipartite graph is pictured in Figure 1.5; often, when we want to emphasize a graph’s “bipartiteness,” we will draw it this way, with the nodes in *X* and *Y* in two parallel columns. But notice, for example, that the two graphs in Figure 1.3 are also bipartite.

Now, in the problem of finding a stable matching, matchings were built from pairs of men and women. In the case of bipartite graphs, the edges are pairs of nodes, so we say that a matching in a graph *G* = *(V*, *E)* is a set of edges *M* ⊆ *E* with the property that each node appears in at most one edge of *M*. *M* is a perfect matching if every node appears in exactly one edge of *M*.

To see that this does capture the same notion we encountered in the Stable Matching Problem, consider a bipartite graph *G*with a set *X* of *n* men, a set *Y* of *n* women, and an edge from every node in *X* to every node in *Y*. Then the matchings and perfect matchings in *G*are precisely the matchings and perfect matchings among the set of men and women.

In the Stable Matching Problem, we added preferences to this picture. Here, we do not consider preferences; but the nature of the problem in arbitrary bipartite graphs adds a different source of complexity: there is not necessarily an edge from every *x* ∈ *X* to every *y* ∈ *Y*, so the set of possible matchings has quite a complicated structure. In other words, it is as though only certain pairs of men and women are willing to be paired off, and we want to figure out how to pair off many people in a way that is consistent with this. Consider, for example, the bipartite graph *G* in Figure 1.5: there are many matchings in *G*, but there is only one perfect matching. (Do you see it?)

Matchings in bipartite graphs can model situations in which objects are being *assigned* to other objects. Thus, the nodes in *X* can represent jobs, the nodes in *Y* can represent machines, and an edge *(xi*, *yj)* can indicate that machine *yj* is capable of processing job *xi*. A perfect matching is then a way of assigning each job to a machine that can process it, with the property that each machine is assigned exactly one job. In the spring, computer science departments across the country are often seen pondering a bipartite graph in which *X* is the set of professors in the department, *Y* is the set of offered courses, and an edge (*xi*, *yj*) indicates that professor *xi* is capable of teaching course *yj*. A perfect matching in this graph consists of an assignment of each professor to a course that he or she can teach, in such a way that every course is covered.

Thus the *Bipartite Matching Problem* is the following: Given an arbitrary bipartite graph *G*, find a matching of maximum size. If |*X*|=|*Y*| = *n*, then there is a perfect matching if and only if the maximum matching has size *n*. We will find that the algorithmic techniques discussed earlier do not seem adequate

*x*1 *y*1

*x*2 *y*2

*x*3 *y*3

*x*4 *y*4

*x*5 *y*5

Figure 1.5 A bipartite graph.

**16** Chapter 1 Introduction: Some Representative Problems

for providing an efficient algorithm for this problem. There is, however, a very

elegant and efficient algorithm to find a maximum matching; it inductively

builds up larger and larger matchings, selectively backtracking along the way.

This process is called *augmentation*, and it forms the central component in a

large class of efficiently solvable problems called *network flow problems*.

1

3

6

2

4 5 7

**Independent Set**

Now let’s talk about an extremely general problem, which includes most of these earlier problems as special cases. Given a graph *G* = *(V*, *E)*, we say a set of nodes *S* ⊆ *V* is *independent* if no two nodes in *S* are joined by an edge. The *Independent Set Problem* is, then, the following: Given *G*, find an independent set that is as large as possible. For example, the maximum size of

Figure 1.6 A graph whose largest independent set has size 4.

an independent set in the graph in Figure 1.6 is four, achieved by the four-node independent set {1, 4, 5, 6}.

The Independent Set Problem encodes any situation in which you are trying to choose from among a collection of objects and there are pairwise *conflicts* among some of the objects. Say you have *n* friends, and some pairs of them don’t get along. How large a group of your friends can you invite to dinner if you don’t want any interpersonal tensions? This is simply the largest independent set in the graph whose nodes are your friends, with an edge between each conflicting pair.

Interval Scheduling and Bipartite Matching can both be encoded as special cases of the Independent Set Problem. For Interval Scheduling, define a graph *G* = *(V*, *E)* in which the nodes are the intervals and there is an edge between each pair of them that overlap; the independent sets in *G* are then just the compatible subsets of intervals. Encoding Bipartite Matching as a special case of Independent Set is a little trickier to see. Given a bipartite graph *G*= *(V*, *E**)*, the objects being chosen are edges, and the conflicts arise between two edges that share an end. (These, indeed, are the pairs of edges that cannot belong to a common matching.) So we define a graph *G* = *(V*, *E)* in which the node set *V* is equal to the edge set *E*of *G*. We define an edge between each pair of elements in *V* that correspond to edges of *G*with a common end. We can now check that the independent sets of *G* are precisely the matchings of *G*. While it is not complicated to check this, it takes a little concentration to deal with this type of “edges-to-nodes, nodes-to-edges” transformation.2

2 For those who are curious, we note that not every instance of the Independent Set Problem can arise in this way from Interval Scheduling or from Bipartite Matching; the full Independent Set Problem really is more general. The graph in Figure 1.3(a) cannot arise as the “conflict graph” in an instance of

1.2 Five Representative Problems **17**

Given the generality of the Independent Set Problem, an efficient algorithm

to solve it would be quite impressive. It would have to implicitly contain

algorithms for Interval Scheduling, Bipartite Matching, and a host of other

natural optimization problems.

The current status of Independent Set is this: no efficient algorithm is

known for the problem, and it is conjectured that no such algorithm exists.

The obvious brute-force algorithm would try all subsets of the nodes, checking

each to see if it is independent, and then recording the largest one encountered.

It is possible that this is close to the best we can do on this problem. We will

see later in the book that Independent Set is one of a large class of problems

that are termed *NP-complete*. No efficient algorithm is known for any of them;

but they are all *equivalent* in the sense that a solution to any one of them

would imply, in a precise sense, a solution to all of them.

Here’s a natural question: Is there anything good we can say about the

complexity of the Independent Set Problem? One positive thing is the following:

If we have a graph *G* on 1,000 nodes, and we want to convince you that it

contains an independent set *S* of size 100, then it’s quite easy. We simply

show you the graph *G*, circle the nodes of *S* in red, and let you check that

no two of them are joined by an edge. So there really seems to be a great

difference in difficulty between *checking* that something is a large independent

set and actually *finding* a large independent set. This may look like a very basic

observation—and it is—but it turns out to be crucial in understanding this class

of problems. Furthermore, as we’ll see next, it’s possible for a problem to be

so hard that there isn’t even an easy way to “check” solutions in this sense.

**Competitive Facility Location**

Finally, we come to our fifth problem, which is based on the following two

player game. Consider two large companies that operate cafe franchises across ´

the country—let’s call them JavaPlanet and Queequeg’s Coffee—and they are

currently competing for market share in a geographic area. First JavaPlanet

opens a franchise; then Queequeg’s Coffee opens a franchise; then JavaPlanet;

then Queequeg’s; and so on. Suppose they must deal with zoning regulations

that require no two franchises be located too close together, and each is trying

to make its locations as convenient as possible. Who will win?

Let’s make the rules of this “game” more concrete. The geographic region

in question is divided into *n* zones, labeled 1, 2, . . . , *n*. Each zone *i* has a

Interval Scheduling, and the graph in Figure 1.3(b) cannot arise as the “conflict graph” in an instance

of Bipartite Matching.

**18** Chapter 1 Introduction: Some Representative Problems

1~~0 1 5 15 5 1 5 1 15 10~~

Figure 1.7 An instance of the Competitive Facility Location Problem.

value *bi*, which is the revenue obtained by either of the companies if it opens

a franchise there. Finally, certain pairs of zones *(i*, *j)* are *adjacent*, and local

zoning laws prevent two adjacent zones from each containing a franchise,

regardless of which company owns them. (They also prevent two franchises

from being opened in the same zone.) We model these conflicts via a graph

*G* = *(V*, *E)*, where *V* is the set of zones, and *(i*, *j)* is an edge in *E* if the

zones *i* and *j* are adjacent. The zoning requirement then says that the full

set of franchises opened must form an independent set in *G*.

Thus our game consists of two players, *P*1 and *P*2, alternately selecting

nodes in *G*, with *P*1 moving first. At all times, the set of all selected nodes

must form an independent set in *G*. Suppose that player *P*2 has a target bound

*B*, and we want to know: is there a strategy for *P*2 so that no matter how *P*1

plays, *P*2 will be able to select a set of nodes with a total value of at least *B*?

We will call this an instance of the *Competitive Facility Location Problem*.

Consider, for example, the instance pictured in Figure 1.7, and suppose

that *P*2’s target bound is *B* = 20. Then *P*2 does have a winning strategy. On the

other hand, if *B* = 25, then *P*2 does not.

One can work this out by looking at the figure for a while; but it requires

some amount of case-checking of the form, “If *P*1 goes here, then *P*2 will go

there; but if *P*1 goes over there, then *P*2 will go here. . . . ” And this appears to

be intrinsic to the problem: not only is it computationally difficult to determine

whether *P*2 has a winning strategy; on a reasonably sized graph, it would even

be hard for us to *convince* you that *P*2 has a winning strategy. There does not

seem to be a short proof we could present; rather, we’d have to lead you on a

lengthy case-by-case analysis of the set of possible moves.

This is in contrast to the Independent Set Problem, where we believe that

finding a large solution is hard but checking a proposed large solution is easy.

This contrast can be formalized in the class of *PSPACE-complete problems*, of

which Competitive Facility Location is an example. PSPACE-complete prob

lems are believed to be strictly harder than NP-complete problems, and this

conjectured lack of short “proofs” for their solutions is one indication of this

greater hardness. The notion of PSPACE-completeness turns out to capture a

large collection of problems involving game-playing and planning; many of

these are fundamental issues in the area of artificial intelligence.

Solved Exercises **19**

Solved Exercises

**Solved Exercise 1**

Consider a town with *n* men and *n* women seeking to get married to one

another. Each man has a preference list that ranks all the women, and each

woman has a preference list that ranks all the men.

The set of all 2*n* people is divided into two categories: *good* people and

*bad* people. Suppose that for some number *k*, 1≤ *k* ≤ *n* − 1, there are *k* good

men and *k* good women; thus there are *n* − *k* bad men and *n* − *k* bad women.

Everyone would rather marry any good person than any bad person.

Formally, each preference list has the property that it ranks each good person

of the opposite gender higher than each bad person of the opposite gender: its

first *k* entries are the good people (of the opposite gender) in some order, and

its next *n* − *k* are the bad people (of the opposite gender) in some order.

Show that in every stable matching, every good man is married to a good

woman.

***Solution*** A natural way to get started thinking about this problem is to

assume the claim is false and try to work toward obtaining a contradiction.

What would it mean for the claim to be false? There would exist some stable

matching *M* in which a good man *m* was married to a bad woman *w*.

Now, let’s consider what the other pairs in *M* look like. There are *k* good

men and *k* good women. Could it be the case that every good woman is married

to a good man in this matching *M*? No: one of the good men (namely, *m*) is

already married to a bad woman, and that leaves only *k* − 1 other good men.

So even if all of them were married to good women, that would still leave some

good woman who is married to a bad man.

Let *w*be such a good woman, who is married to a bad man. It is now

easy to identify an instability in *M*: consider the pair *(m*, *w**)*. Each is good,

but is married to a bad partner. Thus, each of *m* and *w*prefers the other to

their current partner, and hence *(m*, *w**)* is an instability. This contradicts our

assumption that *M* is stable, and hence concludes the proof.

**Solved Exercise 2**

We can think about a generalization of the Stable Matching Problem in which

certain man-woman pairs are explicitly *forbidden*. In the case of employers and

applicants, we could imagine that certain applicants simply lack the necessary

qualifications or certifications, and so they cannot be employed at certain

companies, however desirable they may seem. Using the analogy to marriage

between men and women, we have a set *M* of *n* men, a set *W* of *n* women,

**20** Chapter 1 Introduction: Some Representative Problems

and a set *F* ⊆ *M* × *W* of pairs who are simply *not allowed* to get married. Each

man *m* ranks all the women *w* for which *(m*, *w)*  ∈ *F*, and each woman *w*ranks

all the men *m*for which *(m*, *w**)*  ∈ *F*.

In this more general setting, we say that a matching *S* is *stable* if it does

not exhibit any of the following types of instability.

(i) There are two pairs *(m*, *w)* and *(m*, *w**)* in *S* with the property that

*(m*, *w**)*  ∈ *F*, *m* prefers *w*to *w*, and *w*prefers *m* to *m*. *(The usual kind*

*of instability.)*

(ii) There is a pair *(m*, *w)* ∈ *S*, and a man *m*, so that *m*is not part of any

pair in the matching, *(m*, *w)*  ∈ *F*, and *w* prefers *m*to *m*. *(A single man*

*is more desirable and not forbidden.)*

(iii) There is a pair *(m*, *w)* ∈ *S*, and a woman *w*, so that *w*is not part of

any pair in the matching, *(m*, *w**)*  ∈ *F*, and *m* prefers *w*to *w*. *(A single*

*woman is more desirable and not forbidden.)*

(iv) There is a man *m* and a woman *w*, neither of whom is part of any pair

in the matching, so that *(m*, *w)*  ∈ *F*. *(There are two single people with*

*nothing preventing them from getting married to each other.)*

Note that under these more general definitions, a stable matching need not be

a perfect matching.

Now we can ask: For every set of preference lists and every set of forbidden

pairs, is there always a stable matching? Resolve this question by doing one of

the following two things: (a) give an algorithm that, for any set of preference

lists and forbidden pairs, produces a stable matching; or (b) give an example

of a set of preference lists and forbidden pairs for which there is no stable

matching.

***Solution*** The Gale-Shapley algorithm is remarkably robust to variations on

the Stable Matching Problem. So, if you’re faced with a new variation of the

problem and can’t find a counterexample to stability, it’s often a good idea to

check whether a direct adaptation of the G-S algorithm will in fact produce

stable matchings.

That turns out to be the case here. We will show that there is always a

stable matching, even in this more general model with forbidden pairs, and

we will do this by adapting the G-S algorithm. To do this, let’s consider why

the original G-S algorithm can’t be used directly. The difficulty, of course, is

that the G-S algorithm doesn’t know anything about forbidden pairs, and so

the condition in the While loop,

While there is a man *m* who is free and hasn’t proposed to

every woman*,*

Solved Exercises **21**

won’t work: we don’t want *m* to propose to a woman *w* for which the pair

*(m*, *w)* is forbidden.

Thus, let’s consider a variation of the G-S algorithm in which we make

only one change: we modify the While loop to say,

While there is a man *m* who is free and hasn’t proposed to

every woman *w* for which *(m*, *w)*  ∈ *F.*

Here is the algorithm in full.

Initially all *m* ∈ *M* and *w* ∈ *W* are free

While there is a man *m* who is free and hasn’t proposed to

every woman *w* for which *(m*, *w)*  ∈ *F*

Choose such a man *m*

Let *w* be the highest-ranked woman in *m*’s preference list

to which *m* has not yet proposed

If *w* is free then

*(m*, *w)* become engaged

Else *w* is currently engaged to *m*

If *w* prefers *m*to *m* then

*m* remains free

Else *w* prefers *m* to *m*

*(m*, *w)* become engaged

*m*becomes free

Endif

Endif

Endwhile

Return the set *S* of engaged pairs

We now prove that this yields a stable matching, under our new definition

of stability.

To begin with, facts (1.1), (1.2), and (1.3) from the text remain true (in

particular, the algorithm will terminate in at most *n*2 iterations). Also, we

don’t have to worry about establishing that the resulting matching *S* is perfect

(indeed, it may not be). We also notice an additional pairs of facts. If *m* is

a man who is not part of a pair in *S*, then *m* must have proposed to every

nonforbidden woman; and if *w* is a woman who is not part of a pair in *S*, then

it must be that no man ever proposed to *w*.

Finally, we need only show

(1.9) *There is no instability with respect to the returned matching S.*

**22** Chapter 1 Introduction: Some Representative Problems

Proof. Our general definition of instability has four parts: This means that we

have to make sure that none of the four bad things happens.

First, suppose there is an instability of type (i), consisting of pairs *(m*, *w)*

and *(m*, *w**)* in *S* with the property that *(m*, *w**)*  ∈ *F*, *m* prefers *w*to *w*, and *w*

prefers *m* to *m*. It follows that *m* must have proposed to *w*; so *w*rejected *m*,

and thus she prefers her final partner to *m*—a contradiction.

Next, suppose there is an instability of type (ii), consisting of a pair

*(m*, *w)* ∈ *S*, and a man *m*, so that *m*is not part of any pair in the matching,

*(m*, *w)*  ∈ *F*, and *w* prefers *m*to *m*. Then *m*must have proposed to *w* and

been rejected; again, it follows that *w* prefers her final partner to *m*—a

contradiction.

Third, suppose there is an instability of type (iii), consisting of a pair

*(m*, *w)* ∈ *S*, and a woman *w*, so that *w*is not part of any pair in the matching,

*(m*, *w**)*  ∈ *F*, and *m* prefers *w*to *w*. Then no man proposed to *w*at all;

in particular, *m* never proposed to *w*, and so he must prefer *w* to *w*—a

contradiction.

Finally, suppose there is an instability of type (iv), consisting of a man

*m* and a woman *w*, neither of which is part of any pair in the matching,

so that *(m*, *w)*  ∈ *F*. But for *m* to be single, he must have proposed to every

nonforbidden woman; in particular, he must have proposed to *w*, which means

she would no longer be single—a contradiction.

Exercises

1. Decide whether you think the following statement is true or false. If it is

true, give a short explanation. If it is false, give a counterexample.

*True or false? In every instance of the Stable Matching Problem, there is a*

*stable matching containing a pair (m*, *w) such that m is ranked first on the*

*preference list of w and w is ranked first on the preference list of m.*

2. Decide whether you think the following statement is true or false. If it is

true, give a short explanation. If it is false, give a counterexample.

*True or false? Consider an instance of the Stable Matching Problem in which*

*there exists a man m and a woman w such that m is ranked first on the*

*preference list of w and w is ranked first on the preference list of m. Then in*

*every stable matching S for this instance, the pair (m*, *w) belongs to S.*

3. There are many other settings in which we can ask questions related

to some type of “stability” principle. Here’s one, involving competition

between two enterprises.

Exercises **23**

Suppose we have two television networks, whom we’ll call A and B.

There are *n* prime-time programming slots, and each network has *n* TV

shows. Each network wants to devise a *schedule*—an assignment of each

show to a distinct slot—so as to attract as much market share as possible.

Here is the way we determine how well the two networks perform

relative to each other, given their schedules. Each show has a fixed *rating*,

which is based on the number of people who watched it last year; we’ll

assume that no two shows have exactly the same rating. A network *wins* a

given time slot if the show that it schedules for the time slot has a larger

rating than the show the other network schedules for that time slot. The

goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network A reveals a

schedule *S* and Network B reveals a schedule *T*. On the basis of this pair

of schedules, each network wins certain time slots, according to the rule

above. We’ll say that the pair of schedules *(S*, *T)*is *stable* if neither network

can unilaterally change its own schedule and win more time slots. That

is, there is no schedule *S*such that Network A wins more slots with the

pair *(S*, *T)* than it did with the pair *(S*, *T)*; and symmetrically, there is no

schedule *T*such that Network B wins more slots with the pair *(S*, *T**)* than

it did with the pair *(S*, *T)*.

The analogue of Gale and Shapley’s question for this kind of stability

is the following: For every set of TV shows and ratings, is there always

a stable pair of schedules? Resolve this question by doing one of the

following two things:

(a) give an algorithm that, for any set of TV shows and associated

ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for

which there is no stable pair of schedules.

4. Gale and Shapley published their paper on the Stable Matching Problem

in 1962; but a version of their algorithm had already been in use for

ten years by the National Resident Matching Program, for the problem of

assigning medical residents to hospitals.

Basically, the situation was the following. There were *m* hospitals,

each with a certain number of available positions for hiring residents.

There were *n* medical students graduating in a given year, each interested

in joining one of the hospitals. Each hospital had a ranking of the students

in order of preference, and each student had a ranking of the hospitals

in order of preference. We will assume that there were more students

graduating than there were slots available in the *m* hospitals.

**24** Chapter 1 Introduction: Some Representative Problems

The interest, naturally, was in finding a way of assigning each student

to at most one hospital, in such a way that all available positions in all

hospitals were filled. (Since we are assuming a surplus of students, there

would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is *stable* if neither

of the following situations arises.

. First type of instability: There are students *s* and *s*, and a hospital *h*,

so that

– *s* is assigned to *h*, and

– *s*is assigned to no hospital, and

– *h* prefers *s*to *s*.

. Second type of instability: There are students *s* and *s*, and hospitals

*h* and *h*, so that

– *s* is assigned to *h*, and

– *s*is assigned to *h*, and

– *h* prefers *s*to *s*, and

– *s*prefers *h* to *h*.

So we basically have the Stable Matching Problem, except that (i)

hospitals generally want more than one resident, and (ii) there is a surplus

of medical students.

Show that there is always a stable assignment of students to hospi

tals, and give an algorithm to find one.

5. The Stable Matching Problem, as discussed in the text, assumes that all

men and women have a fully ordered list of preferences. In this problem

we will consider a version of the problem in which men and women can be

*indifferent* between certain options. As before we have a set *M* of *n* men

and a set *W* of *n* women. Assume each man and each woman ranks the

members of the opposite gender, but now we allow ties in the ranking.

For example (with *n* = 4), a woman could say that *m*1 is ranked in first

place; second place is a tie between *m*2 and *m*3 (she has no preference

between them); and *m*4 is in last place. We will say that *w prefers m* to *m*

if *m* is ranked higher than *m*on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions

for stability. And for each, we can ask about the existence of stable

matchings, as follows.

(a) A *strong instability* in a perfect matching *S* consists of a man *m* and

a woman *w*, such that each of *m* and *w* prefers the other to their

partner in *S*. Does there always exist a perfect matching with no

Exercises **25**

strong instability? Either give an example of a set of men and women

with preference lists for which every perfect matching has a strong

instability; or give an algorithm that is guaranteed to find a perfect

matching with no strong instability.

(b) A *weak instability* in a perfect matching *S* consists of a man *m* and

a woman *w*, such that their partners in *S* are *w*and *m*, respectively,

and one of the following holds:

– *m* prefers *w* to *w*, and *w* either prefers *m* to *m*or is indifferent

between these two choices; or

– *w* prefers *m* to *m*, and *m* either prefers *w* to *w*or is indifferent

between these two choices.

In other words, the pairing between *m* and *w* is either preferred

by both, or preferred by one while the other is indifferent. Does

there always exist a perfect matching with no weak instability? Either

give an example of a set of men and women with preference lists

for which every perfect matching has a weak instability; or give an

algorithm that is guaranteed to find a perfect matching with no weak

instability.

6. Peripatetic Shipping Lines, Inc., is a shipping company that owns *n* ships

and provides service to *n* ports. Each of its ships has a *schedule* that says,

for each day of the month, which of the ports it’s currently visiting, or

whether it’s out at sea. (You can assume the “month” here has *m* days,

for some *m > n*.) Each ship visits each port for exactly one day during the

month. For safety reasons, PSL Inc. has the following strict requirement:

*(*†*) No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this

month, via the following scheme. They want to *truncate* each ship’s

schedule: for each ship *Si*, there will be some day when it arrives in its

scheduled port and simply remains there for the rest of the month (for

maintenance). This means that *Si* will not visit the remaining ports on

its schedule (if any) that month, but this is okay. So the *truncation* of

*Si*’s schedule will simply consist of its original schedule up to a certain

specified day on which it is in a port *P*; the remainder of the truncated

schedule simply has it remain in port *P*.

Now the company’s question to you is the following: Given the sched

ule for each ship, find a truncation of each so that condition (†) continues

to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an

algorithm to find them.

**26** Chapter 1 Introduction: Some Representative Problems

Example. Suppose we have two ships and two ports, and the “month” has

four days. Suppose the first ship’s schedule is

*port P*1*; at sea; port P*2*; at sea*

and the second ship’s schedule is

*at sea; port P*1*; at sea; port P*2

Then the (only) way to choose truncations would be to have the first ship

remain in port *P*2 starting on day 3, and have the second ship remain in

port *P*1 starting on day 2.

7. Some of your friends are working for CluNet, a builder of large commu

nication networks, and they are looking at algorithms for switching in a

particular type of input/output crossbar.

Here is the setup. There are *n input wires* and *n output wires*, each

directed from a *source* to a *terminus*. Each input wire meets each output

wire in exactly one distinct point, at a special piece of hardware called

a *junction box*. Points on the wire are naturally ordered in the direction

from source to terminus; for two distinct points *x* and *y* on the same

wire, we say that *x* is *upstream* from *y* if *x* is closer to the source than

*y*, and otherwise we say *x* is *downstream* from *y*. The order in which one

input wire meets the output wires is not necessarily the same as the order

in which another input wire meets the output wires. (And similarly for

the orders in which output wires meet input wires.) Figure 1.8 gives an

example of such a collection of input and output wires.

Now, here’s the switching component of this situation. Each input

wire is carrying a distinct data stream, and this data stream must be

*switched* onto one of the output wires. If the stream of Input *i* is switched

onto Output *j*, at junction box *B*, then this stream passes through all

junction boxes upstream from *B* on Input *i*, then through *B*, then through

all junction boxes downstream from *B* on Output *j*. It does not matter

which input data stream gets switched onto which output wire, but

each input data stream must be switched onto a *different* output wire.

Furthermore—and this is the tricky constraint—no two data streams can

pass through the same junction box following the switching operation.

Finally, here’s the problem. Show that for any specified pattern in

which the input wires and output wires meet each other (each pair meet

ing exactly once), a valid switching of the data streams can always be

found—one in which each input data stream is switched onto a different

output, and no two of the resulting streams pass through the same junc

tion box. Additionally, give an algorithm to find such a valid switching.

Junction

Exercises **27** Output 1

Junction

Junction

Junction

(meets Input 2 before Input 1)

Output 2

(meets Input 2 before Input 1)

Input 1

(meets Output 2 before Output 1)

Input 2

(meets Output 1 before Output 2)

Figure 1.8 An example with two input wires and two output wires. Input 1 has its junction with Output 2 upstream from its junction with Output 1; Input 2 has its junction with Output 1 upstream from its junction with Output 2. A valid solution is to switch the data stream of Input 1 onto Output 2, and the data stream of Input 2 onto Output 1. On the other hand, if the stream of Input 1 were switched onto Output 1, and the stream of Input 2 were switched onto Output 2, then both streams would pass through the junction box at the meeting of Input 1 and Output 2—and this is not allowed.

8. For this problem, we will explore the issue of *truthfulness* in the Stable Matching Problem and specifically in the Gale-Shapley algorithm. The basic question is: Can a man or a woman end up better off by lying about his or her preferences? More concretely, we suppose each participant has a true preference order. Now consider a woman *w*. Suppose *w* prefers man *m* to *m*, but both *m* and *m*are low on her list of preferences. Can it be the case that by switching the order of *m* and *m*on her list of preferences (i.e., by falsely claiming that she prefers *m*to *m*) and running the algorithm with this false preference list, *w* will end up with a man *m*that she truly prefers to both *m* and *m*? (We can ask the same question for men, but will focus on the case of women for purposes of this question.)

Resolve this question by doing one of the following two things:

(a) Give a proof that, for any set of preference lists, switching the order of a pair on the list cannot improve a woman’s partner in the Gale Shapley algorithm; or

**28** Chapter 1 Introduction: Some Representative Problems

(b) Give an example of a set of preference lists for which there is

a switch that would improve the partner of a woman who switched

preferences.

Notes and Further Reading

The Stable Matching Problem was first defined and analyzed by Gale and

Shapley (1962); according to David Gale, their motivation for the problem

came from a story they had recently read in the *New Yorker* about the intricacies

of the college admissions process (Gale, 2001). Stable matching has grown

into an area of study in its own right, covered in books by Gusfield and Irving

(1989) and Knuth (1997c). Gusfield and Irving also provide a nice survey of

the “parallel” history of the Stable Matching Problem as a technique invented

for matching applicants with employers in medicine and other professions.

As discussed in the chapter, our five representative problems will be

central to the book’s discussions, respectively, of greedy algorithms, dynamic

programming, network flow, NP-completeness, and PSPACE-completeness.

We will discuss the problems in these contexts later in the book.

***Chapter 2***

***Basics of Algorithm Analysis***

Analyzing algorithms involves thinking about how their resource require ments—the amount of time and space they use—will scale with increasing input size. We begin this chapter by talking about how to put this notion on a concrete footing, as making it concrete opens the door to a rich understanding of computational tractability. Having done this, we develop the mathematical machinery needed to talk about the way in which different functions scale with increasing input size, making precise what it means for one function to grow faster than another.

We then develop running-time bounds for some basic algorithms, begin ning with an implementation of the Gale-Shapley algorithm from Chapter 1 and continuing to a survey of many different running times and certain char acteristic types of algorithms that achieve these running times. In some cases, obtaining a good running-time bound relies on the use of more sophisticated data structures, and we conclude this chapter with a very useful example of such a data structure: priority queues and their implementation using heaps.

2.1 Computational Tractability

A major focus of this book is to find efficient algorithms for computational problems. At this level of generality, our topic seems to encompass the whole of computer science; so what is specific to our approach here?

First, we will try to identify broad themes and design principles in the development of algorithms. We will look for paradigmatic problems and ap proaches that illustrate, with a minimum of irrelevant detail, the basic ap proaches to designing efficient algorithms. At the same time, it would be pointless to pursue these design principles in a vacuum, so the problems and

**30** Chapter 2 Basics of Algorithm Analysis

approaches we consider are drawn from fundamental issues that arise through

out computer science, and a general study of algorithms turns out to serve as

a nice survey of computational ideas that arise in many areas.

Another property shared by many of the problems we study is their

fundamentally *discrete* nature. That is, like the Stable Matching Problem, they

will involve an implicit search over a large set of combinatorial possibilities;

and the goal will be to efficiently find a solution that satisfies certain clearly

delineated conditions.

As we seek to understand the general notion of computational efficiency,

we will focus primarily on efficiency in running time: we want algorithms that

run quickly. But it is important that algorithms be efficient in their use of other

resources as well. In particular, the amount of *space* (or memory) used by an

algorithm is an issue that will also arise at a number of points in the book, and

we will see techniques for reducing the amount of space needed to perform a

computation.

**Some Initial Attempts at Defining Efficiency**

The first major question we need to answer is the following: How should we

turn the fuzzy notion of an “efficient” algorithm into something more concrete?

A first attempt at a working definition of *efficiency* is the following.

Proposed Definition of Efficiency (1): *An algorithm is efficient if, when*

*implemented, it runs quickly on real input instances.*

Let’s spend a little time considering this definition. At a certain level, it’s hard

to argue with: one of the goals at the bedrock of our study of algorithms is

solving real problems quickly. And indeed, there is a significant area of research

devoted to the careful implementation and profiling of different algorithms for

discrete computational problems.

But there are some crucial things missing from this definition, even if our

main goal is to solve real problem instances quickly on real computers. The

first is the omission of *where*, and *how well*, we implement an algorithm. Even

bad algorithms can run quickly when applied to small test cases on extremely

fast processors; even good algorithms can run slowly when they are coded

sloppily. Also, what is a “real” input instance? We don’t know the full range of

input instances that will be encountered in practice, and some input instances

can be much harder than others. Finally, this proposed definition above does

not consider how well, or badly, an algorithm may *scale* as problem sizes grow

to unexpected levels. A common situation is that two very different algorithms

will perform comparably on inputs of size 100; multiply the input size tenfold,

and one will still run quickly while the other consumes a huge amount of time.

2.1 Computational Tractability **31**

So what we could ask for is a concrete definition of efficiency that is

platform-independent, instance-independent, and of predictive value with

respect to increasing input sizes. Before focusing on any specific consequences

of this claim, we can at least explore its implicit, high-level suggestion: that

we need to take a more mathematical view of the situation.

We can use the Stable Matching Problem as an example to guide us. The

input has a natural “size” parameter *N*; we could take this to be the total size of

the representation of all preference lists, since this is what any algorithm for the

problem will receive as input. *N* is closely related to the other natural parameter

in this problem: *n*, the number of men and the number of women. Since there

are 2*n* preference lists, each of length *n*, we can view *N* = 2*n*2, suppressing

more fine-grained details of how the data is represented. In considering the

problem, we will seek to describe an algorithm at a high level, and then analyze

its running time mathematically as a function of this input size *N*.

**Worst-Case Running Times and Brute-Force Search**

To begin with, we will focus on analyzing the *worst-case* running time: we will

look for a bound on the largest possible running time the algorithm could have

over all inputs of a given size *N*, and see how this scales with *N*. The focus on

worst-case performance initially seems quite draconian: what if an algorithm

performs well on most instances and just has a few pathological inputs on

which it is very slow? This certainly is an issue in some cases, but in general

the worst-case analysis of an algorithm has been found to do a reasonable job

of capturing its efficiency in practice. Moreover, once we have decided to go

the route of mathematical analysis, it is hard to find an effective alternative to

worst-case analysis. Average-case analysis—the obvious appealing alternative,

in which one studies the performance of an algorithm averaged over “random”

instances—can sometimes provide considerable insight, but very often it can

also become a quagmire. As we observed earlier, it’s very hard to express the

full range of input instances that arise in practice, and so attempts to study an

algorithm’s performance on “random” input instances can quickly devolve into

debates over how a random input should be generated: the same algorithm

can perform very well on one class of random inputs and very poorly on

another. After all, real inputs to an algorithm are generally not being produced

from a random distribution, and so average-case analysis risks telling us more

about the means by which the random inputs were generated than about the

algorithm itself.

So in general we will think about the worst-case analysis of an algorithm’s

running time. But what is a reasonable analytical benchmark that can tell us

whether a running-time bound is impressive or weak? A first simple guide

**32** Chapter 2 Basics of Algorithm Analysis

is by comparison with brute-force search over the search space of possible

solutions.

Let’s return to the example of the Stable Matching Problem. Even when

the size of a Stable Matching input instance is relatively small, the *search*

*space* it defines is enormous (there are *n*! possible perfect matchings between

*n* men and *n* women), and we need to find a matching that is stable. The

natural “brute-force” algorithm for this problem would plow through all perfect

matchings by enumeration, checking each to see if it is stable. The surprising

punchline, in a sense, to our solution of the Stable Matching Problem is that we

needed to spend time proportional only to *N* in finding a stable matching from

among this stupendously large space of possibilities. This was a conclusion we

reached at an *analytical level*. We did not implement the algorithm and try it

out on sample preference lists; we reasoned about it mathematically. Yet, at the

same time, our analysis indicated how the algorithm could be implemented in

practice and gave fairly conclusive evidence that it would be a big improvement

over exhaustive enumeration.

This will be a common theme in most of the problems we study: a compact

representation, implicitly specifying a giant search space. For most of these

problems, there will be an obvious brute-force solution: try all possibilities

and see if any one of them works. Not only is this approach almost always too

slow to be useful, it is an intellectual cop-out; it provides us with absolutely

no insight into the structure of the problem we are studying. And so if there

is a common thread in the algorithms we emphasize in this book, it would be

the following alternative definition of efficiency.

Proposed Definition of Efficiency (2): *An algorithm is efficient if it achieves*

*qualitatively better worst-case performance, at an analytical level, than*

*brute-force search.*

This will turn out to be a very useful working definition for us. Algorithms

that improve substantially on brute-force search nearly always contain a

valuable heuristic idea that makes them work; and they tell us something

about the intrinsic structure, and computational tractability, of the underlying

problem itself.

But if there is a problem with our second working definition, it is vague

ness. What do we mean by “qualitatively better performance?” This suggests

that we consider the actual running time of algorithms more carefully, and try

to quantify what a reasonable running time would be.

**Polynomial Time as a Definition of Efficiency**

When people first began analyzing discrete algorithms mathematically—a

thread of research that began gathering momentum through the 1960s—

2.1 Computational Tractability **33**

a consensus began to emerge on how to quantify the notion of a “reasonable”

running time. Search spaces for natural combinatorial problems tend to grow

exponentially in the size *N* of the input; if the input size increases by one, the

number of possibilities increases multiplicatively. We’d like a good algorithm

for such a problem to have a better scaling property: when the input size

increases by a constant factor—say, a factor of 2—the algorithm should only

slow down by some constant factor *C*.

Arithmetically, we can formulate this scaling behavior as follows. Suppose

an algorithm has the following property: There are absolute constants *c >* 0

and *d >* 0 so that on every input instance of size *N*, its running time is

bounded by *cNd* primitive computational steps. (In other words, its running

time is at most proportional to *Nd*.) For now, we will remain deliberately

vague on what we mean by the notion of a “primitive computational step”—

but it can be easily formalized in a model where each step corresponds to

a single assembly-language instruction on a standard processor, or one line

of a standard programming language such as C or Java. In any case, if this

running-time bound holds, for some *c* and *d*, then we say that the algorithm

has a *polynomial running time*, or that it is a *polynomial-time algorithm*. Note

that any polynomial-time bound has the scaling property we’re looking for. If

the input size increases from *N* to 2*N*, the bound on the running time increases

from *cNd* to *c(*2*N)d* = *c* · 2*dNd*, which is a slow-down by a factor of 2*d*. Since *d* is

a constant, so is 2*d*; of course, as one might expect, lower-degree polynomials

exhibit better scaling behavior than higher-degree polynomials.

From this notion, and the intuition expressed above, emerges our third

attempt at a working definition of efficiency.

Proposed Definition of Efficiency (3): *An algorithm is efficient if it has a*

*polynomial running time.*

Where our previous definition seemed overly vague, this one seems much

too prescriptive. Wouldn’t an algorithm with running time proportional to

*n*100—and hence polynomial—be hopelessly inefficient? Wouldn’t we be rel

atively pleased with a nonpolynomial running time of *n*1+.02*(*log *n)*? The an

swers are, of course, “yes” and “yes.” And indeed, however much one may

try to abstractly motivate the definition of efficiency in terms of polynomial

time, a primary justification for it is this: *It really works.* Problems for which

polynomial-time algorithms exist almost invariably turn out to have algorithms

with running times proportional to very moderately growing polynomials like

*n*, *n* log *n*, *n*2, or *n*3. Conversely, problems for which no polynomial-time al

gorithm is known tend to be very difficult in practice. There are certainly

exceptions to this principle in both directions: there are cases, for example, in

**34** Chapter 2 Basics of Algorithm Analysis

Table 2.1 The running times (rounded up) of different algorithms on inputs of

increasing size, for a processor performing a million high-level instructions per second.

In cases where the running time exceeds 1025 years, we simply record the algorithm as

taking a very long time.

*n n* log2 *n n*2 *n*3 1.5*n* 2*n n*! *n* = 10 *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec 4 sec *n* = 30 *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec 18 min 1025 years *n* = 50 *<* 1 sec *<* 1 sec *<* 1 sec *<* 1 sec 11 min 36 years very long *n* = 100 *<* 1 sec *<* 1 sec *<* 1 sec 1 sec 12,892 years 1017 years very long

*n* = 1,000 *<* 1 sec *<* 1 sec 1 sec 18 min very long very long very long *n* = 10,000 *<* 1 sec *<* 1 sec 2 min 12 days very long very long very long *n* = 100,000 *<* 1 sec 2 sec 3 hours 32 years very long very long very long *n* = 1,000,000 1 sec 20 sec 12 days 31,710 years very long very long very long

which an algorithm with exponential worst-case behavior generally runs well

on the kinds of instances that arise in practice; and there are also cases where

the best polynomial-time algorithm for a problem is completely impractical

due to large constants or a high exponent on the polynomial bound. All this

serves to reinforce the point that our emphasis on worst-case, polynomial-time

bounds is only an abstraction of practical situations. But overwhelmingly, the

concrete mathematical definition of polynomial time has turned out to corre

spond surprisingly well in practice to what we observe about the efficiency of

algorithms, and the tractability of problems, in real life.

One further reason why the mathematical formalism and the empirical

evidence seem to line up well in the case of polynomial-time solvability is that

the gulf between the growth rates of polynomial and exponential functions

is enormous. Suppose, for example, that we have a processor that executes

a million high-level instructions per second, and we have algorithms with

running-time bounds of *n*, *n* log2 *n*, *n*2, *n*3, 1.5*n*, 2*n*, and *n*!. In Table 2.1,

we show the running times of these algorithms (in seconds, minutes, days,

or years) for inputs of size *n* = 10, 30, 50, 100, 1,000, 10,000, 100,000, and

1,000,000.

There is a final, fundamental benefit to making our definition of efficiency

so specific: it becomes negatable. It becomes possible to express the notion

that *there is no efficient algorithm for a particular problem*. In a sense, being

able to do this is a prerequisite for turning our study of algorithms into

good science, for it allows us to ask about the existence or nonexistence

of efficient algorithms as a well-defined question. In contrast, both of our

2.2 Asymptotic Order of Growth **35**

previous definitions were completely subjective, and hence limited the extent

to which we could discuss certain issues in concrete terms.

In particular, the first of our definitions, which was tied to the specific

implementation of an algorithm, turned efficiency into a moving target: as

processor speeds increase, more and more algorithms fall under this notion of

efficiency. Our definition in terms of polynomial time is much more an absolute

notion; it is closely connected with the idea that each problem has an intrinsic

level of computational tractability: some admit efficient solutions, and others

do not.

2.2 Asymptotic Order of Growth

Our discussion of computational tractability has turned out to be intrinsically

based on our ability to express the notion that an algorithm’s worst-case

running time on inputs of size *n* grows at a rate that is at most proportional to

some function *f(n)*. The function *f(n)* then becomes a bound on the running

time of the algorithm. We now discuss a framework for talking about this

concept.

We will mainly express algorithms in the pseudo-code style that we used

for the Gale-Shapley algorithm. At times we will need to become more formal,

but this style of specifying algorithms will be completely adequate for most

purposes. When we provide a bound on the running time of an algorithm,

we will generally be counting the number of such pseudo-code steps that

are executed; in this context, one *step* will consist of assigning a value to a

variable, looking up an entry in an array, following a pointer, or performing

an arithmetic operation on a fixed-size integer.

When we seek to say something about the running time of an algorithm on

inputs of size *n*, one thing we could aim for would be a very concrete statement

such as, “On any input of size *n*, the algorithm runs for at most 1.62*n*2 +

3.5*n* + 8 steps.” This may be an interesting statement in some contexts, but as

a general goal there are several things wrong with it. First, getting such a precise

bound may be an exhausting activity, and more detail than we wanted anyway.

Second, because our ultimate goal is to identify broad classes of algorithms that

have similar behavior, we’d actually like to classify running times at a coarser

level of granularity so that similarities among different algorithms, and among

different problems, show up more clearly. And finally, extremely detailed

statements about the number of steps an algorithm executes are often—in

a strong sense—meaningless. As just discussed, we will generally be counting

steps in a pseudo-code specification of an algorithm that resembles a high

level programming language. Each one of these steps will typically unfold

into some fixed number of primitive steps when the program is compiled into

**36** Chapter 2 Basics of Algorithm Analysis

an intermediate representation, and then into some further number of steps

depending on the particular architecture being used to do the computing. So

the most we can safely say is that as we look at different levels of computational

abstraction, the notion of a “step” may grow or shrink by a constant factor—

for example, if it takes 25 low-level machine instructions to perform one

operation in our high-level language, then our algorithm that took at most

1.62*n*2 + 3.5*n* + 8 steps can also be viewed as taking 40.5*n*2 + 87.5*n* + 200 steps

when we analyze it at a level that is closer to the actual hardware.

*O***,** **, and**

For all these reasons, we want to express the growth rate of running times

and other functions in a way that is insensitive to constant factors and low

order terms. In other words, we’d like to be able to take a running time like

the one we discussed above, 1.62*n*2 + 3.5*n* + 8, and say that it grows like *n*2,

up to constant factors. We now discuss a precise way to do this.

***Asymptotic Upper Bounds*** Let *T(n)* be a function—say, the worst-case run

ning time of a certain algorithm on an input of size *n*. (We will assume that

all the functions we talk about here take nonnegative values.) Given another

function *f(n)*, we say that *T(n) is O(f(n))* (read as “*T(n)* is order *f(n)*”) if, for

sufficiently large *n*, the function *T(n)* is bounded above by a constant multiple

of *f(n)*. We will also sometimes write this as *T(n)* = *O(f(n))*. More precisely,

*T(n)* is *O(f(n))* if there exist constants *c >* 0 and *n*0 ≥ 0 so that for all *n* ≥ *n*0,

we have *T(n)* ≤ *c* · *f(n)*. In this case, we will say that *T* is *asymptotically upper*

*bounded by f*. It is important to note that this definition requires a constant *c*

to exist that works for *all n*; in particular, *c* cannot depend on *n*.

As an example of how this definition lets us express upper bounds on

running times, consider an algorithm whose running time (as in the earlier

discussion) has the form *T(n)* = *pn*2 + *qn* + *r* for positive constants *p*, *q*, and

*r*. We’d like to claim that any such function is *O(n*2*)*. To see why, we notice

that for all *n* ≥ 1, we have *qn* ≤ *qn*2, and *r* ≤ *rn*2. So we can write

*T(n)* = *pn*2 + *qn* + *r* ≤ *pn*2 + *qn*2 + *rn*2 = *(p* + *q* + *r)n*2

for all *n* ≥ 1. This inequality is exactly what the definition of *O(*·*)* requires:

*T(n)* ≤ *cn*2, where *c* = *p* + *q* + *r*.

Note that *O(*·*)* expresses only an upper bound, not the exact growth rate

of the function. For example, just as we claimed that the function *T(n)* =

*pn*2 + *qn* + *r* is *O(n*2*)*, it’s also correct to say that it’s *O(n*3*)*. Indeed, we just

argued that *T(n)* ≤ *(p* + *q* + *r)n*2, and since we also have *n*2 ≤ *n*3, we can

conclude that *T(n)* ≤ *(p* + *q* + *r)n*3 as the definition of *O(n*3*)* requires. The

fact that a function can have many upper bounds is not just a trick of the

notation; it shows up in the analysis of running times as well. There are cases

2.2 Asymptotic Order of Growth **37**

where an algorithm has been proved to have running time *O(n*3*)*; some years

pass, people analyze the same algorithm more carefully, and they show that

in fact its running time is *O(n*2*)*. There was nothing wrong with the first result;

it was a correct upper bound. It’s simply that it wasn’t the “tightest” possible

running time.

***Asymptotic Lower Bounds*** There is a complementary notation for lower

bounds. Often when we analyze an algorithm—say we have just proven that

its worst-case running time *T(n)* is *O(n*2*)*—we want to show that this upper

bound is the best one possible. To do this, we want to express the notion that for

arbitrarily large input sizes *n*, the function *T(n)*is *at least* a constant multiple of

some specific function *f(n)*. (In this example, *f(n)* happens to be *n*2.) Thus, we

say that *T(n)* is  *(f(n))* (also written *T(n)* =  *(f(n)))* if there exist constants

*>* 0 and *n*0 ≥ 0 so that for all *n* ≥ *n*0, we have *T(n)* ≥ · *f(n)*. By analogy with

*O(*·*)* notation, we will refer to *T* in this case as being *asymptotically lower*

*bounded by f*. Again, note that the constant must be fixed, independent

of *n*.

This definition works just like *O(*·*)*, except that we are bounding the

function *T(n)* from below, rather than from above. For example, returning

to the function *T(n)* = *pn*2 + *qn* + *r*, where *p*, *q*, and *r* are positive constants,

let’s claim that *T(n)* =  *(n*2*)*. Whereas establishing the upper bound involved

“inflating” the terms in *T(n)* until it looked like a constant times *n*2, now we

need to do the opposite: we need to reduce the size of *T(n)* until it looks like

a constant times *n*2. It is not hard to do this; for all *n* ≥ 0, we have

*T(n)* = *pn*2 + *qn* + *r* ≥ *pn*2,

which meets what is required by the definition of  *(*·*)* with = *p >* 0.

Just as we discussed the notion of “tighter” and “weaker” upper bounds,

the same issue arises for lower bounds. For example, it is correct to say that

our function *T(n)* = *pn*2 + *qn* + *r* is  *(n)*, since *T(n)* ≥ *pn*2 ≥ *pn*.

***Asymptotically Tight Bounds*** If we can show that a running time *T(n)* is

both *O(f(n))* and also  *(f(n))*, then in a natural sense we’ve found the “right”

bound: *T(n)* grows exactly like *f(n)* to within a constant factor. This, for

example, is the conclusion we can draw from the fact that *T(n)* = *pn*2 + *qn* + *r*

is both *O(n*2*)* and  *(n*2*)*.

There is a notation to express this: if a function *T(n)* is both *O(f(n))* and

*(f(n))*, we say that *T(n)* is  *(f(n))*. In this case, we say that *f(n)* is an

*asymptotically tight bound* for *T(n)*. So, for example, our analysis above shows

that *T(n)* = *pn*2 + *qn* + *r* is  *(n*2*)*.

Asymptotically tight bounds on worst-case running times are nice things

to find, since they characterize the worst-case performance of an algorithm

**38** Chapter 2 Basics of Algorithm Analysis

precisely up to constant factors. And as the definition of  *(*·*)* shows, one can

obtain such bounds by closing the gap between an upper bound and a lower

bound. For example, sometimes you will read a (slightly informally phrased)

sentence such as “An upper bound of *O(n*3*)* has been shown on the worst-case

running time of the algorithm, but there is no example known on which the

algorithm runs for more than  *(n*2*)* steps.” This is implicitly an invitation to

search for an asymptotically tight bound on the algorithm’s worst-case running

time.

Sometimes one can also obtain an asymptotically tight bound directly by

computing a limit as *n* goes to infinity. Essentially, if the ratio of functions

*f(n)* and *g(n)* converges to a positive constant as *n* goes to infinity, then

*f(n)* =  *(g(n))*.

(2.1) *Let f and g be two functions that*

lim*n*→∞*f(n)*

*g(n)*

*exists and is equal to some number c >* 0*. Then f(n)* =  *(g(n)).*

Proof. We will use the fact that the limit exists and is positive to show that

*f(n)* = *O(g(n))* and *f(n)* =  *(g(n))*, as required by the definition of  *(*·*)*.

Since

lim*n*→∞*f(n)*

*g(n)*= *c >* 0,

it follows from the definition of a limit that there is some *n*0 beyond which the

ratio is always between 1~~2~~*c* and 2*c*. Thus, *f(n)* ≤ 2*cg(n)* for all *n* ≥ *n*0, which

implies that *f(n)* = *O(g(n))*; and *f(n)* ≥ 1~~2~~*cg(n)* for all *n* ≥ *n*0, which implies

that *f(n)* =  *(g(n))*.

**Properties of Asymptotic Growth Rates**

Having seen the definitions of *O*, , and , it is useful to explore some of their

basic properties.

***Transitivity*** A first property is *transitivity*: if a function *f* is asymptotically

upper-bounded by a function *g*, and if *g* in turn is asymptotically upper

bounded by a function *h*, then *f* is asymptotically upper-bounded by *h*. A

similar property holds for lower bounds. We write this more precisely as

follows.

(2.2)

*(a) If f* = *O(g) and g* = *O(h), then f* = *O(h).*

*(b) If f* =  *(g) and g* =  *(h), then f* =  *(h).*

2.2 Asymptotic Order of Growth **39**

Proof. We’ll prove part (a) of this claim; the proof of part (b) is very similar.

For (a), we’re given that for some constants *c* and *n*0, we have *f(n)* ≤ *cg(n)*

for all *n* ≥ *n*0. Also, for some (potentially different) constants *c*and *n*0, we

have *g(n)* ≤ *c**h(n)* for all *n* ≥ *n*0. So consider any number *n* that is at least as

large as both *n*0 and *n*0. We have *f(n)* ≤ *cg(n)* ≤ *cc**h(n)*, and so *f(n)* ≤ *cc**h(n)*

for all *n* ≥ max*(n*0, *n*0*)*. This latter inequality is exactly what is required for

showing that *f* = *O(h)*.

Combining parts (a) and (b) of (2.2), we can obtain a similar result

for asymptotically tight bounds. Suppose we know that *f* =  *(g)* and that

*g* =  *(h)*. Then since *f* = *O(g)* and *g* = *O(h)*, we know from part (a) that

*f* = *O(h)*; since *f* =  *(g)* and *g* =  *(h)*, we know from part (b) that *f* =  *(h)*.

It follows that *f* =  *(h)*. Thus we have shown

(2.3) *If f* =  *(g) and g* =  *(h), then f* =  *(h).*

***Sums of Functions*** It is also useful to have results that quantify the effect of

adding two functions. First, if we have an asymptotic upper bound that applies

to each of two functions *f* and *g*, then it applies to their sum.

(2.4) *Suppose that f and g are two functions such that for some other function*

*h, we have f* = *O(h) and g* = *O(h). Then f* + *g* = *O(h).*

Proof. We’re given that for some constants *c* and *n*0, we have *f(n)* ≤ *ch(n)*

for all *n* ≥ *n*0. Also, for some (potentially different) constants *c*and *n*0,

we have *g(n)* ≤ *c**h(n)* for all *n* ≥ *n*0. So consider any number *n* that is at

least as large as both *n*0 and *n*0. We have *f(n)* + *g(n)* ≤ *ch(n)* + *c**h(n)*. Thus

*f(n)* + *g(n)* ≤ *(c* + *c**)h(n)* for all *n* ≥ max*(n*0, *n*0*)*, which is exactly what is

required for showing that *f* + *g* = *O(h)*.

There is a generalization of this to sums of a fixed constant number of

functions *k*, where *k* may be larger than two. The result can be stated precisely

as follows; we omit the proof, since it is essentially the same as the proof of

(2.4), adapted to sums consisting of *k* terms rather than just two.

(2.5) *Let k be a fixed constant, and let f*1, *f*2,..., *fk and h be functions such*

*that fi* = *O(h) for all i. Then f*1 + *f*2 + ... + *fk* = *O(h).*

There is also a consequence of (2.4) that covers the following kind of

situation. It frequently happens that we’re analyzing an algorithm with two

high-level parts, and it is easy to show that one of the two parts is slower

than the other. We’d like to be able to say that the running time of the whole

algorithm is asymptotically comparable to the running time of the slow part.

Since the overall running time is a sum of two functions (the running times of

**40** Chapter 2 Basics of Algorithm Analysis

the two parts), results on asymptotic bounds for sums of functions are directly

relevant.

(2.6) *Suppose that f and g are two functions (taking nonnegative values)*

*such that g* = *O(f). Then f* + *g* =  *(f). In other words, f is an asymptotically*

*tight bound for the combined function f* + *g.*

Proof. Clearly *f* + *g* =  *(f)*, since for all *n* ≥ 0, we have *f(n)* + *g(n)* ≥ *f(n)*.

So to complete the proof, we need to show that *f* + *g* = *O(f)*.

But this is a direct consequence of (2.4): we’re given the fact that *g* = *O(f)*,

and also *f* = *O(f)* holds for any function, so by (2.4) we have *f* + *g* = *O(f)*.

This result also extends to the sum of any fixed, constant number of

functions: the most rapidly growing among the functions is an asymptotically

tight bound for the sum.

**Asymptotic Bounds for Some Common Functions**

There are a number of functions that come up repeatedly in the analysis of

algorithms, and it is useful to consider the asymptotic properties of some of

the most basic of these: polynomials, logarithms, and exponentials.

***Polynomials*** Recall that a polynomial is a function that can be written in

the form *f(n)* = *a*0 + *a*1*n* + *a*2*n*2 + ... + *adnd* for some integer constant *d >* 0,

where the final coefficient *ad* is nonzero. This value *d* is called the *degree* of the

polynomial. For example, the functions of the form *pn*2 + *qn* + *r* (with *p*  = 0)

that we considered earlier are polynomials of degree 2.

A basic fact about polynomials is that their asymptotic rate of growth is

determined by their “high-order term”—the one that determines the degree.

We state this more formally in the following claim. Since we are concerned here

only with functions that take nonnegative values, we will restrict our attention

to polynomials for which the high-order term has a positive coefficient *ad >* 0.

(2.7) *Let f be a polynomial of degree d, in which the coefficient ad is positive.*

*Then f* = *O(nd).*

Proof. We write *f* = *a*0 + *a*1*n* + *a*2*n*2 + ... + *adnd*, where *ad >* 0. The upper

bound is a direct application of (2.5). First, notice that coefficients *aj* for *j < d*

may be negative, but in any case we have *ajnj* ≤ |*aj*|*nd* for all *n* ≥ 1. Thus each

term in the polynomial is *O(nd)*. Since *f* is a sum of a constant number of

functions, each of which is *O(nd)*, it follows from (2.5) that *f* is *O(nd)*.

One can also show that under the conditions of (2.7), we have *f* =  *(nd)*,

and hence it follows that in fact *f* =  *(nd)*.

2.2 Asymptotic Order of Growth **41**

This is a good point at which to discuss the relationship between these

types of asymptotic bounds and the notion of *polynomial time*, which we

arrived at in the previous section as a way to formalize the more elusive concept

of efficiency. Using *O(*·*)* notation, it’s easy to formally define polynomial time:

a *polynomial-time algorithm* is one whose running time *T(n)* is *O(nd)* for some

constant *d*, where *d* is independent of the input size.

So algorithms with running-time bounds like *O(n*2*)* and *O(n*3*)* are

polynomial-time algorithms. But it’s important to realize that an algorithm

can be polynomial time even if its running time is not written as *n* raised

to some integer power. To begin with, a number of algorithms have running

times of the form *O(nx)* for some number *x* that is not an integer. For example,

in Chapter 5 we will see an algorithm whose running time is *O(n*1.59*)*; we will

also see exponents less than 1, as in bounds like *O(*√*~~n)~~* = *O(n*1*/*2*)*.

To take another common kind of example, we will see many algorithms

whose running times have the form *O(n* log *n)*. Such algorithms are also

polynomial time: as we will see next, log *n* ≤ *n* for all *n* ≥ 1, and hence

*n* log *n* ≤ *n*2 for all *n* ≥ 1. In other words, if an algorithm has running time

*O(n* log *n)*, then it also has running time *O(n*2*)*, and so it is a polynomial-time

algorithm.

***Logarithms*** Recall that log*b n* is the number *x* such that *bx* = *n*. One way

to get an approximate sense of how fast log*b n* grows is to note that, if we

round it down to the nearest integer, it is one less than the number of digits

in the base-*b* representation of the number *n*. (Thus, for example, 1+ log2 *n*,

rounded down, is the number of bits needed to represent *n*.)

So logarithms are very slowly growing functions. In particular, for every

base *b*, the function log*b n* is asymptotically bounded by every function of the

form *nx*, even for (noninteger) values of *x* arbitrary close to 0.

(2.8) *For every b >* 1 *and every x >* 0*, we have* log*b n* = *O(nx).*

One can directly translate between logarithms of different bases using the

following fundamental identity:

log*a n* = log*b n*

log*b a*.

This equation explains why you’ll often notice people writing bounds like

*O(*log *n)* without indicating the base of the logarithm. This is not sloppy

usage: the identity above says that log*a n* = 1

log*b a* · log*b n*, so the point is that

log*a n* =  *(*log*b n)*, and the base of the logarithm is not important when writing

bounds using asymptotic notation.

**42** Chapter 2 Basics of Algorithm Analysis

***Exponentials*** Exponential functions are functions of the form *f(n)* = *rn* for

some constant base *r*. Here we will be concerned with the case in which *r >* 1,

which results in a very fast-growing function.

In particular, where polynomials raise *n* to a fixed exponent, exponentials

raise a fixed number to *n* as a power; this leads to much faster rates of growth.

One way to summarize the relationship between polynomials and exponentials

is as follows.

(2.9) *For every r >* 1 *and every d >* 0*, we have nd* = *O(rn).*

In particular, every exponential grows faster than every polynomial. And as

we saw in Table 2.1, when you plug in actual values of *n*, the differences in

growth rates are really quite impressive.

Just as people write *O(*log *n)* without specifying the base, you’ll also see

people write “The running time of this algorithm is exponential,” without

specifying which exponential function they have in mind. Unlike the liberal

use of log *n*, which is justified by ignoring constant factors, this generic use of

the term “exponential” is somewhat sloppy. In particular, for different bases

*r > s >* 1, it is never the case that *rn* =  *(sn)*. Indeed, this would require that

for some constant *c >* 0, we would have *rn* ≤ *csn* for all sufficiently large *n*.

But rearranging this inequality would give *(r/s)n* ≤ *c* for all sufficiently large

*n*. Since *r > s*, the expression *(r/s)n* is tending to infinity with *n*, and so it

cannot possibly remain bounded by a fixed constant *c*.

So asymptotically speaking, exponential functions are all different. Still,

it’s usually clear what people intend when they inexactly write “The running

time of this algorithm is exponential”—they typically mean that the running

time grows at least as fast as *some* exponential function, and all exponentials

grow so fast that we can effectively dismiss this algorithm without working out

further details of the exact running time. This is not entirely fair. Occasionally

there’s more going on with an exponential algorithm than first appears, as

we’ll see, for example, in Chapter 10; but as we argued in the first section of

this chapter, it’s a reasonable rule of thumb.

Taken together, then, logarithms, polynomials, and exponentials serve as

useful landmarks in the range of possible functions that you encounter when

analyzing running times. Logarithms grow more slowly than polynomials, and

polynomials grow more slowly than exponentials.

2.3 Implementing the Stable Matching Algorithm

Using Lists and Arrays

We’ve now seen a general approach for expressing bounds on the running

time of an algorithm. In order to asymptotically analyze the running time of

2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays **43**

an algorithm expressed in a high-level fashion—as we expressed the Gale

Shapley Stable Matching algorithm in Chapter 1, for example—one doesn’t

have to actually program, compile, and execute it, but one does have to think

about how the data will be represented and manipulated in an implementation

of the algorithm, so as to bound the number of computational steps it takes.

The implementation of basic algorithms using data structures is something

that you probably have had some experience with. In this book, data structures

will be covered in the context of implementing specific algorithms, and so we

will encounter different data structures based on the needs of the algorithms

we are developing. To get this process started, we consider an implementation

of the Gale-Shapley Stable Matching algorithm; we showed earlier that the

algorithm terminates in at most *n*2 iterations, and our implementation here

provides a corresponding worst-case running time of *O(n*2*)*, counting actual

computational steps rather than simply the total number of iterations. To get

such a bound for the Stable Matching algorithm, we will only need to use two

of the simplest data structures: *lists* and *arrays*. Thus, our implementation also

provides a good chance to review the use of these basic data structures as well.

In the Stable Matching Problem, each man and each woman has a ranking

of all members of the opposite gender. The very first question we need to

discuss is how such a ranking will be represented. Further, the algorithm

maintains a matching and will need to know at each step which men and

women are free, and who is matched with whom. In order to implement the

algorithm, we need to decide which data structures we will use for all these

things.

An important issue to note here is that the choice of data structure is up

to the algorithm designer; for each algorithm we will choose data structures

that make it efficient and easy to implement. In some cases, this may involve

*preprocessing* the input to convert it from its given input representation into a

data structure that is more appropriate for the problem being solved.

**Arrays and Lists**

To start our discussion we will focus on a single list, such as the list of women

in order of preference by a single man. Maybe the simplest way to keep a list

of *n* elements is to use an array *A* of length *n*, and have *A*[*i*] be the *i*th element

of the list. Such an array is simple to implement in essentially all standard

programming languages, and it has the following properties.

. We can answer a query of the form “What is the *i*th element on the list?”

in *O(*1*)* time, by a direct access to the value *A*[*i*].

. If we want to determine whether a particular element *e* belongs to the

list (i.e., whether it is equal to *A*[*i*] for some *i*), we need to check the

**44** Chapter 2 Basics of Algorithm Analysis

elements one by one in *O(n)* time, assuming we don’t know anything

about the order in which the elements appear in *A*.

. If the array elements are sorted in some clear way (either numerically

or alphabetically), then we can determine whether an element *e* belongs

to the list in *O(*log *n)* time using *binary search*; we will not need to use

binary search for any part of our stable matching implementation, but

we will have more to say about it in the next section.

An array is less good for dynamically maintaining a list of elements that

changes over time, such as the list of free men in the Stable Matching algorithm;

since men go from being free to engaged, and potentially back again, a list of

free men needs to grow and shrink during the execution of the algorithm. It

is generally cumbersome to frequently add or delete elements to a list that is

maintained as an array.

An alternate, and often preferable, way to maintain such a dynamic set

of elements is via a linked list. In a linked list, the elements are sequenced

together by having each element point to the next in the list. Thus, for each

element *v* on the list, we need to maintain a pointer to the next element; we

set this pointer to *null* if *i* is the last element. We also have a pointer First

that points to the first element. By starting at First and repeatedly following

pointers to the next element until we reach *null*, we can thus traverse the entire

contents of the list in time proportional to its length.

A generic way to implement such a linked list, when the set of possible

elements may not be fixed in advance, is to allocate a record *e* for each element

that we want to include in the list. Such a record would contain a field *e*.val

that contains the value of the element, and a field *e*.Next that contains a

pointer to the next element in the list. We can create a *doubly linked list*, which

is traversable in both directions, by also having a field *e*.Prev that contains

a pointer to the previous element in the list. (*e*.Prev = *null* if *e* is the first

element.) We also include a pointer Last, analogous to First, that points to

the last element in the list. A schematic illustration of part of such a list is

shown in the first line of Figure 2.1.

A doubly linked list can be modified as follows.

. *Deletion.* To delete the element *e* from a doubly linked list, we can just

“splice it out” by having the previous element, referenced by *e*.Prev, and

the next element, referenced by *e*.Next, point directly to each other. The

deletion operation is illustrated in Figure 2.1.

. *Insertion.* To insert element *e* between elements *d* and *f* in a list, we

“splice it in” by updating *d*.Next and *f*.Prev to point to *e*, and the Next

and Prev pointers of *e* to point to *d* and *f*, respectively. This operation is

2.3 Implementing the Stable Matching Algorithm Using Lists and Arrays **45**

**Before deleting *e*:** *val*

**After deleting *e*:** *val*

Element *e*

*val val*

Element *e*

*val val*

Figure 2.1 A schematic representation of a doubly linked list, showing the deletion of an element *e*.

essentially the reverse of deletion, and indeed one can see this operation at work by reading Figure 2.1 from bottom to top.

Inserting or deleting *e* at the beginning of the list involves updating the First pointer, rather than updating the record of the element before *e*.

While lists are good for maintaining a dynamically changing set, they also have disadvantages. Unlike arrays, we cannot find the *i*th element of the list in *O(*1*)* time: to find the *i*th element, we have to follow the Next pointers starting from the beginning of the list, which takes a total of *O(i)* time.

Given the relative advantages and disadvantages of arrays and lists, it may happen that we receive the input to a problem in one of the two formats and want to convert it into the other. As discussed earlier, such preprocessing is often useful; and in this case, it is easy to convert between the array and list representations in *O(n)* time. This allows us to freely choose the data structure that suits the algorithm better and not be constrained by the way the information is given as input.

**Implementing the Stable Matching Algorithm**

Next we will use arrays and linked lists to implement the Stable Matching algo rithm from Chapter 1. We have already shown that the algorithm terminates in at most *n*2 iterations, and this provides a type of upper bound on the running time. However, if we actually want to implement the G-S algorithm so that it runs in time proportional to *n*2, we need to be able to implement each iteration in constant time. We discuss how to do this now.

For simplicity, assume that the set of men and women are both {1, . . . , *n*}. To ensure this, we can order the men and women (say, alphabetically), and associate number *i* with the *i*th man *mi* or *i*th women *wi* in this order. This

**46** Chapter 2 Basics of Algorithm Analysis

assumption (or notation) allows us to define an array indexed by all men

or all women. We need to have a preference list for each man and for each

woman. To do this we will have two arrays, one for women’s preference lists

and one for the men’s preference lists; we will use ManPref[*m*, *i*] to denote

the *i*th woman on man *m*’s preference list, and similarly WomanPref[*w*, *i*] to

be the *i*th man on the preference list of woman *w*. Note that the amount of

space needed to give the preferences for all 2*n* individuals is *O(n*2*)*, as each

person has a list of length *n*.

We need to consider each step of the algorithm and understand what data

structure allows us to implement it efficiently. Essentially, we need to be able

to do each of four things in constant time.

1. We need to be able to identify a free man.

2. We need, for a man *m*, to be able to identify the highest-ranked woman

to whom he has not yet proposed.

3. For a woman *w*, we need to decide if *w* is currently engaged, and if she

is, we need to identify her current partner.

4. For a woman *w* and two men *m* and *m*, we need to be able to decide,

again in constant time, which of *m* or *m*is preferred by *w*.

First, consider selecting a free man. We will do this by maintaining the set

of free men as a linked list. When we need to select a free man, we take the

first man *m* on this list. We delete *m* from the list if he becomes engaged, and

possibly insert a different man *m*, if some other man *m*becomes free. In this

case, *m*can be inserted at the front of the list, again in constant time.

Next, consider a man *m*. We need to identify the highest-ranked woman

to whom he has not yet proposed. To do this we will need to maintain an extra

array Next that indicates for each man *m* the position of the next woman he

will propose to on his list. We initialize Next[*m*]= 1 for all men *m*. If a man *m*

needs to propose to a woman, he’ll propose to *w* = ManPref[*m*,Next[*m*]], and

once he proposes to *w*, we increment the value of Next[*m*] by one, regardless

of whether or not *w* accepts the proposal.

Now assume man *m* proposes to woman *w*; we need to be able to identify

the man *m*that *w* is engaged to (if there is such a man). We can do this by

maintaining an array Current of length *n*, where Current[*w*] is the woman

*w*’s current partner *m*. We set Current[*w*] to a special null symbol when we

need to indicate that woman *w* is not currently engaged; at the start of the

algorithm, Current[*w*] is initialized to this null symbol for all women *w*.

To sum up, the data structures we have set up thus far can implement the

operations (1)–(3) in *O(*1*)* time each.

2.4 A Survey of Common Running Times **47**

Maybe the trickiest question is how to maintain women’s preferences to

keep step (4) efficient. Consider a step of the algorithm, when man *m* proposes

to a woman *w*. Assume *w* is already engaged, and her current partner is

*m*=Current[*w*]. We would like to decide in *O(*1*)* time if woman *w* prefers *m*

or *m*. Keeping the women’s preferences in an array WomanPref, analogous to

the one we used for men, does not work, as we would need to walk through

*w*’s list one by one, taking *O(n)* time to find *m* and *m*on the list. While *O(n)*

is still polynomial, we can do a lot better if we build an auxiliary data structure

at the beginning.

At the start of the algorithm, we create an *n* × *n* array Ranking, where

Ranking[*w*, *m*] contains the rank of man *m* in the sorted order of *w*’s prefer

ences. By a single pass through *w*’s preference list, we can create this array in

linear time for each woman, for a total initial time investment proportional to

*n*2. Then, to decide which of *m* or *m*is preferred by *w*, we simply compare

the values Ranking[*w*, *m*] and Ranking[*w*, *m*].

This allows us to execute step (4) in constant time, and hence we have

everything we need to obtain the desired running time.

(2.10) *The data structures described above allow us to implement the G-S*

*algorithm in O(n*2*) time.*

2.4 A Survey of Common Running Times

When trying to analyze a new algorithm, it helps to have a rough sense of

the “landscape” of different running times. Indeed, there are styles of analysis

that recur frequently, and so when one sees running-time bounds like *O(n)*,

*O(n* log *n)*, and *O(n*2*)* appearing over and over, it’s often for one of a very

small number of distinct reasons. Learning to recognize these common styles

of analysis is a long-term goal. To get things under way, we offer the following

survey of common running-time bounds and some of the typical approaches

that lead to them.

Earlier we discussed the notion that most problems have a natural “search

space”—the set of all possible solutions—and we noted that a unifying theme

in algorithm design is the search for algorithms whose performance is more

efficient than a brute-force enumeration of this search space. In approaching a

new problem, then, it often helps to think about two kinds of bounds: one on

the running time you hope to achieve, and the other on the size of the problem’s

natural search space (and hence on the running time of a brute-force algorithm

for the problem). The discussion of running times in this section will begin in

many cases with an analysis of the brute-force algorithm, since it is a useful

**48** Chapter 2 Basics of Algorithm Analysis

way to get one’s bearings with respect to a problem; the task of improving on

such algorithms will be our goal in most of the book.

**Linear Time**

An algorithm that runs in *O(n)*, or linear, time has a very natural property:

its running time is at most a constant factor times the size of the input. One

basic way to get an algorithm with this running time is to process the input

in a single pass, spending a constant amount of time on each item of input

encountered. Other algorithms achieve a linear time bound for more subtle

reasons. To illustrate some of the ideas here, we consider two simple linear

time algorithms as examples.

***Computing the Maximum*** Computing the maximum of *n* numbers, for ex

ample, can be performed in the basic “one-pass” style. Suppose the numbers

are provided as input in either a list or an array. We process the numbers

*a*1, *a*2,..., *an* in order, keeping a running estimate of the maximum as we go.

Each time we encounter a number *ai*, we check whether *ai* is larger than our

current estimate, and if so we update the estimate to *ai*.

*max* = *a*1

For *i* = 2 to *n*

If *ai > max* then

set *max* = *ai*

Endif

Endfor

In this way, we do constant work per element, for a total running time of *O(n)*.

Sometimes the constraints of an application force this kind of one-pass

algorithm on you—for example, an algorithm running on a high-speed switch

on the Internet may see a stream of packets flying past it, and it can try

computing anything it wants to as this stream passes by, but it can only perform

a constant amount of computational work on each packet, and it can’t save

the stream so as to make subsequent scans through it. Two different subareas

of algorithms, *online algorithms* and *data stream algorithms*, have developed

to study this model of computation.

***Merging Two Sorted Lists*** Often, an algorithm has a running time of *O(n)*,

but the reason is more complex. We now describe an algorithm for merging

two sorted lists that stretches the one-pass style of design just a little, but still

has a linear running time.

Suppose we are given two lists of *n* numbers each, *a*1, *a*2,..., *an* and

*b*1, *b*2,..., *bn*, and each is already arranged in ascending order. We’d like to

2.4 A Survey of Common Running Times **49**

merge these into a single list *c*1, *c*2,..., *c*2*n* that is also arranged in ascending

order. For example, merging the lists 2, 3, 11, 19 and 4, 9, 16, 25 results in the

output 2, 3, 4, 9, 11, 16, 19, 25.

To do this, we could just throw the two lists together, ignore the fact that

they’re separately arranged in ascending order, and run a sorting algorithm.

But this clearly seems wasteful; we’d like to make use of the existing order in

the input. One way to think about designing a better algorithm is to imagine

performing the merging of the two lists by hand: suppose you’re given two

piles of numbered cards, each arranged in ascending order, and you’d like to

produce a single ordered pile containing all the cards. If you look at the top

card on each stack, you know that the smaller of these two should go first on

the output pile; so you could remove this card, place it on the output, and now

iterate on what’s left.

In other words, we have the following algorithm.

To merge sorted lists *A* = *a*1,..., *an* and *B* = *b*1,..., *bn*:

Maintain a *Current* pointer into each list, initialized to

point to the front elements

While both lists are nonempty:

Let *ai* and *bj* be the elements pointed to by the *Current* pointer

Append the smaller of these two to the output list

Advance the *Current* pointer in the list from which the

smaller element was selected

EndWhile

Once one list is empty, append the remainder of the other list

to the output

See Figure 2.2 for a picture of this process.

Append the smaller of

*ai* and *bj* to the output.

*ai*

*////// A*

Merged result

*/// bj*

*B*

Figure 2.2 To merge sorted lists *A* and *B*, we repeatedly extract the smaller item from the front of the two lists and append it to the output.

**50** Chapter 2 Basics of Algorithm Analysis

Now, to show a linear-time bound, one is tempted to describe an argument

like what worked for the maximum-finding algorithm: “We do constant work

per element, for a total running time of *O(n)*.” But it is actually not true that

we do only constant work per element. Suppose that *n* is an even number, and

consider the lists *A* = 1, 3, 5, . . . , 2*n* − 1 and *B* = *n*, *n* + 2, *n* + 4, . . . , 3*n* − 2.

The number *b*1 at the front of list *B* will sit at the front of the list for *n/*2

iterations while elements from *A* are repeatedly being selected, and hence

it will be involved in  *(n)* comparisons. Now, it is true that each element

can be involved in at most *O(n)* comparisons (at worst, it is compared with

each element in the other list), and if we sum this over all elements we get

a running-time bound of *O(n*2*)*. This is a correct bound, but we can show

something much stronger.

The better way to argue is to bound the number of iterations of the While

loop by an “accounting” scheme. Suppose we *charge* the cost of each iteration

to the element that is selected and added to the output list. An element can

be charged only once, since at the moment it is first charged, it is added

to the output and never seen again by the algorithm. But there are only 2*n*

elements total, and the cost of each iteration is accounted for by a charge to

some element, so there can be at most 2*n* iterations. Each iteration involves a

constant amount of work, so the total running time is *O(n)*, as desired.

While this merging algorithm iterated through its input lists in order, the

“interleaved” way in which it processed the lists necessitated a slightly subtle

running-time analysis. In Chapter 3 we will see linear-time algorithms for

graphs that have an even more complex flow of control: they spend a constant

amount of time on each node and edge in the underlying graph, but the order

in which they process the nodes and edges depends on the structure of the

graph.

*O(n* log *n)* **Time**

*O(n* log *n)* is also a very common running time, and in Chapter 5 we will

see one of the main reasons for its prevalence: it is the running time of any

algorithm that splits its input into two equal-sized pieces, solves each piece

recursively, and then combines the two solutions in linear time.

Sorting is perhaps the most well-known example of a problem that can be

solved this way. Specifically, the *Mergesort* algorithm divides the set of input

numbers into two equal-sized pieces, sorts each half recursively, and then

merges the two sorted halves into a single sorted output list. We have just

seen that the merging can be done in linear time; and Chapter 5 will discuss

how to analyze the recursion so as to get a bound of *O(n* log *n)* on the overall

running time.

2.4 A Survey of Common Running Times **51**

One also frequently encounters *O(n* log *n)* as a running time simply be

cause there are many algorithms whose most expensive step is to sort the

input. For example, suppose we are given a set of *n* time-stamps *x*1, *x*2,..., *xn*

on which copies of a file arrived at a server, and we’d like to find the largest

interval of time between the first and last of these time-stamps during which

no copy of the file arrived. A simple solution to this problem is to first sort the

time-stamps *x*1, *x*2,..., *xn* and then process them in sorted order, determining

the sizes of the gaps between each number and its successor in ascending

order. The largest of these gaps is the desired subinterval. Note that this algo

rithm requires *O(n* log *n)* time to sort the numbers, and then it spends constant

work on each number in ascending order. In other words, the remainder of the

algorithm after sorting follows the basic recipe for linear time that we discussed

earlier.

**Quadratic Time**

Here’s a basic problem: suppose you are given *n* points in the plane, each

specified by *(x*, *y)* coordinates, and you’d like to find the pair of points that

are closest together. The natural brute-force algorithm for this problem would

enumerate all pairs of points, compute the distance between each pair, and

then choose the pair for which this distance is smallest.

What is the running time of this algorithm? The number of pairs of points

is  *n*2= *n(n*−1*)*

~~2~~ , and since this quantity is bounded by 1~~2~~*n*2, it is *O(n*2*)*. More

crudely, the number of pairs is *O(n*2*)* because we multiply the number of

ways of choosing the first member of the pair (at most *n*) by the number

of ways of choosing the second member of the pair (also at most *n*). The

distance between points *(xi*, *yi)* and *(xj*, *yj* *)* can be computed by the formula

*(xi* − *xj)*2 + *(yi* − *yj)*2 in constant time, so the overall running time is *O(n*2*)*.

This example illustrates a very common way in which a running time of *O(n*2*)*

arises: performing a search over all pairs of input items and spending constant

time per pair.

Quadratic time also arises naturally from a pair of *nested loops*: An algo

rithm consists of a loop with *O(n)* iterations, and each iteration of the loop

launches an internal loop that takes *O(n)* time. Multiplying these two factors

of *n* together gives the running time.

The brute-force algorithm for finding the closest pair of points can be

written in an equivalent way with two nested loops:

For each input point *(xi*, *yi)*

For each other input point *(xj*, *yj)*

Compute distance *d* =

*(xi* − *xj)*2 + *(yi* − *yj)*2

**52** Chapter 2 Basics of Algorithm Analysis

If *d* is less than the current minimum, update minimum to *d*

Endfor

Endfor

Note how the “inner” loop, over *(xj*, *yj)*, has *O(n)* iterations, each taking

constant time; and the “outer” loop, over *(xi*, *yi)*, has *O(n)* iterations, each

invoking the inner loop once.

It’s important to notice that the algorithm we’ve been discussing for the

Closest-Pair Problem really is just the brute-force approach: the natural search

space for this problem has size *O(n*2*)*, and we’re simply enumerating it. At

first, one feels there is a certain inevitability about this quadratic algorithm—

we have to measure all the distances, don’t we?—but in fact this is an illusion.

In Chapter 5 we describe a very clever algorithm that finds the closest pair of

points in the plane in only *O(n* log *n)* time, and in Chapter 13 we show how

randomization can be used to reduce the running time to *O(n)*.

**Cubic Time**

More elaborate sets of nested loops often lead to algorithms that run in

*O(n*3*)* time. Consider, for example, the following problem. We are given sets

*S*1, *S*2,..., *Sn*, each of which is a subset of {1, 2, . . . , *n*}, and we would like

to know whether some pair of these sets is disjoint—in other words, has no

elements in common.

What is the running time needed to solve this problem? Let’s suppose that

each set *Si* is represented in such a way that the elements of *Si* can be listed in

constant time per element, and we can also check in constant time whether a

given number *p* belongs to *Si*. The following is a direct way to approach the

problem.

For pair of sets *Si* and *Sj*

Determine whether *Si* and *Sj* have an element in common

Endfor

This is a concrete algorithm, but to reason about its running time it helps to

open it up (at least conceptually) into three nested loops.

For each set *Si*

For each other set *Sj*

For each element *p* of *Si*

Determine whether *p* also belongs to *Sj*

Endfor

If no element of *Si* belongs to *Sj* then

2.4 A Survey of Common Running Times **53**

Report that *Si* and *Sj* are disjoint

Endif

Endfor

Endfor

Each of the sets has maximum size *O(n)*, so the innermost loop takes time

*O(n)*. Looping over the sets *Sj* involves *O(n)* iterations around this innermost

loop; and looping over the sets *Si* involves *O(n)* iterations around this. Multi

plying these three factors of *n* together, we get the running time of *O(n*3*)*.

For this problem, there are algorithms that improve on *O(n*3*)* running

time, but they are quite complicated. Furthermore, it is not clear whether

the improved algorithms for this problem are practical on inputs of reasonable

size.

*O(nk)* **Time**

In the same way that we obtained a running time of *O(n*2*)* by performing brute

force search over all pairs formed from a set of *n* items, we obtain a running

time of *O(nk)* for any constant *k* when we search over all subsets of size *k*.

Consider, for example, the problem of finding independent sets in a graph,

which we discussed in Chapter 1. Recall that a set of nodes is independent

if no two are joined by an edge. Suppose, in particular, that for some fixed

constant *k*, we would like to know if a given *n*-node input graph *G* has an

independent set of size *k*. The natural brute-force algorithm for this problem

would enumerate all subsets of *k* nodes, and for each subset *S* it would check

whether there is an edge joining any two members of *S*. That is,

For each subset *S* of *k* nodes

Check whether *S* constitutes an independent set

If *S* is an independent set then

Stop and declare success

Endif

Endfor

If no *k*-node independent set was found then

Declare failure

Endif

To understand the running time of this algorithm, we need to consider two

quantities. First, the total number of *k*-element subsets in an *n*-element set is

*n k*

= *n(n* − 1*)(n* − 2*)* ...*(n* − *k* + 1*) k(k* − 1*)(k* − 2*)* ...*(*2*)(*1*)* ≤*nk k*!.

**54** Chapter 2 Basics of Algorithm Analysis

Since we are treating *k* as a constant, this quantity is *O(nk)*. Thus, the outer

loop in the algorithm above will run for *O(nk)* iterations as it tries all *k*-node

subsets of the *n* nodes of the graph.

Inside this loop, we need to test whether a given set *S* of *k* nodes constitutes

an independent set. The definition of an independent set tells us that we need

to check, for each pair of nodes, whether there is an edge joining them. Hence

this is a search over pairs, like we saw earlier in the discussion of quadratic

time; it requires looking at  *k*2, that is, *O(k*2*)*, pairs and spending constant time

on each.

Thus the total running time is *O(k*2*nk)*. Since we are treating *k* as a constant

here, and since constants can be dropped in *O(*·*)* notation, we can write this

running time as *O(nk)*.

Independent Set is a principal example of a problem believed to be compu

tationally hard, and in particular it is believed that no algorithm to find *k*-node

independent sets in arbitrary graphs can avoid having some dependence on *k*

in the exponent. However, as we will discuss in Chapter 10 in the context of

a related problem, even once we’ve conceded that brute-force search over *k*

element subsets is necessary, there can be different ways of going about this

that lead to significant differences in the efficiency of the computation.

**Beyond Polynomial Time**

The previous example of the Independent Set Problem starts us rapidly down

the path toward running times that grow faster than any polynomial. In

particular, two kinds of bounds that come up very frequently are 2*n* and *n*!,

and we now discuss why this is so.

Suppose, for example, that we are given a graph and want to find an

independent set of *maximum* size (rather than testing for the existence of one

with a given number of nodes). Again, people don’t know of algorithms that

improve significantly on brute-force search, which in this case would look as

follows.

For each subset *S* of nodes

Check whether *S* constitutes an independent set

If *S* is a larger independent set than the largest seen so far then

Record the size of *S* as the current maximum

Endif

Endfor

This is very much like the brute-force algorithm for *k*-node independent sets,

except that now we are iterating over *all* subsets of the graph. The total number