



# Algorithms

FOURTH EDITION

PART II

ROBERT SEDGEWICK | KEVIN WAYNE

# Algorithms

FOURTH EDITION

**PART II**

*This page intentionally left blank*

# Algorithms

FOURTH EDITION

## PART II

Robert Sedgewick  
and  
Kevin Wayne

Princeton University

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco  
New York • Toronto • Montreal • London • Munich • Paris • Madrid  
Capetown • Sydney • Tokyo • Singapore • Mexico City

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 the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at (800) 382-3419 or [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com).

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the United States, please contact [international@pearsoned.com](mailto:international@pearsoned.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-379911-8

ISBN-10: 0-13-379911-5

First digital release, February 2014

---

*To Adam, Andrew, Brett, Robbie  
and especially Linda*

---

---

*To Jackie and Alex*

---

# CONTENTS

*Note: This is an online edition of Chapters 4 through 6 of Algorithms, Fourth Edition, which contains the content covered in our online course Algorithms, Part II.*

*For more information, see <http://algs4.cs.princeton.edu>.*

<i>Preface</i> . . . . .	<i>ix</i>
--------------------------	-----------

*Chapters 1 through 3, which correspond to our online course Algorithms, Part I, are available as Algorithms, Fourth Edition, Part I.*

<b>4. Graphs</b> . . . . .	<b>515</b>
4.1 Undirected Graphs	518
<i>Glossary</i> • <i>Undirected graph type</i> • <i>Adjacency-lists representation</i> • <i>Depth-first search</i> • <i>Breadth-first search</i> • <i>Connected components</i> • <i>Degrees of separation</i>	
4.2 Directed Graphs	566
<i>Glossary</i> • <i>Digraph data type</i> • <i>Depth-first search</i> • <i>Directed cycle detection</i> • <i>Precedence-constrained scheduling</i> • <i>Topological sort</i> • <i>Strong connectivity</i> • <i>Kosaraju-Sharir algorithm</i> • <i>Transitive closure</i>	
4.3 Minimum Spanning Trees	604
<i>Cut property</i> • <i>Greedy algorithm</i> • <i>Edge-weighted graph data type</i> • <i>Prim's algorithm</i> • <i>Kruskal's algorithm</i>	
4.4 Shortest Paths	638
<i>Properties of shortest paths</i> • <i>Edge-weighted digraph data types</i> • <i>Generic shortest paths algorithm</i> • <i>Dijkstra's algorithm</i> • <i>Shortest paths in edge-weighted DAGs</i> • <i>Critical-path method</i> • <i>Bellman-Ford algorithm</i> • <i>Negative cycle detection</i> • <i>Arbitrage</i>	

<b>5. Strings . . . . .</b>	<b>695</b>
<b>5.1 String Sorts</b>	<b>702</b>
<i>Key-indexed counting • LSD string sort • MSD string sort • 3-way string quicksort</i>	
<b>5.2 Tries</b>	<b>730</b>
<i>String symbol table API • R-way tries • Ternary search tries • Character-based operations</i>	
<b>5.3 Substring Search</b>	<b>758</b>
<i>Brute-force algorithm • Knuth-Morris-Pratt algorithm • Boyer-Moore algorithm • Rabin-Karp fingerprint algorithm</i>	
<b>5.4 Regular Expressions</b>	<b>788</b>
<i>Describing patterns with REs • Applications • Nondeterministic finite-state automata • Simulating an NFA • Building an NFA corresponding to an RE</i>	
<b>5.5 Data Compression</b>	<b>810</b>
<i>Rules of the game • Reading and writing binary data • Limitations • Run-length coding • Huffman compression • LZW compression</i>	
<b>6. Context . . . . .</b>	<b>853</b>
<b>Event-Driven Simulation</b>	<b>856</b>
<i>Hard-disc model • Collision prediction • Collision resolution</i>	
<b>B-trees</b>	<b>866</b>
<i>Cost model • Search and insert</i>	
<b>Suffix Arrays</b>	<b>875</b>
<i>Suffix sorting • Longest repeated substring • Keyword in context</i>	
<b>Network-Flow Algorithms</b>	<b>886</b>
<i>Maximum flow • Minimum cut • Ford-Fulkerson algorithm</i>	
<b>Reduction</b>	<b>903</b>
<i>Sorting • Shortest path • Bipartite matching • Linear programming</i>	
<b>Intractability</b>	<b>910</b>
<i>Longest-paths problem • P vs. NP • Boolean satisfiability • NP-completeness</i>	

*This page intentionally left blank*

# PREFACE

This book is intended to survey the most important computer algorithms in use today, and to teach fundamental techniques to the growing number of people in need of knowing them. It is intended for use as a textbook for a second course in computer science, after students have acquired basic programming skills and familiarity with computer systems. The book also may be useful for self-study or as a reference for people engaged in the development of computer systems or applications programs, since it contains implementations of useful algorithms and detailed information on performance characteristics and clients. The broad perspective taken makes the book an appropriate introduction to the field.

THE STUDY OF ALGORITHMS AND DATA STRUCTURES is fundamental to any computer-science curriculum, but it is not just for programmers and computer-science students. Everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms in this book represent a body of knowledge developed over the last 50 years that has become indispensable. From  $N$ -body simulation problems in physics to genetic-sequencing problems in molecular biology, the basic methods described here have become essential in scientific research; from architectural modeling systems to aircraft simulation, they have become essential tools in engineering; and from database systems to internet search engines, they have become essential parts of modern software systems. And these are but a few examples—as the scope of computer applications continues to grow, so grows the impact of the basic methods covered here.

In Chapter 1, we develop our fundamental approach to studying algorithms, including coverage of data types for stacks, queues, and other low-level abstractions that we use throughout the book. In Chapters 2 and 3, we survey fundamental algorithms for sorting and searching; and in Chapters 4 and 5, we cover algorithms for processing graphs and strings. Chapter 6 is an overview placing the rest of the material in the book in a larger context.

**Distinctive features** The orientation of the book is to study algorithms likely to be of practical use. The book teaches a broad variety of algorithms and data structures and provides sufficient information about them that readers can confidently implement, debug, and put them to work in any computational environment. The approach involves:

**Algorithms.** Our descriptions of algorithms are based on complete implementations and on a discussion of the operations of these programs on a consistent set of examples. Instead of presenting pseudo-code, we work with real code, so that the programs can quickly be put to practical use. Our programs are written in Java, but in a style such that most of our code can be reused to develop implementations in other modern programming languages.

**Data types.** We use a modern programming style based on data abstraction, so that algorithms and their data structures are encapsulated together.

**Applications.** Each chapter has a detailed description of applications where the algorithms described play a critical role. These range from applications in physics and molecular biology, to engineering computers and systems, to familiar tasks such as data compression and searching on the web.

**A scientific approach.** We emphasize developing mathematical models for describing the performance of algorithms, using the models to develop hypotheses about performance, and then testing the hypotheses by running the algorithms in realistic contexts.

**Breadth of coverage.** We cover basic abstract data types, sorting algorithms, searching algorithms, graph processing, and string processing. We keep the material in algorithmic context, describing data structures, algorithm design paradigms, reduction, and problem-solving models. We cover classic methods that have been taught since the 1960s and new methods that have been invented in recent years.

Our primary goal is to introduce the most important algorithms in use today to as wide an audience as possible. These algorithms are generally ingenious creations that, remarkably, can each be expressed in just a dozen or two lines of code. As a group, they represent problem-solving power of amazing scope. They have enabled the construction of computational artifacts, the solution of scientific problems, and the development of commercial applications that would not have been feasible without them.

**Booksit**e An important feature of the book is its relationship to the booksite `algs4.cs.princeton.edu`. This site is freely available and contains an extensive amount of material about algorithms and data structures, for teachers, students, and practitioners, including:

**An online synopsis.** The text is summarized in the booksite to give it the same overall structure as the book, but linked so as to provide easy navigation through the material.

**Full implementations.** All code in the book is available on the booksite, in a form suitable for program development. Many other implementations are also available, including advanced implementations and improvements described in the book, answers to selected exercises, and client code for various applications. The emphasis is on testing algorithms in the context of meaningful applications.

**Exercises and answers.** The booksite expands on the exercises in the book by adding drill exercises (with answers available with a click), a wide variety of examples illustrating the reach of the material, programming exercises with code solutions, and challenging problems.

**Dynamic visualizations.** Dynamic simulations are impossible in a printed book, but the website is replete with implementations that use a graphics class to present compelling visual demonstrations of algorithm applications.

**Course materials.** A complete set of lecture slides is tied directly to the material in the book and on the booksite. A full selection of programming assignments, with check lists, test data, and preparatory material, is also included.

**Online course.** A full set of lecture videos and self-assessment materials provide opportunities for students to learn or review the material on their own and for instructors to replace or supplement their lectures.

**Links to related material.** Hundreds of links lead students to background information about applications and to resources for studying algorithms.

Our goal in creating this material was to provide a complementary approach to the ideas. Generally, you should read the book when learning specific algorithms for the first time or when trying to get a global picture, and you should use the booksite as a reference when programming or as a starting point when searching for more detail while online.

**Use in the curriculum** The book is intended as a textbook in a second course in computer science. It provides full coverage of core material and is an excellent vehicle for students to gain experience and maturity in programming, quantitative reasoning, and problem-solving. Typically, one course in computer science will suffice as a prerequisite—the book is intended for anyone conversant with a modern programming language and with the basic features of modern computer systems.

The algorithms and data structures are expressed in Java, but in a style accessible to people fluent in other modern languages. We embrace modern Java abstractions (including generics) but resist dependence upon esoteric features of the language.

Most of the mathematical material supporting the analytic results is self-contained (or is labeled as beyond the scope of this book), so little specific preparation in mathematics is required for the bulk of the book, although mathematical maturity is definitely helpful. Applications are drawn from introductory material in the sciences, again self-contained.

The material covered is a fundamental background for any student intending to major in computer science, electrical engineering, or operations research, and is valuable for any student with interests in science, mathematics, or engineering.

**Context** The book is intended to follow our introductory text, *An Introduction to Programming in Java: An Interdisciplinary Approach*, which is a broad introduction to the field. Together, these two books can support a two- or three-semester introduction to computer science that will give any student the requisite background to successfully address computation in any chosen field of study in science, engineering, or the social sciences.

The starting point for much of the material in the book was the Sedgewick series of *Algorithms* books. In spirit, this book is closest to the first and second editions of that book, but this text benefits from decades of experience teaching and learning that material. Sedgewick's current *Algorithms in C/C++/Java, Third Edition* is more appropriate as a reference or a text for an advanced course; this book is specifically designed to be a textbook for a one-semester course for first- or second-year college students and as a modern introduction to the basics and a reference for use by working programmers.

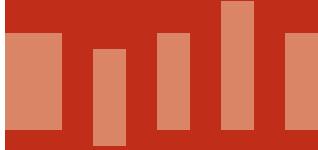
**Acknowledgments** This book has been nearly 40 years in the making, so full recognition of all the people who have made it possible is simply not feasible. Earlier editions of this book list dozens of names, including (in alphabetical order) Andrew Appel, Trina Avery, Marc Brown, Lyn Dupré, Philippe Flajolet, Tom Freeman, Dave Hanson, Janet Incerpi, Mike Schidlowsky, Steve Summit, and Chris Van Wyk. All of these people deserve acknowledgement, even though some of their contributions may have happened decades ago. For this fourth edition, we are grateful to the hundreds of students at Princeton and several other institutions who have suffered through preliminary versions of the work, and to readers around the world for sending in comments and corrections through the booksite.

We are grateful for the support of Princeton University in its unwavering commitment to excellence in teaching and learning, which has provided the basis for the development of this work.

Peter Gordon has provided wise counsel throughout the evolution of this work almost from the beginning, including a gentle introduction of the “back to the basics” idea that is the foundation of this edition. For this fourth edition, we are grateful to Barbara Wood for her careful and professional copyediting, to Julie Nahil for managing the production, and to many others at Pearson for their roles in producing and marketing the book. All were extremely responsive to the demands of a rather tight schedule without the slightest sacrifice to the quality of the result.

*Robert Sedgewick  
Kevin Wayne*

*Princeton, New Jersey  
January 2014*



## FOUR

# Graphs

<b>4.1</b>	Undirected Graphs . . . . .	518
<b>4.2</b>	Directed Graphs . . . . .	566
<b>4.3</b>	Minimum Spanning Trees . . . . .	604
<b>4.4</b>	Shortest Paths . . . . .	638

**P**airwise connections between items play a critical role in a vast array of computational applications. The relationships implied by these connections lead immediately to a host of natural questions: Is there a way to connect one item to another by following the connections? How many other items are connected to a given item? What is the shortest chain of connections between this item and this other item?

To model such situations, we use abstract mathematical objects called *graphs*. In this chapter, we examine basic properties of graphs in detail, setting the stage for us to study a variety of algorithms that are useful for answering questions of the type just posed. These algorithms serve as the basis for attacking problems in important applications whose solution we could not even contemplate without good algorithmic technology.

Graph theory, a major branch of mathematics, has been studied intensively for hundreds of years. Many important and useful properties of graphs have been discovered, many important algorithms have been developed, and many difficult problems are still actively being studied. In this chapter, we introduce a variety of fundamental graph algorithms that are important in diverse applications.

Like so many of the other problem domains that we have studied, the algorithmic investigation of graphs is relatively recent. Although a few of the fundamental algorithms are centuries old, the majority of the interesting ones have been discovered within the last several decades and have benefited from the emergence of the algorithmic technology that we have been studying. Even the simplest graph algorithms lead to useful computer programs, and the nontrivial algorithms that we examine are among the most elegant and interesting algorithms known.

To illustrate the diversity of applications that involve graph processing, we begin our exploration of algorithms in this fertile area by introducing several examples.

**Maps.** A person who is planning a trip may need to answer questions such as “What is the shortest route from Providence to Princeton?” A seasoned traveler who has experienced traffic delays on the shortest route may ask the question “What is the fastest way to get from Providence to Princeton?” To answer such questions, we process information about connections (roads) between items (intersections).

**Web content.** When we browse the web, we encounter pages that contain references (links) to other pages and we move from page to page by clicking on the links. The entire web is a graph, where the items are pages and the connections are links. Graph-processing algorithms are essential components of the search engines that help us locate information on the web.

**Circuits.** An electric circuit comprises devices such as transistors, resistors, and capacitors that are intricately wired together. We use computers to control machines that make circuits and to check that the circuits perform desired functions. We need to answer simple questions such as “Is a short-circuit present?” as well as complicated questions such as “Can we lay out this circuit on a chip without making any wires cross?” The answer to the first question depends on only the properties of the connections (wires), whereas the answer to the second question requires detailed information about the wires, the devices that those wires connect, and the physical constraints of the chip.

**Schedules.** A manufacturing process requires a variety of jobs to be performed, under a set of constraints that specify that certain jobs cannot be started until certain other jobs have been completed. How do we schedule the jobs such that we both respect the given constraints and complete the whole process in the least amount of time?

**Commerce.** Retailers and financial institutions track buy/sell orders in a market. A connection in this situation represents the transfer of cash and goods between an institution and a customer. Knowledge of the nature of the connection structure in this instance may enhance our understanding of the nature of the market.

**Matching.** Students apply for positions in selective institutions such as social clubs, universities, or medical schools. Items correspond to the students and the institutions; connections correspond to the applications. We want to discover methods for matching interested students with available positions.

**Computer networks.** A computer network consists of interconnected sites that send, forward, and receive messages of various types. We are interested in knowing about the nature of the interconnection structure because we want to lay wires and build switches that can handle the traffic efficiently.

**Software.** A compiler builds graphs to represent relationships among modules in a large software system. The items are the various classes or modules that comprise the system; connections are associated either with the possibility that a method in one class might call another (static analysis) or with actual calls while the system is in operation (dynamic analysis). We need to analyze the graph to determine how best to allocate resources to the program most efficiently.

**Social networks.** When you use a social network, you build explicit connections with your friends. Items correspond to people; connections are to friends or followers. Understanding the properties of these networks is a modern graph-processing application of intense interest not just to companies that support such networks, but also in politics, diplomacy, entertainment, education, marketing, and many other domains.

THESE EXAMPLES INDICATE THE RANGE OF APPLICATIONS for which graphs are the appropriate abstraction and also the range of computational problems that we might encounter when we work with graphs. Thousands of such problems have been studied, but many problems can be addressed in the context of one of several basic graph models—we will study the most important ones in this chapter. In practical applications, it is common for the volume of data involved to be truly huge, so that efficient algorithms make the difference between whether or not a solution is at all feasible.

To organize the presentation, we progress through the four most important types of graph models: *undirected graphs* (with simple connections), *digraphs* (where the direction of each connection is significant), *edge-weighted graphs* (where each connection has an associated weight), and *edge-weighted digraphs* (where each connection has both a direction and a weight).

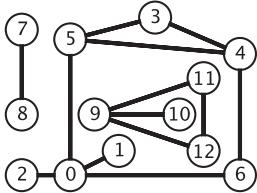
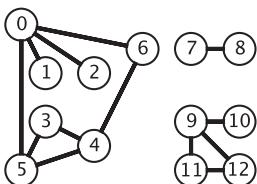
application	item	connection
<i>map</i>	intersection	road
<i>web content</i>	page	link
<i>circuit</i>	device	wire
<i>schedule</i>	job	constraint
<i>commerce</i>	customer	transaction
<i>matching</i>	student	application
<i>computer network</i>	site	connection
<i>software</i>	method	call
<i>social network</i>	person	friendship

Typical graph applications

## 4.1 UNDIRECTED GRAPHS

OUR STARTING POINT is the study of graph models where *edges* are nothing more than connections between *vertices*. We use the term *undirected graph* in contexts where we need to distinguish this model from other models (such as the title of this section), but, since this is the simplest model, we start with the following definition:

**Definition.** A *graph* is a set of *vertices* and a collection of *edges* that each connect a pair of vertices.



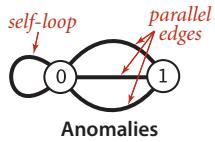
Two drawings of the same graph

at left represent the same graph, because the graph is nothing more than its (unordered) set of vertices and its (unordered) collection of edges (vertex pairs).

**Anomalies.** Our definition allows two simple anomalies:

- A *self-loop* is an edge that connects a vertex to itself.
- Two edges that connect the same pair of vertices are *parallel*.

Mathematicians sometimes refer to graphs with parallel edges as *multigraphs* and graphs with no parallel edges or self-loops as *simple graphs*. Typically, our implementations allow self-loops and parallel edges (because they arise in applications), but we do not include them in examples. Thus, we can refer to every edge just by naming the two vertices it connects.



**Glossary** A substantial amount of nomenclature is associated with graphs. Most of the terms have straightforward definitions, and, for reference, we consider them in one place: here.

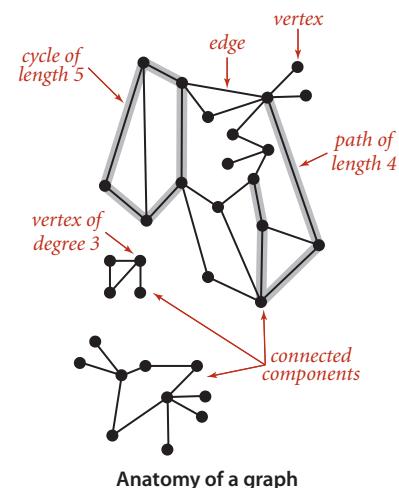
When there is an edge connecting two vertices, we say that the vertices are *adjacent* to one another and that the edge is *incident* to both vertices. The *degree* of a vertex is the number of edges incident to it. A *subgraph* is a subset of a graph's edges (and associated vertices) that constitutes a graph. Many computational tasks involve identifying subgraphs of various types. Of particular interest are edges that take us through a *sequence* of vertices in a graph.

**Definition.** A *path* in a graph is a sequence of vertices connected by edges. A *simple path* is one with no repeated vertices. A *cycle* is a path with at least one edge whose first and last vertices are the same. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices). The *length* of a path or a cycle is its number of edges.

Most often, we work with simple cycles and simple paths and drop the *simple* modifier; when we want to allow repeated vertices, we refer to *general* paths and cycles. We say that one vertex is *connected* to another if there exists a path that contains both of them. We use notation like  $u-v-w-x$  to represent a path from  $u$  to  $x$  and  $u-v-w-x-u$  to represent a cycle from  $u$  to  $v$  to  $w$  to  $x$  and back to  $u$  again. Several of the algorithms that we consider find paths and cycles. Moreover, paths and cycles lead us to consider the structural properties of a graph as a whole:

**Definition.** A graph is *connected* if there is a path from every vertex to every other vertex in the graph. A graph that is *not connected* consists of a set of *connected components*, which are maximal connected subgraphs.

Intuitively, if the vertices were physical objects, such as knots or beads, and the edges were physical connections, such as strings or wires, a connected graph would stay in one piece if picked up by any vertex, and a graph that is not connected comprises two or more such pieces. Generally, processing a graph necessitates processing the connected components one at a time.



An *acyclic* graph is a graph with no cycles. Several of the algorithms that we consider are concerned with finding acyclic subgraphs of a given graph that satisfy certain properties. We need additional terminology to refer to these structures:

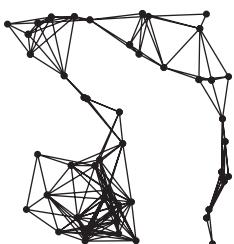
**Definition.** A *tree* is an acyclic connected graph. A disjoint set of trees is called a *forest*. A *spanning tree* of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A *spanning forest* of a graph is the union of spanning trees of its connected components.

This definition of tree is quite general: with suitable refinements it embraces the trees that we typically use to model program behavior (function-call hierarchies) and data structures (BSTs, 2-3 trees, and so forth). Mathematical properties of trees are well-studied and intuitive, so we state them without proof. For example, a graph  $G$  with  $V$  vertices is a tree if and only if it satisfies any of the following five conditions:

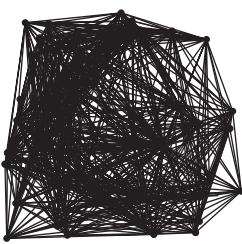
- $G$  has  $V-1$  edges and no cycles.
- $G$  has  $V-1$  edges and is connected.
- $G$  is connected, but removing any edge disconnects it.
- $G$  is acyclic, but adding any edge creates a cycle.
- Exactly one simple path connects each pair of vertices in  $G$ .

Several of the algorithms that we consider find spanning trees and forests, and these properties play an important role in their analysis and implementation.

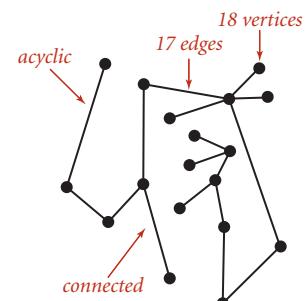
sparse ( $E = 200$ )



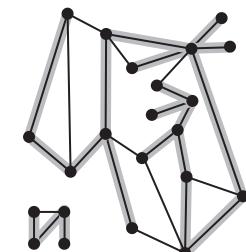
dense ( $E = 1000$ )



Two graphs ( $V = 50$ )



A tree

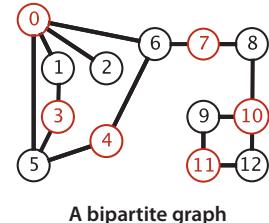


A spanning forest

The *density* of a graph is the proportion of possible pairs of vertices that are connected by edges. A *sparse* graph has relatively few of the possible edges present; a *dense* graph has relatively few of the possible edges missing. Generally, we think of a graph as being sparse if its number of different edges is within a small constant factor of  $V$  and as being dense otherwise. This rule of thumb

leaves a gray area (when the number of edges is, say,  $\sim c V^{3/2}$ ) but the distinction between sparse and dense is typically very clear in applications. The applications that we consider nearly always involve sparse graphs.

A *bipartite graph* is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set. The figure at right gives an example of a bipartite graph, where one set of vertices is colored red and the other set of vertices is colored black. Bipartite graphs arise in a natural way in many situations, one of which we will consider in detail at the end of this section.



WITH THESE PREPARATIONS, we are ready to move on to consider graph-processing algorithms. We begin by considering an API and implementation for a graph data type, then we consider classic algorithms for searching graphs and for identifying connected components. To conclude the section, we consider real-world applications where vertex names need not be integers and graphs may have huge numbers of vertices and edges.

**Undirected graph data type** Our starting point for developing graph-processing algorithms is an API that defines the fundamental graph operations. This scheme allows us to address graph-processing tasks ranging from elementary maintenance operations to sophisticated solutions of difficult problems.

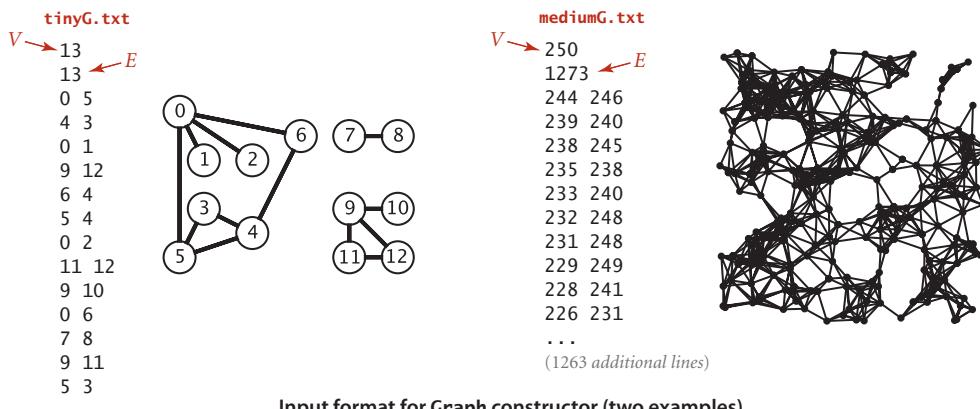
public class Graph	
Graph(int V)	<i>create a V-vertex graph with no edges</i>
Graph(In in)	<i>read a graph from input stream in</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
void addEdge(int v, int w)	<i>add edge v-w to this graph</i>
Iterable<Integer> adj(int v)	<i>vertices adjacent to v</i>
String toString()	<i>string representation</i>

API for an undirected graph

This API contains two constructors, methods to return the number of vertices and edges, a method to add an edge, a `toString()` method, and a method `adj()` that allows client code to iterate through the vertices adjacent to a given vertex (the order of iteration is not specified). Remarkably, we can build all of the algorithms that we consider in this section on the basic abstraction embodied in `adj()`.

The second constructor assumes an input format consisting of  $2E + 2$  integer values:  $V$ , then  $E$ , then  $E$  pairs of values between 0 and  $V - 1$ , each pair denoting an edge. As examples, we use the two graphs `tinyG.txt` and `mediumG.txt` that are depicted below.

Several examples of `Graph` client code are shown in the table on the facing page.



task	implementation
<i>compute the degree of v</i>	<pre>public static int degree(Graph G, int v) {     int degree = 0;     for (int w : G.adj(v)) degree++;     return degree; }</pre>
<i>compute maximum degree</i>	<pre>public static int maxDegree(Graph G) {     int max = 0;     for (int v = 0; v &lt; G.V(); v++)         if (degree(G, v) &gt; max)             max = degree(G, v);     return max; }</pre>
<i>compute average degree</i>	<pre>public static double averageDegree(Graph G) {   return 2.0 * G.E() / G.V(); }</pre>
<i>count self-loops</i>	<pre>public static int numberOfSelfLoops(Graph G) {     int count = 0;     for (int v = 0; v &lt; G.V(); v++)         for (int w : G.adj(v))             if (v == w) count++;     return count/2; // each edge counted twice }</pre>
<i>string representation of the graph's adjacency lists (instance method in Graph)</i>	<pre>public String toString() {     String s = V + " vertices, " + E + " edges\n";     for (int v = 0; v &lt; V; v++)     {         s += v + ": ";         for (int w : this.adj(v))             s += w + " ";         s += "\n";     }     return s; }</pre>

Typical graph-processing code

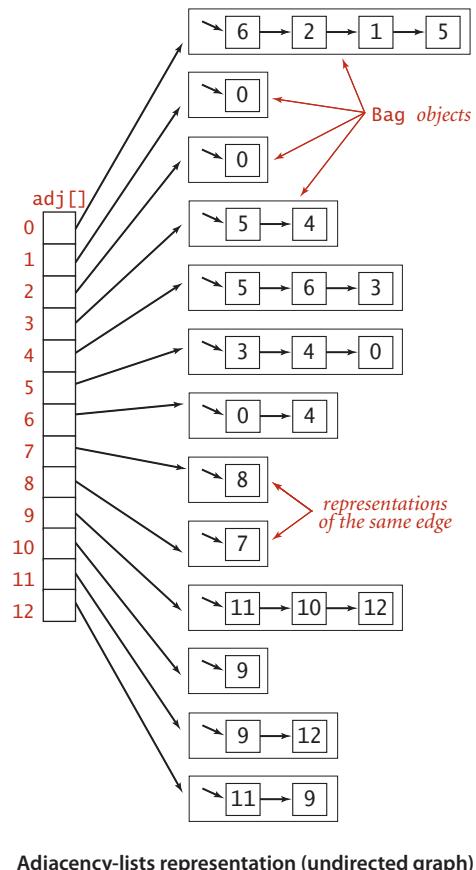
**Representation alternatives.** The next decision that we face in graph processing is which graph representation (data structure) to use to implement this API. We have two basic requirements:

- We must have the *space* to accommodate the types of graphs that we are likely to encounter in applications.
- We want to develop *time*-efficient implementations of Graph instance methods—the basic methods that we need to develop graph-processing clients.

These requirements are a bit vague, but they are still helpful in choosing among the three data structures that immediately suggest themselves for representing graphs:

- An *adjacency matrix*, where we maintain a  $V$ -by- $V$  boolean array, with the entry in row  $v$  and column  $w$  defined to be `true` if there is an edge in the graph that connects vertex  $v$  and vertex  $w$ , and to be `false` otherwise. This representation fails on the first count—graphs with millions of vertices are common and the space cost for the  $V^2$  boolean values needed is prohibitive.
- An *array of edges*, using an Edge class with two instance variables of type `int`. This direct representation is simple, but it fails on the second count—implementing `adj()` would involve examining all the edges in the graph.
- An *array of adjacency lists*, where we maintain a vertex-indexed array of lists of the vertices adjacent to each vertex. This data structure satisfies both requirements for typical applications and is the one that we will use throughout this chapter.

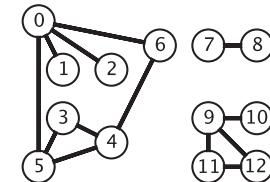
Beyond these performance objectives, a detailed examination reveals other considerations that can be important in some applications. For example, allowing parallel edges precludes the use of an adjacency matrix, since the adjacency matrix has no way to represent them.



**Adjacency-lists data structure.** The standard graph representation for graphs that are not dense is called the *adjacency-lists data structure*, where we keep track of all the vertices adjacent to each vertex on a linked list that is associated with that vertex. We maintain an array of lists so that, given a vertex, we can immediately access its list. To implement lists, we use our Bag ADT from SECTION 1.3 with a linked-list implementation, so that we can add new edges in constant time and iterate through adjacent vertices in constant time per adjacent vertex. The Graph implementation onpage 526 is based on this approach, and the figure on the facing page depicts the data structures built by this code for `tinyG.txt`. To add an edge connecting  $v$  and  $w$ , we add  $w$  to  $v$ 's adjacency list and  $v$  to  $w$ 's adjacency list. Thus, each edge appears *twice* in the data structure. This Graph implementation achieves the following performance characteristics:

- Space usage proportional to  $V + E$
- Constant time to add an edge
- Time proportional to the degree of  $v$  to iterate through vertices adjacent to  $v$  (constant time per adjacent vertex processed)

These characteristics are optimal for this set of operations, which suffice for the graph-processing applications that we consider. Parallel edges and self-loops are allowed (we do not check for them). Note: It is important to realize that the order in which edges are added to the graph determines the order in which vertices appear in the array of adjacency lists built by Graph. Many different arrays of adjacency lists can represent the same graph. When using the constructor that reads edges from an input stream, this means that the input format and the order in which edges are specified in the file determine the order in which vertices appear in the array of adjacency lists built by Graph. Since our algorithms use `adj(0)` and process all adjacent vertices without regard to the order in which they appear in the lists, this difference does not affect their correctness, but it is important to bear it in mind when debugging or following traces. To facilitate these activities, we assume that Graph has a test client that reads a graph from the input stream named as command-line argument and then prints it (relying on the `toString()` implementation on page 523) to show the order in which vertices appear in adjacency lists, which is the order in which algorithms process them (see EXERCISE 4.1.7).



<code>tinyG.txt</code>	
V	13
E	13
0	5
1	3
2	0
3	5 4
4	5 6 3
5	3 4 0
6	0 4
7	8
8	7
9	11 10 12
10	9
11	9 12
12	11 9

% java Graph tinyG.txt  
13 vertices, 13 edges

first adjacent vertex in input is last on list

second representation of each edge appears in red

Output for list-of-edges input

## Graph data type

```
public class Graph
{
    private final int V;           // number of vertices
    private int E;                // number of edges
    private Bag<Integer>[] adj;   // adjacency lists

    public Graph(int V)
    {
        this.V = V; this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];          // Create array of lists.
        for (int v = 0; v < V; v++)                  // Initialize all lists
            adj[v] = new Bag<Integer>();             // to empty.
    }

    public Graph(In in)
    {
        this(in.readInt());                      // Read V and construct this graph.
        int E = in.readInt();                    // Read E.
        for (int i = 0; i < E; i++)
        { // Add an edge.
            int v = in.readInt();               // Read a vertex,
            int w = in.readInt();               // read another vertex,
            addEdge(v, w);                   // and add edge connecting them.
        }
    }

    public int V() { return V; }
    public int E() { return E; }

    public void addEdge(int v, int w)
    {
        adj[v].add(w);                         // Add w to v's list.
        adj[w].add(v);                         // Add v to w's list.
        E++;
    }

    public Iterable<Integer> adj(int v)
    { return adj[v]; }
}
```

This `Graph` implementation maintains a vertex-indexed array of lists of integers. Every edge appears twice: if an edge connects  $v$  and  $w$ , then  $w$  appears in  $v$ 's list and  $v$  appears in  $w$ 's list. The second constructor reads a graph from an input stream, in the format  $V$  followed by  $E$  followed by a list of pairs of `int` values between 0 and  $V-1$ . See page 523 for `toString()`.

IT IS CERTAINLY REASONABLE to contemplate other operations that might be useful in applications, and to consider methods for

- Adding a vertex
- Deleting a vertex

One way to handle such operations is to expand the API and use a symbol table (ST) instead of a vertex-indexed array (with this change we also do not need our convention that vertex names be integer indices). We might also consider methods for

- Deleting an edge
- Checking whether the graph contains the edge  $v-w$

To implement these two operations (and disallow parallel edges) we might use a SET instead of a Bag for adjacency lists. We refer to this alternative as an *adjacency set* representation. We do not use either of these two alternatives in this book for several reasons:

- Our clients do not need to add vertices, delete vertices and edges, or check whether an edge exists.
- When clients do need these operations, they typically are invoked infrequently or for short adjacency lists, so an easy option is to use a brute-force implementation that iterates through an adjacency list.
- The SET and ST representations slightly complicate algorithm implementation code, diverting attention from the algorithms themselves.
- A performance penalty of  $\log V$  is involved in some situations.

It is not difficult to adapt our algorithms to accommodate other designs (for example disallowing parallel edges or self-loops) without undue performance penalties. The table below summarizes performance characteristics of the alternatives that we have mentioned. Typical applications process huge sparse graphs, so we use the adjacency-lists representation throughout.

underlying data structure	space	add edge $v-w$	check whether $w$ is adjacent to $v$	iterate through vertices adjacent to $v$
<i>list of edges</i>	$E$	1	$E$	$E$
<i>adjacency matrix</i>	$V^2$	1	1	$V$
<i>adjacency lists</i>	$E + V$	1	$degree(v)$	$degree(v)$
<i>adjacency sets</i>	$E + V$	$\log V$	$\log V$	$degree(v)$

Order-of-growth performance for typical Graph implementations

**Design pattern for graph processing.** Since we consider a large number of graph-processing algorithms, our initial design goal is to decouple our implementations from the graph representation. To do so, we develop, for each given task, a task-specific class so that clients can create objects to perform the task. Generally, the constructor does some preprocessing to build data structures so as to efficiently respond to client queries. A typical client program builds a graph, passes that graph to an algorithm implementation class (as argument to a constructor), and then calls client query methods to learn various properties of the graph. As a warmup, consider this API:

```
public class Search
    Search(Graph G, int s) find vertices connected to a source vertex s
    boolean marked(int v) is v connected to s?
    int count() how many vertices are connected to s?


---


    Graph-processing API (warmup)
```

We use the term *source* to distinguish the vertex provided as argument to the constructor from the other vertices in the graph. In this API, the job of the constructor is to find the vertices in the graph that are connected to the source. Then client code calls the instance methods `marked()` and `count()` to learn characteristics of the graph. The name `marked()` refers to an approach used by the basic algorithms that we consider throughout this chapter: they follow paths from the source to other vertices in the graph, marking each vertex encountered. The example client `TestSearch` shown on the facing page takes an input stream name and a source vertex number from the command line, reads a graph from the input stream (using the second `Graph` constructor), builds a `Search` object for the given graph and source, and uses `marked()` to print the vertices in that graph that are connected to the source. It also calls `count()` and prints whether or not the graph is connected (the graph is connected if and only if the search marked all of its vertices).

WE HAVE ALREADY SEEN one way to implement the Search API: the union-find algorithms of CHAPTER 1. The constructor can build a UF object, do a `union()` operation for each of the graph's edges, and implement `marked(v)` by calling `connected(s, v)`. Implementing `count()` requires using a weighted UF implementation and extending its API to use a `count()` method that returns `wt[find(v)]` (see EXERCISE 4.1.8). This implementation is simple and efficient, but the implementation that we consider next is even simpler and more efficient. It is based on *depth-first search*, a fundamental recursive method that follows the graph's edges to find the vertices connected to the source. Depth-first search is the basis for several of the graph-processing algorithms that we consider throughout this chapter.

```
public class TestSearch
{
    public static void main(String[] args)
    {
        Graph G = new Graph(new In(args[0]));
        int s = Integer.parseInt(args[1]);
        Search search = new Search(G, s);

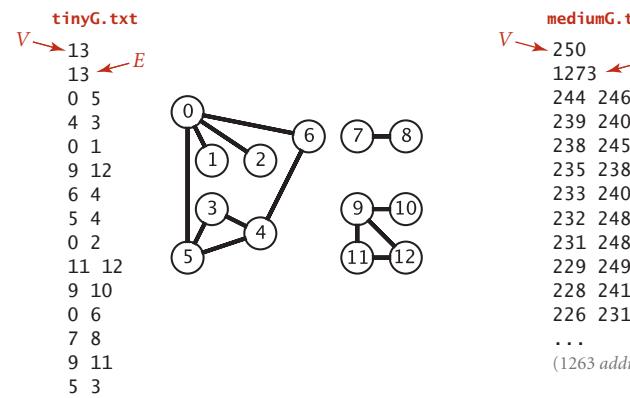
        for (int v = 0; v < G.V(); v++)
            if (search.marked(v))
                StdOut.print(v + " ");
        StdOut.println();

        if (search.count() != G.V())
            StdOut.print("NOT ");
        StdOut.println("connected");
    }
}
```

Sample graph-processing client (warmup)

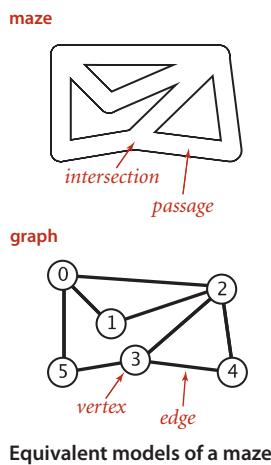
```
% java TestSearch tinyG.txt 0
0 1 2 3 4 5 6
NOT connected

% java TestSearch tinyG.txt 9
9 10 11 12
NOT connected
```



**Depth-first search** We often learn properties of a graph by systematically examining each of its vertices and each of its edges. Determining some simple graph properties—for example, computing the degrees of all the vertices—is easy if we just examine each edge (in any order whatever). But many other graph properties are related to paths, so a natural way to learn them is to move from vertex to vertex along the graph’s

edges. Nearly all of the graph-processing algorithms that we consider use this same basic abstract model, albeit with various different strategies. The simplest is a classic method that we now consider.

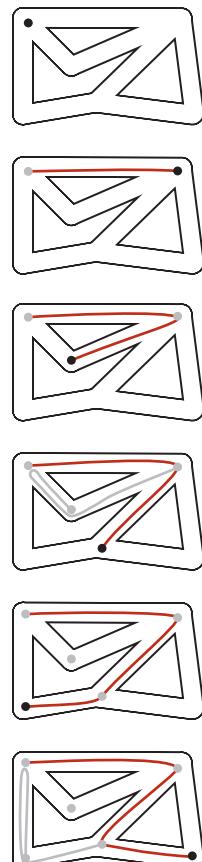


**Searching in a maze.** It is instructive to think about the process of searching through a graph in terms of an equivalent problem that has a long and distinguished history—finding our way through a maze that consists of passages connected by intersections. Some mazes can be handled with a simple rule, but most mazes require a more sophisticated strategy. Using the terminology *maze* instead of *graph*, *passage* instead of *edge*, and *intersection* instead of *vertex* is making mere semantic distinctions, but, for the moment, doing so will help to

give us an intuitive feel for the problem. One trick for exploring a maze without getting lost that has been known since antiquity (dating back at least to the legend of Theseus and the Minotaur) is known as *Tremaux exploration*. To explore all passages in a maze:

- Take any unmarked passage, unrolling a string behind you.
- Mark all intersections and passages when you first visit them.
- Retrace steps (using the string) when approaching a marked intersection.
- Retrace steps when no unvisited options remain at an intersection encountered while retracing steps.

The string guarantees that you can always find a way out and the marks guarantee that you avoid visiting any passage or intersection twice. Knowing that you have explored the whole maze demands a more complicated argument that is better approached in the context of graph search. Tremaux exploration is an intuitive starting point, but it differs in subtle ways from exploring a graph, so we now move on to searching in graphs.



Tremaux exploration

**Warmup.** The classic recursive method for searching in a connected graph (visiting all of its vertices and edges) mimics Tremaux maze exploration but is even simpler to describe. To search a graph, invoke a recursive method that visits vertices. To visit a vertex:

- Mark it as having been visited.
- Visit (recursively) all the vertices that are adjacent to it and that have not yet been marked.

This method is called *depth-first search* (DFS). An implementation of our Search API using this method is shown at right. It maintains an array of boolean values to mark all of the vertices that are connected to the source. The recursive method marks the given vertex and calls itself for any unmarked vertices on its adjacency list. If the graph is connected, every adjacency-list entry is checked.

```
public class DepthFirstSearch
{
    private boolean[] marked;
    private int count;

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        count++;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

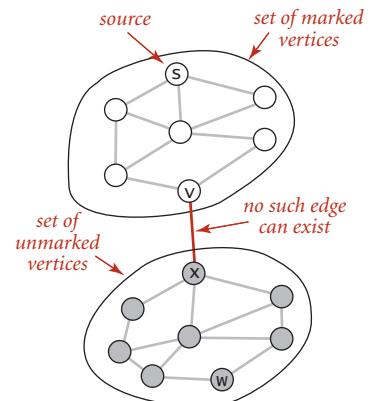
    public boolean marked(int w)
    { return marked[w]; }

    public int count()
    { return count; }
}
```

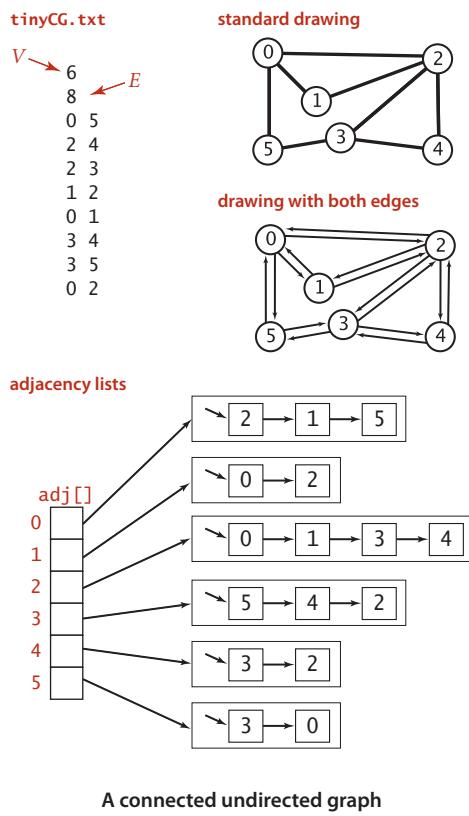
Depth-first search

**Proposition A.** DFS marks all the vertices connected to a given source in time proportional to the sum of their degrees.

**Proof:** First, we prove that the algorithm marks all the vertices connected to the source  $s$  (and no others). Every marked vertex is connected to  $s$ , since the algorithm finds vertices only by following edges. Now, suppose that some unmarked vertex  $w$  is connected to  $s$ . Since  $s$  itself is marked, any path from  $s$  to  $w$  must have at least one edge from the set of marked vertices to the set of unmarked vertices, say  $v-x$ . But the algorithm would have discovered  $x$  after marking  $v$ , so no such edge can exist, a contradiction. The time bound follows because marking ensures that each vertex is visited once (taking time proportional to its degree to check marks).



**One-way passages.** The method call–return mechanism in the program corresponds to the string in the maze: when we have processed all the edges incident to a vertex (explored all the passages leaving an intersection), we “return” (in both senses of the word). To draw a proper correspondence with Tremaux exploration of a maze, we need to imagine a maze constructed entirely of one-way passages (one in each direction).



In the same way that we encounter each passage in the maze twice (once in each direction), we encounter each edge in the graph *twice* (once at each of its vertices). In Tremaux exploration, we either explore a passage for the first time or return along it from a marked vertex; in DFS of an undirected graph, we either do a recursive call when we encounter an edge  $v-w$  (if  $w$  is not marked) or skip the edge (if  $w$  is marked). The second time that we encounter the edge, in the opposite orientation  $w-v$ , we always ignore it, because the destination vertex  $v$  has certainly already been visited (the first time that we encountered the edge).

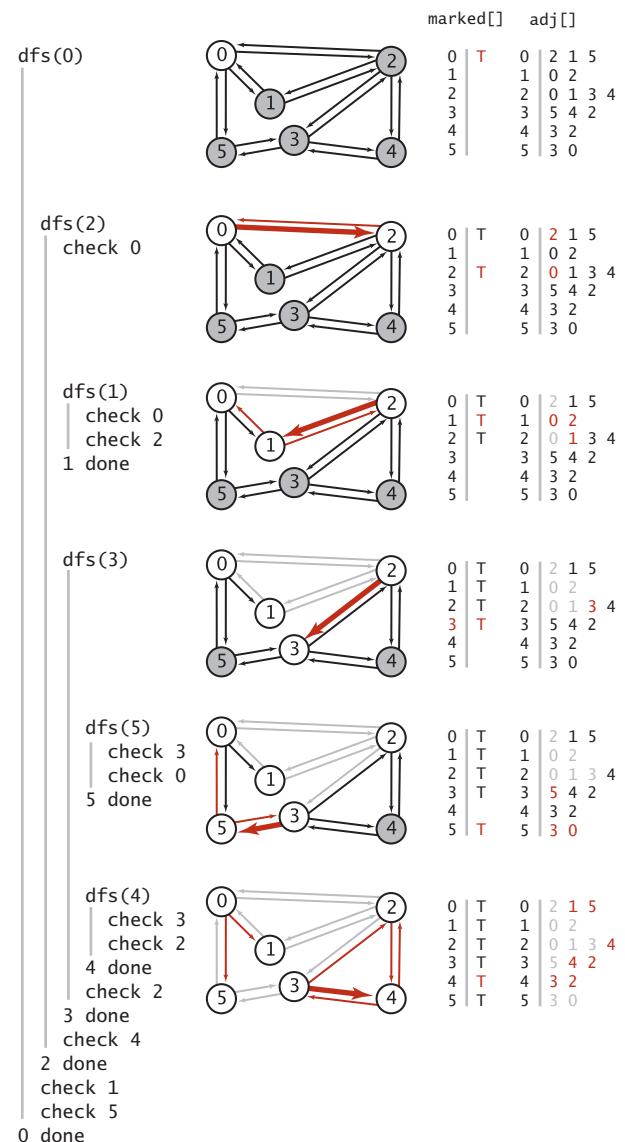
**Tracing DFS.** As usual, one good way to understand an algorithm is to trace its behavior on a small example. This is particularly true of depth-first search. The first thing to bear in mind when doing a trace is that the order in which edges are examined and vertices visited depends upon the *representation*, not just the graph or the algorithm. Since DFS only examines vertices connected to the source, we use the small connected graph depicted at left as an example for traces.

In this example, vertex 2 is the first vertex visited

after 0 because it happens to be first on 0’s adjacency list. The second thing to bear in mind when doing a trace is that, as mentioned above, DFS traverses each edge in the graph twice, always finding a marked vertex the second time. One effect of this observation is that tracing a DFS takes twice as long as you might think! Our example graph has only eight edges, but we need to trace the action of the algorithm on the 16 entries on the adjacency lists.

**Detailed trace of depth-first search.** The figure at right shows the contents of the data structures just after each vertex is marked for our small example, with source 0. The search begins when the constructor calls the recursive `dfs()` to mark and visit vertex 0 and proceeds as follows:

- Since 2 is first on 0's adjacency list and is unmarked, `dfs()` recursively calls itself to mark and visit 2 (in effect, the system puts 0 and the current position on 0's adjacency list on a stack).
- Now, 0 is first on 2's adjacency list and is marked, so `dfs()` skips it. Then, since 1 is next on 2's adjacency list and is unmarked, `dfs()` recursively calls itself to mark and visit 1.
- Visiting 1 is different: since both vertices on its list (0 and 2) are already marked, no recursive calls are needed, and `dfs()` returns from the recursive call `dfs(1)`. The next edge examined is 2-3 (since 3 is the vertex after 1 on 2's adjacency list), so `dfs()` recursively calls itself to mark and visit 3.
- Vertex 5 is first on 3's adjacency list and is unmarked, so `dfs()` recursively calls itself to mark and visit 5.
- Both vertices on 5's list (3 and 0) are already marked, so no recursive calls are needed,
- Vertex 4 is next on 3's adjacency list and is unmarked, so `dfs()` recursively calls itself to mark and visit 4, the last vertex to be marked.
- After 4 is marked, `dfs()` needs to check the vertices on its list, then the remaining vertices on 3's list, then 2's list, then 0's list, but no more recursive calls happen because all vertices are marked.



Trace of depth-first search to find vertices connected to 0

THIS BASIC RECURSIVE SCHEME IS JUST A START—depth-first search is effective for many graph-processing tasks. For example, in this section, we consider the use of depth-first search to address a problem that we first posed in CHAPTER 1:

**Connectivity.** Given a graph, support queries of the form *Are two given vertices connected?* and *How many connected components does the graph have?*

This problem is easily solved within our standard graph-processing design pattern, and we will compare and contrast this solution with the union-find algorithms that we considered in SECTION 1.5.

The question “Are two given vertices connected?” is equivalent to the question “Is there a path connecting two given vertices?” and might be named the *path detection* problem. However, the union-find data structures that we considered in SECTION 1.5 do not address the problems of *finding* such a path. Depth-first search is the first of several approaches that we consider to solve this problem, as well:

**Single-source paths.** Given a graph and a source vertex  $s$ , support queries of the form *Is there a path from  $s$  to a given target vertex  $v$ ? If so, find such a path.*

DFS is deceptively simple because it is based on a familiar concept and is so easy to implement; in fact, it is a subtle and powerful algorithm that researchers have learned to put to use to solve numerous difficult problems. These two are the first of several that we will consider.

**Finding paths** The single-source paths problem is fundamental to graph processing. In accordance with our standard design pattern, we use the following API:

```
public class Paths
    Paths(Graph G, int s) find paths in G from source s
    boolean hasPathTo(int v) is there a path from s to v?
    Iterable<Integer> pathTo(int v) path from s to v; null if no such path
    API for paths implementations
```

The constructor takes a source vertex  $s$  as argument and computes paths from  $s$  to each vertex connected to  $s$ . After creating a `Paths` object for a source  $s$ , the client can use the instance method `pathTo()` to iterate through the vertices on a path from  $s$  to any vertex connected to  $s$ . For the moment, we accept any path; later, we shall develop implementations that find paths having certain properties. The test client at right takes a graph from the input stream and a source from the command line and prints a path from the source to each vertex connected to it.

```
public static void main(String[] args)
{
    Graph G = new Graph(new In(args[0]));
    int s = Integer.parseInt(args[1]);
    Paths search = new Paths(G, s);
    for (int v = 0; v < G.V(); v++)
    {
        StdOut.print(s + " to " + v + ": ");
        if (search.hasPathTo(v))
            for (int x : search.pathTo(v))
                if (x == s) StdOut.print(x);
                else StdOut.print("-" + x);
        StdOut.println();
    }
}
```

Test client for `paths` implementations

**Implementation.** ALGORITHM 4.1 onpage 536 is a DFS-based implementation of `Paths` that extends the `DepthFirstSearch` warmup onpage 531 by adding as an instance variable an array `edgeTo[]` of `int` values that serves the purpose of the ball of string in Tremaux exploration: it gives a way to find a path back to  $s$  for every vertex connected to  $s$ . Instead of just keeping track of the path from the current vertex back to the start,

we remember a path from *each* vertex to the start. To accomplish this, we remember the edge  $v-w$  that takes us to each vertex  $w$  *for the first time*, by setting `edgeTo[w]` to  $v$ . In other words,  $v-w$  is the last edge on the known path from  $s$  to  $w$ . The result of the search is a tree rooted at the source; `edgeTo[]` is a parent-link representation of that tree. A small example is drawn to

```
% java Paths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-2-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-3-4
0 to 5: 0-2-3-5
```

**ALGORITHM 4.1** Depth-first search to find paths in a graph

```

public class DepthFirstPaths
{
    private boolean[] marked; // Has dfs() been called for this vertex?
    private int[] edgeTo; // last vertex on known path to this vertex
    private final int s; // source

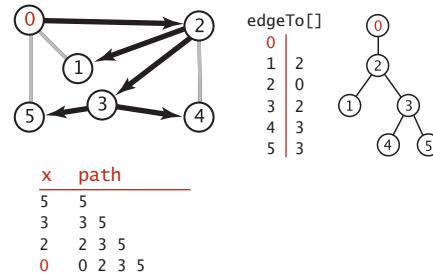
    public DepthFirstPaths(Graph G, int s)
    {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
            {
                edgeTo[w] = v;
                dfs(G, w);
            }
    }

    public boolean hasPathTo(int v)
    {
        return marked[v];
    }

    public Iterable<Integer> pathTo(int v)
    {
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        for (int x = v; x != s; x = edgeTo[x])
            path.push(x);
        path.push(s);
        return path;
    }
}

```



This Graph client uses depth-first search to find paths to all the vertices in a graph that are connected to a given start vertex  $s$ . Code from `DepthFirstSearch` (page 531) is printed in gray. To save known paths to each vertex, this code maintains a vertex-indexed array `edgeTo[]` such that `edgeTo[w] = v` means that  $v-w$  was the edge used to access  $w$  for the first time. The `edgeTo[]` array is a parent-link representation of a tree rooted at  $s$  that contains all the vertices connected to  $s$ .

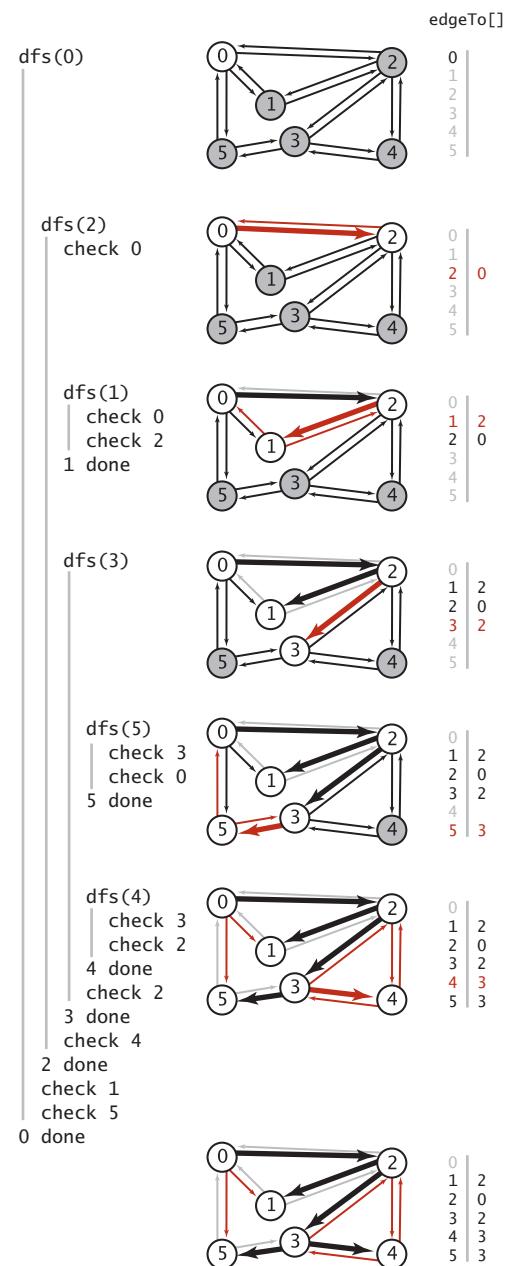
the right of the code in ALGORITHM 4.1. To recover the path from  $s$  to any vertex  $v$ , the `pathTo()` method in ALGORITHM 4.1 uses a variable  $x$  to travel up the tree, setting  $x$  to `edgeTo[x]`, just as we did for union-find in SECTION 1.5, putting each vertex encountered onto a stack until reaching  $s$ . Returning the stack to the client as an `Iterable` enables the client to follow the path from  $s$  to  $v$ .

**Detailed trace.** The figure at right shows the contents of `edgeTo[]` just after each vertex is marked for our example, with source 0. The contents of `marked[]` and `adj[]` are the same as in the trace of `DepthFirstSearch` on page 533, as is the detailed description of the recursive calls and the edges checked, so these aspects of the trace are omitted. The depth-first search adds the edges 0-2, 2-1, 2-3, 3-5, and 3-4 to `edgeTo[]`, in that order. These edges form a tree rooted at the source and provide the information needed for `pathTo()` to provide for the client the path from 0 to 1, 2, 3, 4, or 5, as just described.

THE CONSTRUCTOR in `DepthFirstPaths` differs only in a few assignment statements from the constructor in `DepthFirstSearch`, so PROPOSITION A on page 531 applies. In addition, we have:

**Proposition A (continued).** DFS allows us to provide clients with a path from a given source to any marked vertex in time proportional its length.

**Proof:** By induction on the number of vertices visited, it follows that the `edgeTo[]` array in `DepthFirstPaths` represents a tree rooted at the source. The `pathTo()` method builds the path in time proportional to its length.



Trace of depth-first search to find all paths from 0

**Breadth-first search** The paths discovered by depth-first search depend not just on the graph, but also on the representation and the nature of the recursion. Naturally, we are often interested in solving the following problem:

**Single-source shortest paths.** Given a graph and a source vertex  $s$ , support queries of the form *Is there a path from  $s$  to a given target vertex  $v$ ?* If so, find a *shortest* such path (one with a minimal number of edges).

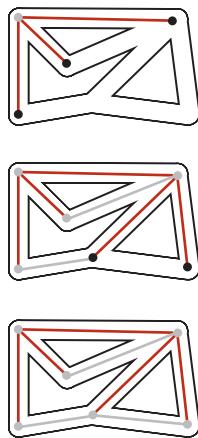
The classical method for accomplishing this task, called *breadth-first search* (BFS), is also the basis of numerous algorithms for processing graphs, so we consider it in detail in this section. DFS offers us little assistance in solving this problem, because the order

in which it takes us through the graph has no relationship to the goal of finding shortest paths. In contrast, BFS is based on this goal. To find a shortest path from  $s$  to  $v$ , we start at  $s$  and check for  $v$  among all the vertices that we can reach by following one edge, then we check for  $v$  among all the vertices that we can reach from  $s$  by following two edges, and so forth. DFS is analogous to one person exploring a maze. BFS is analogous to a group of searchers exploring by fanning out in all directions, each unrolling his or her own ball of string. When more than one passage needs to be explored, we imagine that the searchers split up to explore all of them; when two groups of searchers meet up, they join forces (using the ball of string held by the one getting there first).

In a program, when we come to a point during a graph search where we have more than one edge to traverse, we choose one and save the others to be explored later. In DFS, we use a pushdown stack (that is managed by the system to support the recursive search method) for this purpose. Using the LIFO rule that characterizes the pushdown stack corresponds to exploring passages that are close by in a maze. We choose, of the passages yet to be explored, the one that was most recently encountered. In BFS, we want to explore the vertices in order of their distance from the source. It turns out that this order is easily arranged: use a (FIFO) queue instead of a (LIFO) stack. We choose, of the passages yet to be explored, the one that was least recently encountered.

**Implementation.** ALGORITHM 4.2 on page 540 is an implementation of BFS. It is based on maintaining a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, then perform the following steps until the queue is empty:

- Remove the next vertex  $v$  from the queue.
- Put onto the queue all unmarked vertices that are adjacent to  $v$  and mark them.

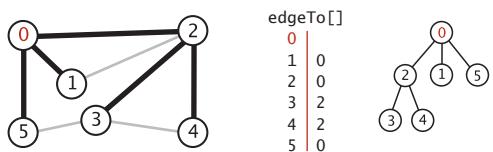


Breadth-first  
maze exploration

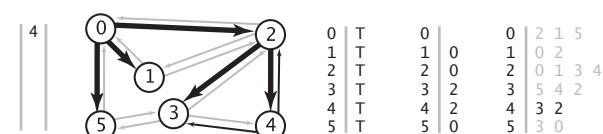
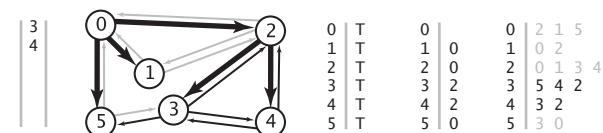
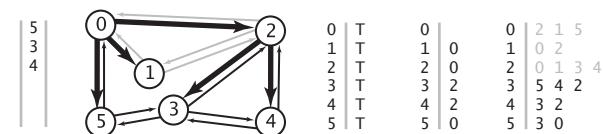
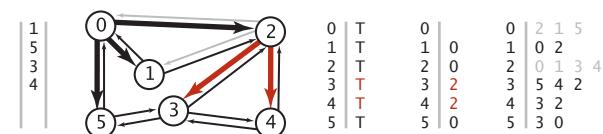
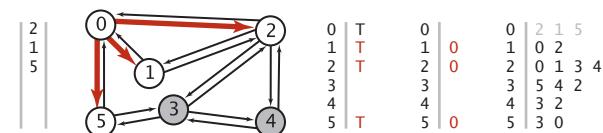
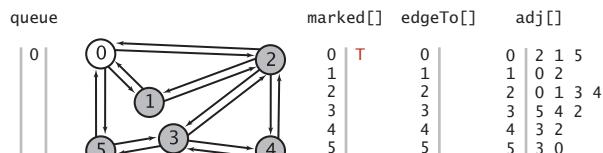
The `bfs()` method in ALGORITHM 4.2 is *not* recursive. Instead of the implicit stack provided by recursion, it uses an explicit queue. The product of the search, as for DFS, is an array `edgeTo[]`, a parent-link representation of a tree rooted at `s`, which defines the shortest paths from `s` to every vertex that is connected to `s`. The paths can be constructed for the client using the same `pathTo()` implementation that we used for DFS in ALGORITHM 4.1.

The figure at right shows the step-by-step development of BFS on our sample graph, showing the contents of the data structures at the beginning of each iteration of the loop. Vertex 0 is put on the queue, then the loop completes the search as follows:

- Removes 0 from the queue and puts its adjacent vertices 2, 1, and 5 on the queue, marking each and setting the `edgeTo[]` entry for each to 0.
- Removes 2 from the queue, checks its adjacent vertices 0 and 1, which are marked, and puts its adjacent vertices 3 and 4 on the queue, marking each and setting the `edgeTo[]` entry for each to 2.
- Removes 1 from the queue and checks its adjacent vertices 0 and 2, which are marked.
- Removes 5 from the queue and checks its adjacent vertices 3 and 0, which are marked.
- Removes 3 from the queue and checks its adjacent vertices 5, 4, and 2, which are marked.
- Removes 4 from the queue and checks its adjacent vertices 3 and 2, which are marked.



Outcome of breadth-first search to find all paths from 0



Trace of breadth-first search to find all paths from 0

### ALGORITHM 4.2 Breadth-first search to find paths in a graph

```

public class BreadthFirstPaths
{
    private boolean[] marked; // Is a shortest path to this vertex known?
    private int[] edgeTo; // last vertex on known path to this vertex
    private final int s; // source

    public BreadthFirstPaths(Graph G, int s)
    {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        bfs(G, s);
    }

    private void bfs(Graph G, int s)
    {
        Queue<Integer> queue = new Queue<Integer>();
        marked[s] = true; // Mark the source
        queue.enqueue(s); // and put it on the queue.
        while (!queue.isEmpty())
        {
            int v = queue.dequeue(); // Remove next vertex from the queue.
            for (int w : G.adj(v))
                if (!marked[w]) // For every unmarked adjacent vertex,
                {
                    edgeTo[w] = v; // save last edge on a shortest path,
                    marked[w] = true; // mark it because path is known,
                    queue.enqueue(w); // and add it to the queue.
                }
        }
    }

    public boolean hasPathTo(int v)
    { return marked[v]; }

    public Iterable<Integer> pathTo(int v)
    // Same as for DFS (see page 536).
    }
}

```

This Graph client uses breadth-first search to find paths in a graph with the fewest number of edges from the source  $s$  given in the constructor. The  $bfs()$  method marks all vertices connected to  $s$ , so clients can use  $hasPathTo()$  to determine whether a given vertex  $v$  is connected to  $s$  and  $pathTo()$  to get a path from  $s$  to  $v$  with the property that no other such path from  $s$  to  $v$  has fewer edges.

For this example, the `edgeTo[]` array is complete after the second step. As with DFS, once all vertices have been marked, the rest of the computation is just checking edges to vertices that have already been marked.

**Proposition B.** For any vertex  $v$  reachable from  $s$ , BFS computes a shortest path from  $s$  to  $v$  (no path from  $s$  to  $v$  has fewer edges).

**Proof:** It is easy to prove by induction that the queue always consists of zero or more vertices of distance  $k$  from the source, followed by zero or more vertices of distance  $k+1$  from the source, for some integer  $k$ , starting with  $k$  equal to 0. This property implies, in particular, that vertices enter and leave the queue in order of their distance from  $s$ . When a vertex  $v$  enters the queue, no shorter path to  $v$  will be found before it comes off the queue, and no path to  $v$  that is discovered after it comes off the queue can be shorter than  $v$ 's tree path length.

**Proposition B (continued).** BFS takes time proportional to  $V+E$  in the worst case.

**Proof:** As for PROPOSITION A (page 531), BFS marks all the vertices connected to  $s$  in time proportional to the sum of their degrees. If the graph is connected, this sum equals the sum of the degrees of all the vertices, or  $2E$ . Initializing the `marked[]` and `edgeTo[]` arrays takes time proportional to  $V$ .

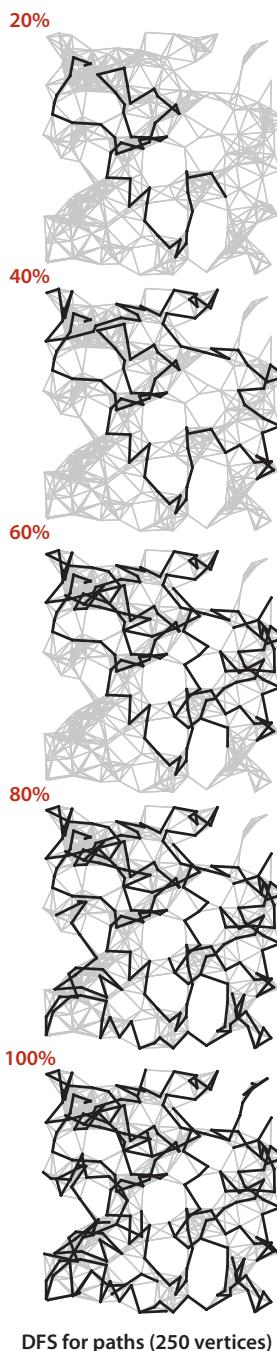
Note that we can also use BFS to implement the Search API that we implemented with DFS, since the solution depends on only the ability of the search to examine every vertex and edge connected to the source.

As implied at the outset, DFS and BFS are the first of several instances that we will examine of a general approach to searching graphs. We put the source vertex on the data structure, then perform the following steps until the data structure is empty:

- Take the next unmarked vertex  $v$  from the data structure and mark it.
- Put onto the data structure all unmarked vertices that are adjacent to  $v$ .

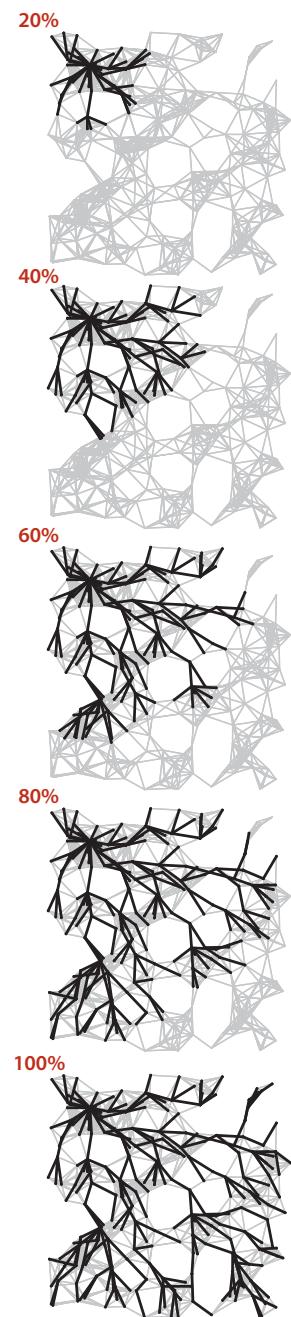
The algorithms differ only in the rule used to take the next vertex from the data structure (least recently added for BFS, most recently added for DFS). This difference leads to completely different views of the graph, even though all the

```
% java BreadthFirstPaths tinyCG.txt 0
0 to 0: 0
0 to 1: 0-1
0 to 2: 0-2
0 to 3: 0-2-3
0 to 4: 0-2-4
0 to 5: 0-5
```



vertices and edges connected to the source are examined no matter what rule is used. Our implementations of BFS and DFS gain efficiency over the general approach by eagerly marking vertices as they are added to the data structure (BFS) and lazily adding unmarked vertices to the data structure (DFS).

THE DIAGRAMS ON EITHER SIDE of this page, which show the progress of DFS and BFS for our sample graph `mediumG.txt`, make plain the differences between the paths that are discovered by the two approaches. DFS wends its way through the graph, storing on the stack the points where other paths branch off; BFS sweeps through the graph, using a queue to remember the frontier of visited places. DFS explores the graph by looking for new vertices far away from the start point, taking closer vertices only when dead ends are encountered; BFS completely covers the area close to the starting point, moving farther away only when everything nearby has been examined. DFS paths tend to be long and winding; BFS paths are short and direct. Depending upon the application, one property or the other may be desirable (or properties of paths may be immaterial). In SECTION 4.4, we will be considering other implementations of the Paths API that find paths having other specified properties.



**Connected components** Our next direct application of depth-first search is to find the connected components of a graph. Recall from SECTION 1.5 (see page 216) that “is connected to” is an *equivalence relation* that divides the vertices into *equivalence classes* (the connected components). For this common graph-processing task, we define the following API:

public class CC	
CC(Graph G)	<i>preprocessing constructor</i>
boolean connected(int v, int w)	<i>are v and w connected?</i>
int count()	<i>number of connected components</i>
int id(int v)	<i>component identifier for v (between 0 and count()-1)</i>

API for connected components

The `id()` method is for client use in indexing an array by component, as in the test client below, which reads a graph and then prints its number of connected components and then the vertices in each component, one component per line. To do so, it builds an array of `Queue` objects, then uses each vertex’s component identifier as an index into this array, to add the vertex to the appropriate `Queue`. This client is a model for the typical situation where we want to independently process connected components.

**Implementation.** The implementation `CC` (ALGORITHM 4.3 on the next page) uses our `marked[]` array to find a vertex to serve as the starting point for a depth-first search in each component. The first call to the recursive DFS is for vertex 0—it marks all vertices connected to 0. Then the `for` loop in the constructor looks for an unmarked vertex and calls the recursive `dfs()` to mark all vertices connected to that vertex. Moreover, it maintains a vertex-indexed array `id[]` that associates the same `int` value to every vertex

```
public static void main(String[] args)
{
    Graph G = new Graph(new In(args[0]));
    CC cc = new CC(G);

    int M = cc.count();
    StdOut.println(M + " components");

    Queue<Integer>[] components;
    components = (Queue<Integer>[]) new Queue[M];
    for (int i = 0; i < M; i++)
        components[i] = new Queue<Integer>();
    for (int v = 0; v < G.V(); v++)
        components[cc.id(v)].enqueue(v);
    for (int i = 0; i < M; i++)
    {
        for (int v: components[i])
            StdOut.print(v + " ");
        StdOut.println();
    }
}
```

Test client for connected components API

**ALGORITHM 4.3** Depth-first search to find connected components in a graph

```

public class CC
{
    private boolean[] marked;
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int s = 0; s < G.V(); s++)
            if (!marked[s])
                {
                    dfs(G, s);
                    count++;
                }
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean connected(int v, int w)
    { return id[v] == id[w]; }

    public int id(int v)
    { return id[v]; }

    public int count()
    { return count; }
}

```

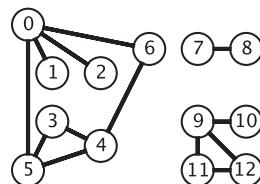
```

% java Graph tinyG.txt
13 vertices, 13 edges
0: 6 2 1 5
1: 0
2: 0
3: 5 4
4: 5 6 3
5: 3 4 0
6: 0 4
7: 8
8: 7
9: 11 10 12
10: 9
11: 9 12
12: 11 9

% java CC tinyG.txt
3 components
0 1 2 3 4 5 6
7 8
9 10 11 12

```

This Graph client provides its clients with the ability to independently process a graph's connected components. Code from `DepthFirstSearch` (page 531) is left in gray. The computation is based on a vertex-indexed array `id[]` such that `id[v]` is set to  $i$  if  $v$  is in the  $i$ th connected component processed. The constructor finds an unmarked vertex and calls the recursive `dfs()` to mark and identify all the vertices connected to it, continuing until all vertices have been marked and identified. Implementations of the instance methods `connected()`, `id()`, and `count()` are immediate.

**tinyG.txt**

		marked[]										id[]															
		0	1	2	3	4	5	6	7	8	9	10	11	12	0	1	2	3	4	5	6	7	8	9	10	11	12

dfs(0)	0	T												0											
dfs(6)	0	T												0											
check 0																									
dfs(4)	0	T												0											
dfs(5)	0	T												0											
dfs(3)	0	T												0											
check 5																									
check 4																									
3 done																									
check 4																									
check 0																									
5 done																									
check 6																									
check 3																									
4 done																									
6 done																									
dfs(2)	0	T												0											
check 0																									
2 done																									
dfs(1)	0	T												0											
check 0																									
1 done																									
check 5																									
0 done																									
dfs(7)	1	T	T	T	T	T	T	T	T	T				0	0	0	0	0	0	0	1				
dfs(8)	1	T	T	T	T	T	T	T	T	T	T			0	0	0	0	0	0	0	1	1			
check 7																									
8 done																									
7 done																									
dfs(9)	2	T	T	T	T	T	T	T	T	T	T	T		0	0	0	0	0	0	0	1	1	2		
dfs(11)	2	T	T	T	T	T	T	T	T	T	T	T	T	0	0	0	0	0	0	0	1	1	2		
check 9																									
dfs(12)	2	T	T	T	T	T	T	T	T	T	T	T	T	T	0	0	0	0	0	0	0	1	1	2	
check 11																									
check 9																									
12 done																									
11 done																									
dfs(10)	2	T	T	T	T	T	T	T	T	T	T	T	T	T	0	0	0	0	0	0	0	1	1	2	2
check 9																									
10 done																									
check 12																									
9 done																									

Trace of depth-first search to find connected components

in each component. This array makes the implementation of `connected()` simple, in precisely the same manner as `connected()` in SECTION 1.5 (just check if identifiers are equal). In this case, the identifier 0 is assigned to all the vertices in the first component processed, 1 is assigned to all the vertices in the second component processed, and so forth, so that the identifiers are all between 0 and `count()-1`, as specified in the API. This convention enables the use of component-indexed arrays, as in the test client on page 543.

**Proposition C.** DFS uses preprocessing time and space proportional to  $V+E$  to support constant-time connectivity queries in a graph.

**Proof:** Immediate from the code. Each adjacency-list entry is examined exactly once, and there are  $2E$  such entries (two for each edge); initializing the `marked[]` and `id[]` arrays takes time proportional to  $V$ . Instance methods examine or return one or two instance variables.

**Union-find.** How does the DFS-based solution for graph connectivity in CC compare with the union-find approach of CHAPTER 1? In theory, DFS is faster than union-find because it provides a constant-time guarantee, which union-find does not; in practice, this difference is negligible, and union-find is faster because it does not have to build a full representation of the graph. More important, union-find is an online algorithm (we can check whether two vertices are connected in near-constant time at any point, even while adding edges), whereas the DFS solution must first preprocess the graph. Therefore, for example, we prefer union-find when determining connectivity is our only task or when we have a large number of queries intermixed with edge insertions but may find the DFS solution more appropriate for use in a graph ADT because it makes efficient use of existing infrastructure.

THE PROBLEMS THAT WE HAVE SOLVED with DFS are fundamental. It is a simple approach, and recursion provides us a way to reason about the computation and develop compact solutions to graph-processing problems. Two additional examples, for solving the following problems, are given in the table on the facing page.

**Cycle detection.** Support this query: *Is a given graph acyclic?*

**Two-colorability.** Support this query: *Can the vertices of a given graph be assigned one of two colors in such a way that no edge connects vertices of the same color?* which is equivalent to this question: *Is the graph bipartite?*

As usual with DFS, the simple code masks a more sophisticated computation, so studying these examples, tracing their behavior on small sample graphs, and extending them to provide a cycle or a coloring, respectively, are worthwhile (and left for exercises).

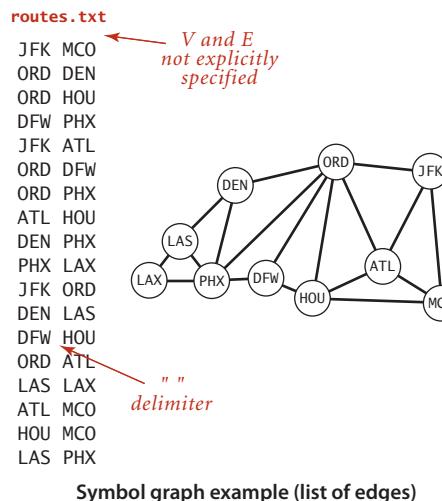
task	implementation
<i>is G acyclic?</i> <i>(assumes</i> <i>no self-loops or</i> <i>parallel edges)</i>	<pre> public class Cycle {     private boolean[] marked;     private boolean hasCycle;      public Cycle(Graph G)     {         marked = new boolean[G.V()];         for (int s = 0; s &lt; G.V(); s++)             if (!marked[s])                 dfs(G, s, s);     }      private void dfs(Graph G, int v, int u)     {         marked[v] = true;         for (int w : G.adj(v))             if (!marked[w])                 dfs(G, w, v);             else if (w != u) hasCycle = true;     }      public boolean hasCycle()     { return hasCycle; } } </pre>
<i>is G bipartite?</i> <i>(two-colorable)</i>	<pre> public class TwoColor {     private boolean[] marked;     private boolean[] color;     private boolean isTwoColorable = true;      public TwoColor(Graph G)     {         marked = new boolean[G.V()];         color = new boolean[G.V()];         for (int s = 0; s &lt; G.V(); s++)             if (!marked[s])                 dfs(G, s);     }      private void dfs(Graph G, int v)     {         marked[v] = true;         for (int w : G.adj(v))             if (!marked[w])             {                 color[w] = !color[v];                 dfs(G, w);             }             else if (color[w] == color[v]) isTwoColorable = false;     }      public boolean isBipartite()     { return isTwoColorable; } } </pre>

More examples of graph processing with DFS

**Symbol graphs** Typical applications involve processing graphs defined in files or on web pages, using strings, not integer indices, to define and refer to vertices. To accommodate such applications, we define an input format with the following properties:

- Vertex names are strings.
- A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
- Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.
- The number of vertices  $V$  and the number of edges  $E$  are both implicitly defined.

Shown below is a small example, the file `routes.txt`, which represents a model for a small transportation system where vertices are U.S. airport codes and edges connecting them are airline routes between the vertices. The file is simply a list of edges. Shown



on the facing page is a larger example, taken from the file `movies.txt`, from the *Internet Movie Database* (IMDB), that we introduced in SECTION 3.5. Recall that this file consists of lines listing a movie name followed by a list of the performers in the movie. In the context of graph processing, we can view it as defining a graph with movies and performers as vertices and each line defining the adjacency list of edges connecting each movie to its performers. Note that the graph is a *bipartite* graph—there are no edges connecting performers to performers or movies to movies.

**API.** The following API defines a `Graph` client that allows us to immediately use our graph-processing routines for graphs defined by such files:

---

```
public class SymbolGraph
{
    SymbolGraph(String filename,
                String delim)           build graph specified in
                                         filename using delim to
                                         separate vertex names

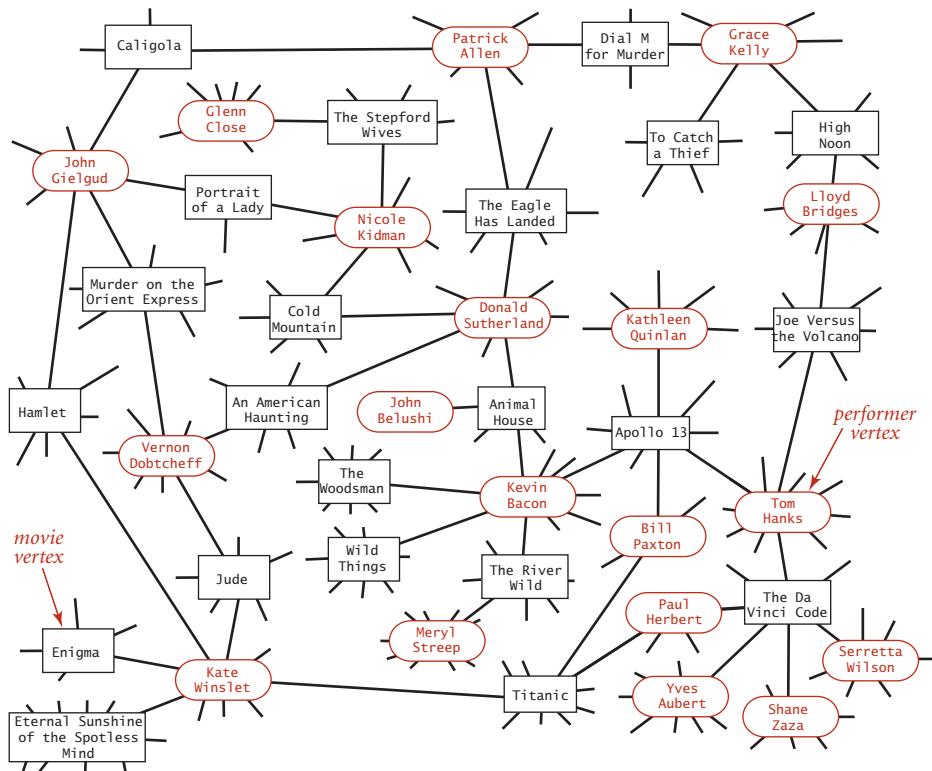
    boolean contains(String key)   is key a vertex?

    int index(String key)         index associated with key

    String name(int v)           key associated with index v

    Graph G()                   underlying Graph
}
```

API for graphs with symbolic vertex names



**movies.txt** *V and E  
not explicitly  
specified*

...

Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara...

Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)...

Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...

Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine ...

To Be or Not to Be (1942)/Verebes, Ernö (I)/.../Lombard, Carole (I)...

To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...

To Catch a Thief (1955)/París, Manuel/.../Grant, Cary/.../Kelly, Grace/...

To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../Tucci, Maria...

...

*movie* *performers*

*"/" delimiter*

## Symbol graph example (adjacency lists)

```

public static void main(String[] args)
{
    String filename = args[0];
    String delim = args[1];
    SymbolGraph sg = new SymbolGraph(filename, delim);

    Graph G = sg.G();
    while (StdIn.hasNextLine())
    {
        String source = StdIn.readLine();
        for (int v : G.adj(sg.index(source)))
            StdOut.println(" " + sg.name(v));
    }
}

```

Test client for symbol graph API

```

% java SymbolGraph routes.txt " "
JFK
ORD
ATL
MCO
LAX
LAS
PHX

```

```

% java SymbolGraph movies.txt "/"
Tin Men (1987)
Hershey, Barbara
Geppi, Cindy
...
Blumenfeld, Alan
DeBoy, David
Bacon, Kevin
Woodsman, The (2004)
Wild Things (1998)
...
Apollo 13 (1995)
Animal House (1978)

```

This API provides a constructor to read and build the graph and client methods `name()` and `index()` for translating vertex names between the strings on the input stream and the integer indices used by our graph-processing methods.

**Test client.** The test client at left builds a graph from the file named as the first command-line argument (using the delimiter as specified by the second command-line ar-

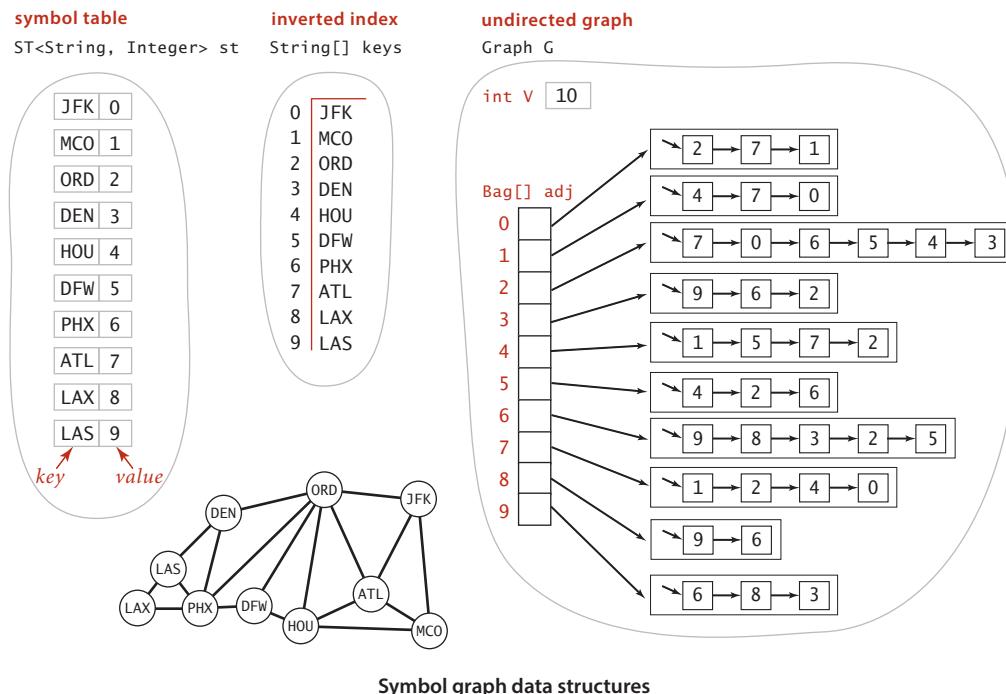
gument) and then takes queries from standard input. The user specifies a vertex name and gets the list of vertices adjacent to that vertex. This client immediately provides the useful inverted index functionality that we considered in SECTION 3.5. In the case of `routes.txt`, you can type an airport code to find the direct flights from that airport, information that is not directly available in the data file. In the case of `movies.txt`, you can type the name of a performer to see the list of the movies in the database in which that performer appeared, or you can type the name of a movie to see the list of performers that appear in that movie. Typing a movie name and getting its cast is not much more than regurgitating the corresponding line in the input file, but typing the name of a performer and getting the list of movies in which that performer has appeared is inverting the index. Even though the database is built around connecting movies to performers, the bipartite graph model embraces the idea that it also connects performers to movies. The bipartite graph model automatically serves as an inverted index and also provides the basis for more sophisticated processing, as we will see.

THIS APPROACH IS CLEARLY EFFECTIVE for any of the graph-processing methods that we consider: any client can use `index()` when it wants to convert a vertex name to an index for use in graph processing and `name()` when it wants to convert an index from graph processing into a name for use in the context of the application.

**Implementation.** A full `SymbolGraph` implementation is given on page 552. It builds three data structures:

- A symbol table `st` with `String` keys (vertex names) and `int` values (indices)
- An array `keys[]` that serves as an inverted index, giving the vertex name associated with each integer index
- A Graph `G` built using the indices to refer to vertices

`SymbolGraph` uses two passes through the data to build these data structures, primarily because the number of vertices  $V$  is needed to build the Graph. In typical real-world applications, keeping the value of  $V$  and  $E$  in the graph definition file (as in our `Graph` constructor at the beginning of this section) is somewhat inconvenient—with `SymbolGraph`, we can maintain files such as `routes.txt` or `movies.txt` by adding or deleting entries without regard to the number of different names involved.



## Symbol graph data type

```

public class SymbolGraph
{
    private ST<String, Integer> st;                      // String -> index
    private String[] keys;                                  // index -> String
    private Graph G;                                      // the graph

    public SymbolGraph(String stream, String sp)
    {
        st = new ST<String, Integer>();                  // First pass
        In in = new In(stream);                           // builds the index
        while (in.hasNextLine())
        {
            String[] a = in.readLine().split(sp);          // by reading strings
            for (int i = 0; i < a.length; i++)              // to associate each
                if (!st.contains(a[i]))                   // distinct string
                    st.put(a[i], st.size());                // with an index.
            }
        keys = new String[st.size()];                     // Inverted index
        for (String name : st.keys())                    // to get string keys
            keys[st.get(name)] = name;                  // is an array.

        G = new Graph(st.size());
        in = new In(stream);                            // Second pass
        while (in.hasNextLine())                       // builds the graph
        {
            String[] a = in.readLine().split(sp);          // by connecting the
            int v = st.get(a[0]);                         // first vertex
            for (int i = 1; i < a.length; i++)             // on each line
                G.addEdge(v, st.get(a[i]));               // to all the others.
            }
        }

        public boolean contains(String s) { return st.contains(s); }
        public int index(String s) { return st.get(s); }
        public String name(int v) { return keys[v]; }
        public Graph G() { return G; }
    }
}

```

This Graph client allows clients to define graphs with `String` vertex names instead of integer indices. It maintains instance variables `st` (a symbol table that maps names to indices), `keys` (an array that maps indices to names), and `G` (a graph, with integer vertex names). To build these data structures, it makes two passes through the graph definition (each line has a string and a list of adjacent strings, separated by the delimiter `sp`).

**Degrees of separation.** One of the classic applications of graph processing is to find the degree of separation between two individuals in a social network. To fix ideas, we discuss this application in terms of a recently popularized pastime known as the *Kevin Bacon game*, which uses the movie-performer graph that we just considered. Kevin Bacon is a prolific actor who has appeared in many movies. We assign every performer a *Kevin Bacon number* as follows: Bacon himself is 0, any performer who has been in the same cast as Bacon has a Kevin Bacon number of 1, any other performer (except Bacon) who has been in the same cast as a performer whose number is 1 has a Kevin Bacon number of 2, and so forth. For example, Meryl Streep has a Kevin Bacon number of 1 because she appeared in *The River Wild* with Kevin Bacon. Nicole Kidman's number is 2: although she did not appear in any movie with Kevin Bacon, she was in *Days of Thunder* with Tom Cruise, and Cruise appeared in *A Few Good Men* with Kevin Bacon. Given the name of a performer, the simplest version of the game is to find some alternating sequence of movies and performers that leads back to Kevin Bacon. For example, a movie buff might know that Tom Hanks was in *Joe Versus the Volcano* with Lloyd Bridges, who was in *High Noon* with Grace Kelly, who was in *Dial M for Murder* with Patrick Allen, who was in *The Eagle Has Landed* with Donald Sutherland, who was in *Animal House* with Kevin Bacon. But this knowledge does not suffice to establish Tom Hanks's Bacon number (it is actually 1 because he was in *Apollo 13* with Kevin Bacon). You can see that the Kevin Bacon number has to be defined by counting the movies in the *shortest* such sequence, so it is hard to be sure whether someone wins the game without using a computer. Of course, as illustrated in the `SymbolGraph` client `DegreesOfSeparation` on page 555, `BreadthFirstPaths` is the program we need to find a shortest path that establishes the Kevin Bacon number of any performer in `movies.txt`. This program takes a source vertex from the command line, then takes queries from standard input and prints a shortest path from the source to the query vertex. Since the graph associated with `movies.txt` is bipartite, all paths alternate between movies and performers, and the printed path is a “proof” that the path is valid (but not a proof that it is the shortest such path—you need to educate your

```
% java DegreesOfSeparation movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
Bacon, Kevin
Woodsman, The (2004)
Grier, David Alan
Bewitched (2005)
Kidman, Nicole
Grant, Cary
Bacon, Kevin
Planes, Trains & Automobiles (1987)
Martin, Steve (I)
Dead Men Don't Wear Plaid (1982)
Grant, Cary
```

friends about PROPOSITION B for that). `DegreesOfSeparation` also finds shortest paths in graphs that are not bipartite: for example, it finds a way to get from one airport to another in `routes.txt` using the fewest connections.

YOU MIGHT ENJOY USING `DegreesOfSeparation` to answer some entertaining questions about the movie business. For example, you can find separations between movies, not just performers. More important, the concept of separation has been widely studied in many other contexts. For example, mathematicians play this same game with the graph defined by paper co-authorship and their connection to P. Erdős, a prolific 20th-century mathematician. Similarly, everyone in New Jersey seems to have a Bruce Springsteen number of 2, because everyone in the state seems to know someone who claims to know Bruce. To play the Erdős game, you would need a database of all mathematical papers; playing the Springsteen game is a bit more challenging. On a more serious note, degrees of separation play a crucial role in the design of computer and communications networks, and in our understanding of natural networks in all fields of science.

```
% java DegreesOfSeparation movies.txt "/" "Animal House (1978)"
Titanic (1997)
    Animal House (1978)
    Allen, Karen (I)
    Raiders of the Lost Ark (1981)
    Taylor, Rocky (I)
    Titanic (1997)
To Catch a Thief (1955)
    Animal House (1978)
    Vernon, John (I)
    Topaz (1969)
    Hitchcock, Alfred (I)
    To Catch a Thief (1955)
```

## Degrees of separation

```
public class DegreesOfSeparation
{
    public static void main(String[] args)
    {
        SymbolGraph sg = new SymbolGraph(args[0], args[1]);
        Graph G = sg.G();
        String source = args[2];
        if (!sg.contains(source))
        { StdOut.println(source + " not in database."); return; }
        int s = sg.index(source);
        BreadthFirstPaths bfs = new BreadthFirstPaths(G, s);
        while (!StdIn.isEmpty())
        {
            String sink = StdIn.readLine();
            if (sg.contains(sink))
            {
                int t = sg.index(sink);
                if (bfs.hasPathTo(t))
                    for (int v : bfs.pathTo(t))
                        StdOut.println(" " + sg.name(v));
                else StdOut.println("Not connected");
            }
            else StdOut.println("Not in database.");
        }
    }
}
```

This `SymbolGraph` and `BreadthFirstPaths` client finds shortest paths in graphs. For `movies.txt`, it plays the Kevin Bacon game.

```
% java DegreesOfSeparation routes.txt " " JFK
LAS
JFK
ORD
PHX
LAS
DFW
JFK
ORD
DFW
```

**Summary** In this section, we have introduced several basic concepts that we will expand upon and further develop throughout the rest of this chapter:

- Graph nomenclature
- A graph representation that enables processing of huge sparse graphs
- A design pattern for graph processing, where we implement algorithms by developing clients that preprocess the graph in the constructor, building data structures that can efficiently support client queries about the graph
- Depth-first search and breadth-first search
- A class providing the capability to use symbolic vertex names

The table below summarizes the implementations of graph algorithms that we have considered. These algorithms are a proper introduction to graph processing, since variants on their code will resurface as we consider more complicated types of graphs and applications, and (consequently) more difficult graph-processing problems. The same questions involving connections and paths among vertices become much more difficult when we add direction and then weights to graph edges, but the same approaches are effective in addressing them and serve as a starting point for addressing more difficult problems.

problem	solution	reference
<i>single-source connectivity</i>	DepthFirstSearch	page 531
<i>single-source paths</i>	DepthFirstPaths	page 536
<i>single-source shortest paths</i>	BreadthFirstPaths	page 540
<i>connectivity</i>	CC	page 544
<i>cycle detection</i>	Cycle	page 547
<i>two-colorability (bipartiteness)</i>	TwoColor	page 547
<b>(Undirected) graph-processing problems addressed in this section</b>		

**Q&A**

**Q.** Why not jam all of the algorithms into `Graph.java`?

**A.** Yes, we might just add query methods (and whatever private fields and methods each might need) to the basic `Graph` ADT definition. While this approach has some of the virtues of data abstraction that we have embraced, it also has some serious drawbacks, because the world of graph processing is significantly more expansive than the kinds of basic data structures treated in SECTION 1.3. Chief among these drawbacks are the following:

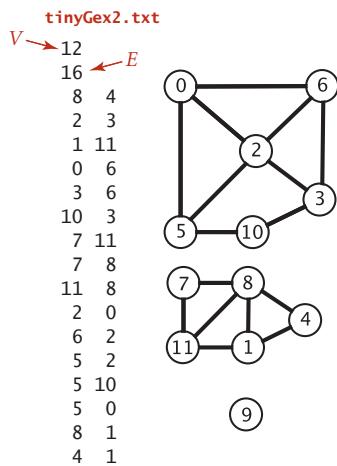
- There are many more graph-processing operations to implement than we can accurately define in a single API.
- Simple graph-processing tasks have to use the same API needed by complicated tasks.
- One method can access a field intended for use by another method, contrary to encapsulation principles that we would like to follow.

This situation is not unusual: APIs of this kind have come to be known as *wide* interfaces (see page 97). In a chapter filled with graph-processing algorithms, an API of this sort would be wide indeed.

**Q.** Does `SymbolGraph` really need two passes?

**A.** No. You could pay an extra  $\log V$  factor and support `adj(O)` directly as an ST instead of a Bag. We have an implementation along these lines in our book *An Introduction to Programming in Java: An Interdisciplinary Approach*.

## EXERCISES



**4.1.1** What is the maximum number of edges in a graph with  $V$  vertices and no parallel edges? What is the minimum number of edges in a graph with  $V$  vertices, none of which are isolated (have degree 0)?

**4.1.2** Draw, in the style of the figure in the text (page 524), the adjacency lists built by Graph's input stream constructor for the file `tinyGex2.txt` depicted at left.

**4.1.3** Create a copy constructor for Graph that takes as input a graph  $G$  and creates and initializes a new copy of the graph. Any changes a client makes to  $G$  should not affect the newly created graph.

**4.1.4** Add a method `hasEdge()` to Graph which takes two `int` arguments  $v$  and  $w$  and returns true if the graph has an edge  $v-w$ , false otherwise.

**4.1.5** Modify Graph to disallow parallel edges and self-loops.

**4.1.6** Consider the four-vertex graph with edges  $0-1$ ,  $1-2$ ,  $2-3$ , and  $3-0$ . Draw an array of adjacency-lists that could *not* have been built calling `addEdge()` for these edges *no matter what order*.

**4.1.7** Develop a test client for Graph that reads a graph from the input stream named as command-line argument and then prints it, relying on `toString()`.

**4.1.8** Develop an implementation for the Search API on page 528 that uses UF, as described in the text.

**4.1.9** Show, in the style of the figure onpage 533, a detailed trace of the call `dfs(0)` for the graph built by Graph's input stream constructor for the file `tinyGex2.txt` (see EXERCISE 4.1.2). Also, draw the tree represented by `edgeTo[]`.

**4.1.10** Prove that every connected graph has a vertex whose removal (including all incident edges) will not disconnect the graph, and write a DFS method that finds such a vertex. Hint: Consider a vertex whose adjacent vertices are all marked.

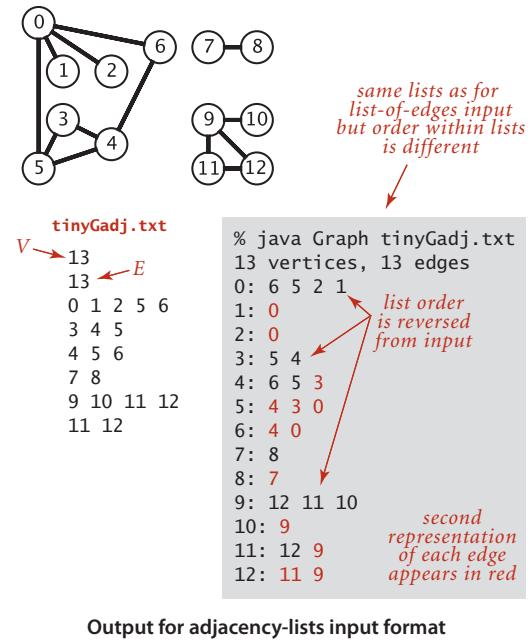
**4.1.11** Draw the tree represented by `edgeTo[]` after the call `bfs(G, 0)` in ALGORITHM 4.2 for the graph built by Graph's input stream constructor for the file `tinyGex2.txt` (see EXERCISE 4.1.2).

**4.1.12** What does the BFS tree tell us about the distance from  $v$  to  $w$  when neither is at the root?

**4.1.13** Add a `distTo()` method to the `BreadthFirstPaths` API and implementation, which returns the number of edges on the shortest path from the source to a given vertex. A `distTo()` query should run in constant time.

**4.1.14** Suppose you use a stack instead of a queue when running breadth-first search. Does it still compute shortest paths?

**4.1.15** Modify the input stream constructor for `Graph` to also allow adjacency lists from standard input (in a manner similar to `SymbolGraph`), as in the example `tinyGadj.txt` shown at right. After the number of vertices and edges, each line contains a vertex and its list of adjacent vertices.



**4.1.16** The *eccentricity* of a vertex  $v$  is the length of the shortest path from that vertex to the furthest vertex from  $v$ . The *diameter* of a graph is the maximum eccentricity of any vertex. The *radius* of a graph is the smallest eccentricity of any vertex. A *center* is a vertex whose eccentricity is the radius. Implement the following API:

---

```
public class GraphProperties
```

---

<code>GraphProperties(Graph G)</code>	<i>constructor (exception if G not connected)</i>
<code>int diameter()</code>	<i>diameter of G</i>
<code>int radius()</code>	<i>radius of G</i>
<code>int center()</code>	<i>a center of G</i>

**4.1.17** The *Wiener index* of a graph is the sum of the lengths of the shortest paths between all pairs of vertices. Mathematical chemists use this quantity to analyze *molecular graphs*, where vertices correspond to atoms and edges correspond to chemical bonds. Add a method `wiener()` to `GraphProperties` that returns the Wiener index of a graph.

**EXERCISES (continued)**

**4.1.18** The *girth* of a graph is the length of its shortest cycle. If a graph is acyclic, then its girth is infinite. Add a method `girth()` to `GraphProperties` that returns the girth of the graph. *Hint:* Run BFS from each vertex. The shortest cycle containing  $s$  is an edge between  $s$  and some vertex  $v$  concatenated with a shortest path between  $s$  and  $v$  (that doesn't use the edge  $s-v$ ).

**4.1.19** Show, in the style of the figure on page 545, a detailed trace of `CC` for finding the connected components in the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see EXERCISE 4.1.2).

**4.1.20** Show, in the style of the figures in this section, a detailed trace of `Cycle` for finding a cycle in the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see EXERCISE 4.1.2). What is the order of growth of the running time of the `Cycle` constructor, in the worst case?

**4.1.21** Show, in the style of the figures in this section, a detailed trace of `TwoColor` for finding a two-coloring of the graph built by `Graph`'s input stream constructor for the file `tinyGex2.txt` (see EXERCISE 4.1.2). What is the order of growth of the running time of the `TwoColor` constructor, in the worst case?

**4.1.22** Run `SymbolGraph` with `movies.txt` to find the Kevin Bacon number of this year's Oscar nominees.

**4.1.23** Write a program `BaconHistogram` that prints a histogram of Kevin Bacon numbers, indicating how many performers from `movies.txt` have a Bacon number of 0, 1, 2, 3, ... . Include a category for those who have an infinite number (not connected to Kevin Bacon).

**4.1.24** Compute the number of connected components in `movies.txt`, the size of the largest component, and the number of components of size less than 10. Find the eccentricity, diameter, radius, a center, and the girth of the largest component in the graph. Does it contain Kevin Bacon?

**4.1.25** Modify `DegreesOfSeparation` to take an `int` value  $y$  as a command-line argument and ignore movies that are more than  $y$  years old.

**4.1.26** Write a `SymbolGraph` client like `DegreesOfSeparation` that uses *depth-first* search instead of breadth-first search to find paths connecting two performers, producing output like that shown on the facing page.

**4.1.27** Determine the amount of memory used by `Graph` to represent a graph with  $V$  vertices and  $E$  edges, using the memory-cost model of SECTION 1.4.

**4.1.28** Two graphs are *isomorphic* if there is a way to rename the vertices of one to make it identical to the other. Draw all the nonisomorphic graphs with two, three, four, and five vertices.

**4.1.29** Modify `Cycle` so that it works even if the graph contains self-loops and parallel edges.

```
% java DegreesOfSeparationDFS movies.txt "/" "Bacon, Kevin"
Kidman, Nicole
    Bacon, Kevin
    Woodsman, The (2004)
    Sedgwick, Kyra
    Something to Talk About (1995)
    Gillan, Lisa Roberts
    Runaway Bride (1999)
    Schertler, Jean
    ...
    ... [1782 movies] (!)
    Eskelson, Dana
    Interpreter, The (2005)
    Silver, Tracey (II)
    Copycat (1995)
    Chua, Jeni
    Metro (1997)
    Ejogo, Carmen
    Avengers, The (1998)
    Atkins, Eileen
    Hours, The (2002)
    Kidman, Nicole
```

## CREATIVE PROBLEMS

**4.1.30 Eulerian and Hamiltonian cycles.** Consider the graphs defined by the following four sets of edges:

```

0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7

```

Which of these graphs have Euler cycles (cycles that visit each edge exactly once)? Which of them have Hamilton cycles (cycles that visit each vertex exactly once)? Develop a linear-time DFS-based algorithm to determine whether a graph has an Euler cycle (and if so find one).

**4.1.31 Graph enumeration.** How many different undirected graphs are there with  $V$  vertices and  $E$  edges (and no parallel edges)?

**4.1.32 Parallel edge detection.** Devise a linear-time algorithm to count the parallel edges in a graph.

**4.1.33 Odd cycles.** Prove that a graph is two-colorable (bipartite) if and only if it contains no odd-length cycle.

**4.1.34 Symbol graph.** Implement a one-pass `SymbolGraph` (it need not be a `Graph` client). Your implementation may pay an extra  $\log V$  factor for graph operations, for symbol-table lookups.

**4.1.35 Biconnectedness.** A graph is *biconnected* if every pair of vertices is connected by two disjoint paths. An *articulation point* in a connected graph is a vertex that would disconnect the graph if it (and its incident edges) were removed. Prove that any graph with no articulation points is biconnected. *Hint:* Given a pair of vertices  $s$  and  $t$  and a path connecting them, use the fact that none of the vertices on the path are articulation points to construct two disjoint paths connecting  $s$  and  $t$ .

**4.1.36 Two-edge connectivity.** A *bridge* in a graph is an edge that, if removed, would increase the number of connected components. A graph that has no bridges is said to be *two-edge connected*. Develop a linear-time DFS-based algorithm for determining whether a given graph is edge connected.

**4.1.37** *Euclidean graphs.* Design and implement an API `EuclideanGraph` for graphs whose vertices are points in the plane that include coordinates. Include a method `show()` that uses `StdDraw` to draw the graph.

**4.1.38** *Image processing.* Implement the *flood fill* operation on the implicit graph defined by connecting adjacent points that have the same color in an image. *Hint:* Avoid an explicit stack.

## EXPERIMENTS

**4.1.39 Random graphs.** Write a program `ErdosRenyiGraph` that takes integer values  $V$  and  $E$  from the command line and builds a graph by generating  $E$  random pairs of integers between 0 and  $V-1$ . *Note:* This generator produces self-loops and parallel edges.

**4.1.40 Random simple graphs.** Write a program `RandomSimpleGraph` that takes integer values  $V$  and  $E$  from the command line and produces, with equal likelihood, each of the possible *simple* graphs with  $V$  vertices and  $E$  edges.

**4.1.41 Random sparse graphs.** Write a program `RandomSparseGraph` to generate random sparse graphs for a well-chosen set of values of  $V$  and  $E$  such that you can use it to run meaningful empirical tests on graphs drawn from the Erdős-Renyi model.

**4.1.42 Random Euclidean graphs.** Write a `EuclideanGraph` client (see EXERCISE 4.1.37) `RandomEuclideanGraph` that produces random graphs by generating  $V$  random points in the unit square ( $x$  and  $y$ -coordinates between 0 and 1), then connecting each point with all points that are within a circle of radius  $d$  centered at that point. *Note:* The graph will almost certainly be connected if  $d$  is larger than the threshold value  $\sqrt{\ln V}/(\pi V)$  and almost certainly disconnected if  $d$  is smaller than that value.

**4.1.43 Random grid graphs.** Write a `EuclideanGraph` client `RandomGridGraph` that generates random graphs by connecting vertices arranged in a  $\sqrt{V}$ -by- $\sqrt{V}$  grid to their neighbors (see EXERCISE 1.5.18). Augment your program to add  $R$  extra random edges. For large  $R$ , shrink the grid so that the total number of edges remains about  $2V$ . Add an option such that an extra edge goes from a vertex  $s$  to a vertex  $t$  with probability inversely proportional to the Euclidean distance between  $s$  and  $t$ .

**4.1.44 Real-world graphs.** Find a large weighted graph on the web—perhaps a map with distances, telephone connections with costs, or an airline rate schedule. Write a program `RandomRealGraph` that builds a graph by choosing  $V$  vertices at random and  $E$  edges at random from the subgraph induced by those vertices.

**4.1.45 Random interval graphs.** Consider a collection of  $V$  intervals on the real line (pairs of real numbers). Such a collection defines an *interval graph* with one vertex corresponding to each interval, with edges between vertices if the corresponding intervals intersect (have any points in common). Write a program that generates  $V$  random intervals in the unit interval, all of length  $d$ , then builds the corresponding interval graph. *Hint:* Use a BST.

**4.1.46** *Random transportation graphs.* One way to define a transportation system is with a set of sequences of vertices, each sequence defining a path connecting the vertices. For example, the sequence 0-9-3-2 defines the edges 0-9, 9-3, and 3-2. Write a `EuclideanGraph` client `RandomTransportation` that builds a graph from an input file consisting of one sequence per line, using symbolic names. Develop input suitable to allow you to use your program to build a graph corresponding to the Paris Métro system.

*Testing all algorithms and studying all parameters against all graph models is unrealistic. For each problem listed below, write a client that addresses the problem for any given input graph, then choose among the generators above to run experiments for that graph model. Use your judgment in selecting experiments, perhaps in response to results of previous experiments. Write a narrative explaining your results and any conclusions that might be drawn.*

**4.1.47** *Path lengths in DFS.* Run experiments to determine empirically the probability that `DepthFirstPaths` finds a path between two randomly chosen vertices and to calculate the average length of the paths found, for various graph models.

**4.1.48** *Path lengths in BFS.* Run experiments to determine empirically the probability that `BreadthFirstPaths` finds a path between two randomly chosen vertices and to calculate the average length of the paths found, for various graph models.

**4.1.49** *Connected components.* Run experiments to determine empirically the distribution of the number of components in random graphs of various types, by generating large numbers of graphs and drawing a histogram.

**4.1.50** *Two-colorable.* Most graphs are not two-colorable, and DFS tends to discover that fact quickly. Run empirical tests to study the number of edges examined by `TwoColor`, for various graph models.

## 4.2 DIRECTED GRAPHS

In *directed graphs*, edges are one-way: the pair of vertices that defines each edge is an ordered pair that specifies a one-way adjacency. Many applications (for example, graphs that represent the web, scheduling constraints, or telephone calls) are naturally expressed in terms of directed graphs. The one-way restriction is natural, easy to enforce in our implementations, and seems innocuous; but it implies added combinatorial structure that has profound implications for our algorithms and makes working with directed graphs quite different from working with undirected graphs. In this section, we consider classic algorithms for exploring and processing directed graphs.

application	vertex	edge
<i>food web</i>	species	predator-prey
<i>internet content</i>	page	hyperlink
<i>program</i>	module	external reference
<i>cellphone</i>	phone	call
<i>scholarship</i>	paper	citation
<i>financial</i>	stock	transaction
<i>internet</i>	machine	connection

### Typical digraph applications

are worth restating. The slight differences in the wording to account for edge directions imply structural properties that will be the focus of this section.

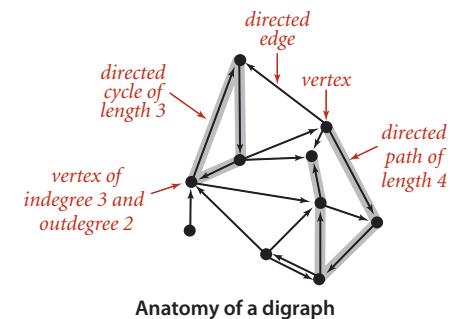
**Definition.** A *directed graph* (or *digraph*) is a set of *vertices* and a collection of *directed edges*. Each directed edge connects an ordered pair of vertices.

We say that a directed edge *points from* the first vertex in the pair and *points to* the second vertex in the pair. The *outdegree* of a vertex in a digraph is the number of edges pointing *from* it; the *indegree* of a vertex is the number of edges pointing *to* it. We drop the modifier *directed* when referring to edges in digraphs when the distinction is obvious in context. The first vertex in a directed edge is called its *tail*; the second vertex is called its *head*. We draw directed edges as arrows pointing from tail to head. We use the notation  $v \rightarrow w$  to refer to an edge that points from  $v$  to  $w$  in a digraph. As with undirected graphs, our code handles parallel edges and self-loops, but they are not present in examples and we generally ignore them in the text. Ignoring anomalies, there are

four different ways in which two vertices might be related in a digraph: no edge; an edge  $v \rightarrow w$  from  $v$  to  $w$ ; an edge  $w \rightarrow v$  from  $w$  to  $v$ ; or two edges  $v \rightarrow w$  and  $w \rightarrow v$ , which indicate connections in both directions.

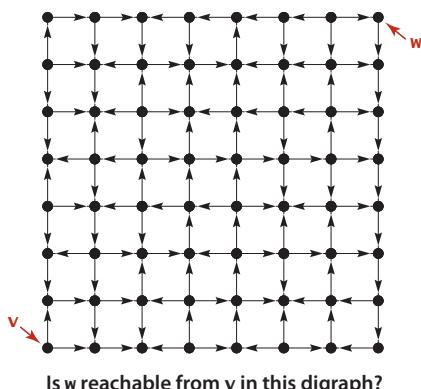
**Definition.** A *directed path* in a digraph is a sequence of vertices in which there is a (directed) edge pointing from each vertex in the sequence to its successor in the sequence. A *directed cycle* is a directed path with at least one edge whose first and last vertices are the same. A *simple path* is a path with no repeated vertices. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices). The *length* of a path or a cycle is its number of edges.

As for undirected graphs, we assume that directed paths are simple unless we specifically relax this assumption by referring to specific repeated vertices (as in our definition of directed cycle) or to *general* directed paths. We say that a vertex  $w$  is *reachable* from a vertex  $v$  if there is a directed path from  $v$  to  $w$ . Also, we adopt the convention that each vertex is reachable from itself. Except for this case, the fact that  $w$  is reachable from  $v$  in a digraph indicates nothing about whether  $v$  is reachable from  $w$ . This distinction is obvious, but critical, as we shall see.



UNDERSTANDING THE ALGORITHMS in this section requires an appreciation of the distinction between reachability in digraphs and connectivity in undirected graphs. Developing such an appreciation is more complicated than you might think. For example,

although you are likely to be able to tell at a glance whether two vertices in a small undirected graph are connected, a directed path in a digraph is not so easy to spot, as indicated in the example at left. Processing digraphs is akin to traveling around in a city where all the streets are one-way, with the directions not necessarily assigned in any uniform pattern. Getting from one point to another in such a situation could be a challenge indeed. Counter to this intuition is the fact that the standard data structure that we use for representing digraphs is *simpler* than the corresponding representation for undirected graphs!



**Digraph data type** The API below and the class Digraph shown on the facing page are virtually identical to those for Graph (page 526).

public class Digraph	
Digraph(int V)	<i>create a V-vertex digraph with no edges</i>
Digraph(In in)	<i>read a digraph from input stream in</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
void addEdge(int v, int w)	<i>add edge v-&gt;w to this digraph</i>
Iterable<Integer> adj(int v)	<i>vertices connected to v by edges pointing from v</i>
Digraph reverse()	<i>reverse of this digraph</i>
String toString()	<i>string representation</i>

API for a digraph

**Representation.** We use the adjacency-lists representation, where an edge  $v \rightarrow w$  is represented as a list node containing  $w$  in the linked list corresponding to  $v$ . This representation is essentially the same as for undirected graphs but is even more straightforward because each edge occurs just once, as shown on the facing page.

**Input format.** The code for the constructor that takes a digraph from an input stream is identical to the corresponding constructor in Graph—the input format is the same, but all edges are interpreted to be directed edges. In the list-of-edges format, a pair  $v \ w$  is interpreted as an edge  $v \rightarrow w$ .

**Reversing a digraph.** Digraph also adds to the API a method `reverse()` which returns a copy of the digraph, with all edges reversed. This method is sometimes needed in digraph processing because it allows clients to find the edges that point *to* each vertex, while `adj()` gives just vertices connected by edges that point *from* each vertex.

**Symbolic names.** It is also a simple matter to allow clients to use symbolic names in digraph applications. To implement a class `SymbolDigraph` like `SymbolGraph` on page 552, replace `Graph` by `Digraph` everywhere.

IT IS WORTHWHILE to take the time to consider carefully the difference, by comparing code and the figure at right with their counterparts for undirected graphs on page 524 and page 526. In the adjacency-lists representation of an undirected graph, we know that if  $v$  is on  $w$ 's list, then  $w$  will be on  $v$ 's list; the adjacency-lists representation of a digraph has no such symmetry. This difference has profound implications in processing digraphs.

## Directed graph (digraph) data type

```

public class Digraph
{
    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    public Digraph(int V)
    {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public int V() { return V; }
    public int E() { return E; }

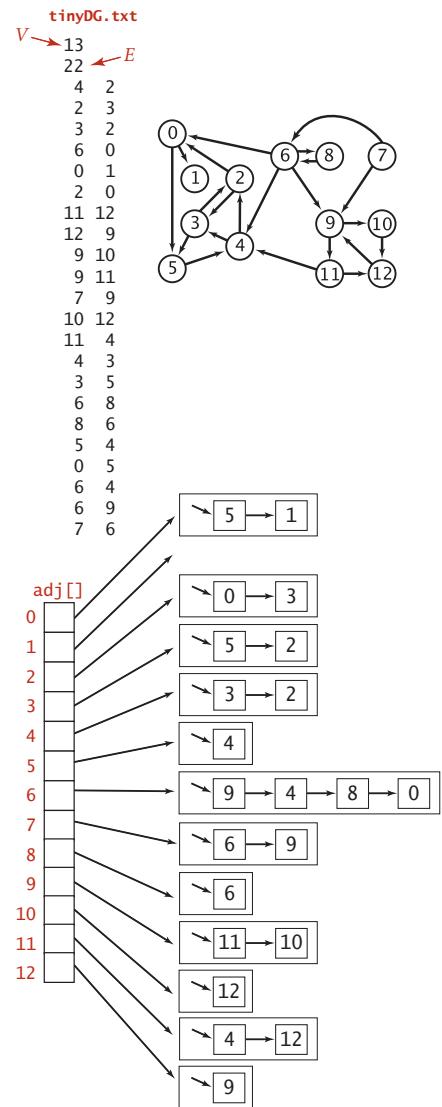
    public void addEdge(int v, int w)
    {
        adj[v].add(w);
        E++;
    }

    public Iterable<Integer> adj(int v)
    { return adj[v]; }

    public Digraph reverse()
    {
        Digraph R = new Digraph(V);
        for (int v = 0; v < V; v++)
            for (int w : adj(v))
                R.addEdge(w, v);
        return R;
    }
}

```

This Digraph data type is identical to Graph (page 526) except that addEdge() only calls add() once, and it has an instance method reverse() that returns a copy with all its edges reversed. Since the code is easily derived from the corresponding code for Graph, we omit the toString() method (see the table on page 523) and the input stream constructor (see page 526).



Digraph input format and adjacency-lists representation

**Reachability in digraphs** Our first graph-processing algorithm for undirected graphs was DepthFirstSearch on page 531, which solves the single-source connectivity problem, allowing clients to determine which vertices are connected to a given source. The *identical code* with Graph changed to Digraph solves the analogous problem for digraphs:

**Single-source reachability.** Given a digraph and a source vertex  $s$ , support queries of the form *Is there a directed path from  $s$  to a given target vertex  $v$ ?*

DirectedDFS on the facing page is a slight embellishment of DepthFirstSearch that implements the following API:

```
public class DirectedDFS
```

---

DirectedDFS(Digraph G, int s)	<i>find vertices in G that are reachable from s</i>
DirectedDFS(Digraph G, Iterable<Integer> sources)	<i>find vertices in G that are reachable from sources</i>
boolean marked(int v)	<i>is v reachable?</i>

**API for reachability in digraphs**

By adding a second constructor that takes a list of vertices, this API supports for clients the following generalization of the problem:

**Multiple-source reachability.** Given a digraph and a *set* of source vertices, support queries of the form *Is there a directed path from some vertex in the set to a given target vertex  $v$ ?*

This problem arises in the solution of a classic string-processing problem that we consider in SECTION 5.4.

DirectedDFS uses our standard graph-processing paradigm and a standard recursive depth-first search to solve these problems. It calls the recursive `dfs()` for each source, which marks every vertex encountered.

**Proposition D.** DFS marks all the vertices in a digraph reachable from a given set of sources in time proportional to the sum of the outdegrees of the vertices marked.

**Proof:** Same as PROPOSITION A on page 531.

A trace of the operation of this algorithm for our sample digraph appears on page 572. This trace is somewhat simpler than the corresponding trace for undirected graphs,

---

**ALGORITHM 4.4** Reachability in digraphs
 

---

```

public class DirectedDFS
{
    private boolean[] marked;
    public DirectedDFS(Digraph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }
    public DirectedDFS(Digraph G, Iterable<Integer> sources)
    {
        marked = new boolean[G.V()];
        for (int s : sources)
            if (!marked[s]) dfs(G, s);
    }
    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }
    public boolean marked(int v)
    { return marked[v]; }
    public static void main(String[] args)
    {
        Digraph G = new Digraph(new In(args[0]));
        Bag<Integer> sources = new Bag<Integer>();
        for (int i = 1; i < args.length; i++)
            sources.add(Integer.parseInt(args[i]));
        DirectedDFS reachable = new DirectedDFS(G, sources);
        for (int v = 0; v < G.V(); v++)
            if (reachable.marked(v)) StdOut.print(v + " ");
        StdOut.println();
    }
}
  
```

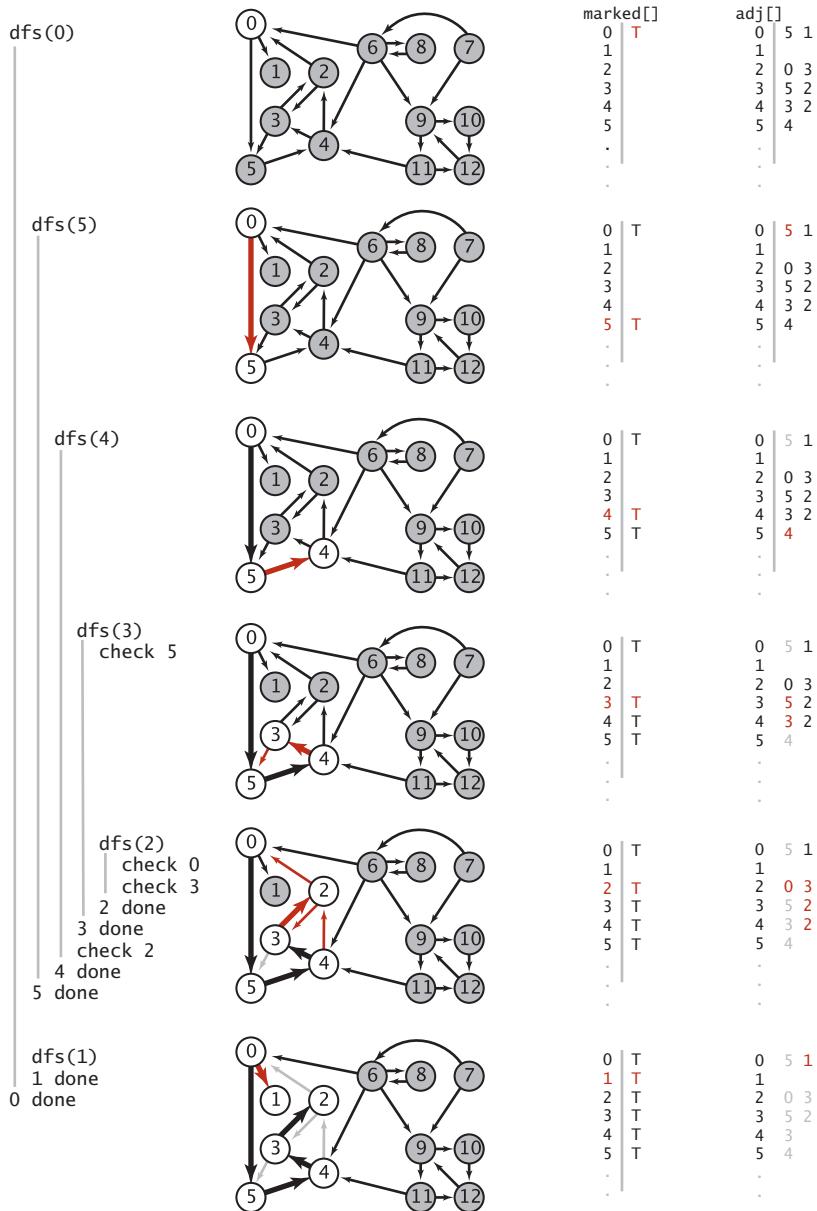
```
% java DirectedDFS tinyDG.txt 1
1
```

```
% java DirectedDFS tinyDG.txt 2
0 1 2 3 4 5
```

```
% java DirectedDFS tinyDG.txt 1 2 6
0 1 2 3 4 5 6 8 9 10 11 12
```

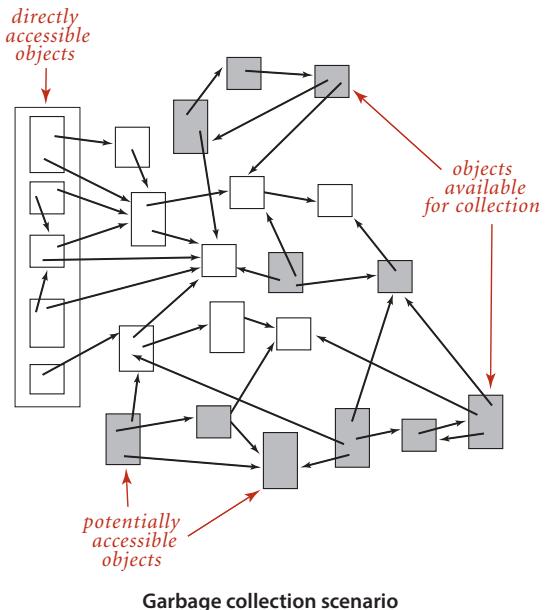
---

This implementation of depth-first search provides clients the ability to test which vertices are reachable from a given vertex or a given set of vertices.



because DFS is fundamentally a digraph-processing algorithm, with one representation of each edge. Following this trace is a worthwhile way to help cement your understanding of depth-first search in digraphs.

**Mark-and-sweep garbage collection.** An important application of multiple-source reachability is found in typical memory-management systems, including many implementations of Java. A digraph where each vertex represents an object and each edge represents a reference to an object is an appropriate model for the memory usage of a running Java program. At any point in the execution of a program, certain objects are known to be directly accessible, and any object not reachable from that set of objects can be returned to available memory. A mark-and-sweep garbage collection strategy reserves one bit per object for the purpose of garbage collection, then periodically *marks* the set of potentially accessible objects by running a digraph reachability algorithm like `DirectedDFS` and *sweeps* through all objects, collecting the unmarked ones for use for new objects.



**Finding paths in digraphs.** `DepthFirstPaths` (ALGORITHM 4.1 on page 536) and `BreadthFirstPaths` (ALGORITHM 4.2 on page 540) are also fundamentally digraph-processing algorithms. Again, the identical APIs and code (with `Graph` changed to `Digraph`) effectively solve the following problems:

**Single-source directed paths.** Given a digraph and a source vertex  $s$ , support queries of the form *Is there a directed path from  $s$  to a given target vertex  $v$ ?* If so, find such a path.

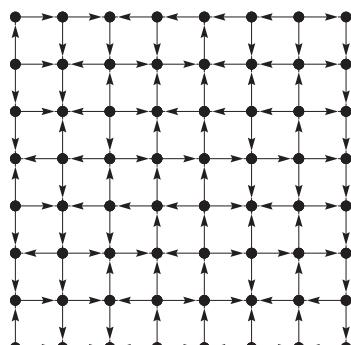
**Single-source shortest directed paths.** Given a digraph and a source vertex  $s$ , support queries of the form *Is there a directed path from  $s$  to a given target vertex  $v$ ?* If so, find a *shortest* such path (one with a minimal number of edges).

On the booksite and in the exercises at the end of this section, we refer to these solutions as `DepthFirstDirectedPaths` and `BreadthFirstDirectedPaths`, respectively.

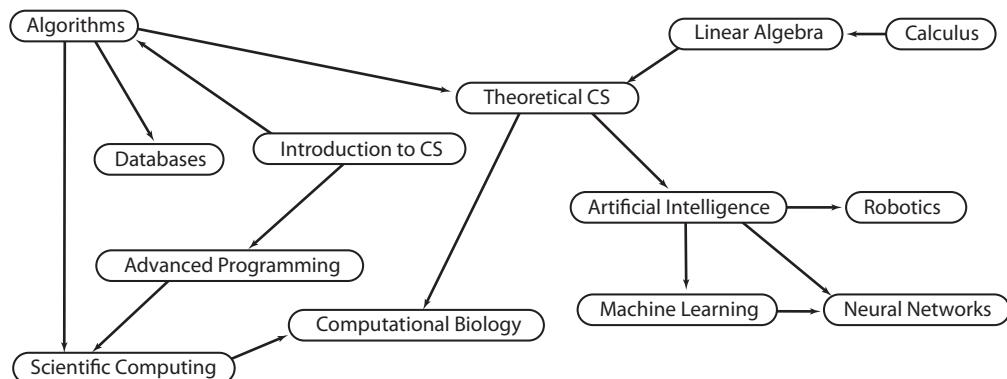
**Cycles and DAGs** Directed cycles are of particular importance in applications that involve processing digraphs. Identifying directed cycles in a typical digraph can be a challenge without the help of a computer, as shown at right. In principle, a digraph might have a huge number of cycles; in practice, we typically focus on a small number of them, or simply are interested in knowing that none are present.

To motivate the study of the role of directed cycles in digraph processing we consider, as a running example, the following prototypical application where digraph models arise directly:

**Scheduling problems.** A widely applicable problem-solving model has to do with arranging for the completion of a set of jobs, under a set of constraints, by specifying when and how the jobs are to be performed. Constraints might involve functions of the time taken or other resources consumed by the jobs. The most important type of constraints is *precedence constraints*, which specify that certain jobs must be performed before certain others. Different types of additional constraints lead to many different types of scheduling problems, of varying difficulty. Literally thousands of different problems have been studied, and researchers still seek better algorithms for many of them. As an example, consider a college student planning a course schedule, under the constraint that certain courses are prerequisite for certain other courses, as in the example below.



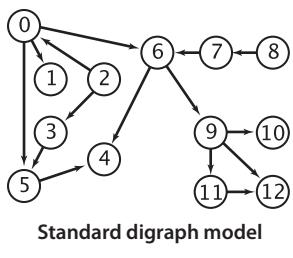
Does this digraph have a directed cycle?



A precedence-constrained scheduling problem

If we further assume that the student can take only one course at a time, we have an instance of the following problem:

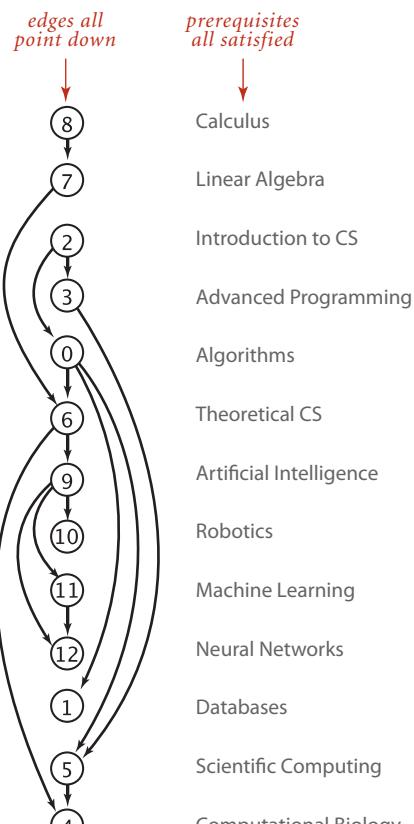
**Precedence-constrained scheduling.** Given a set of jobs to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs such that they are all completed while still respecting the constraints?



For any such problem, a digraph model is immediate, with vertices corresponding to jobs and directed edges corresponding to precedence constraints. For economy, we switch the example to our standard model with vertices labeled as integers, as shown at left. In digraphs, precedence-constrained scheduling amounts to the following fundamental problem:

**Topological sort.** Given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order (or report that doing so is not possible).

A topological order for our example model is shown at right. All edges point down, so it clearly represents a solution to the precedence-constrained scheduling problem that this digraph models: the student can satisfy all course prerequisites by taking the courses in this order. This application is typical—some other representative applications are listed in the table below.



application	vertex	edge
<i>job schedule</i>	job	precedence constraint
<i>course schedule</i>	course	prerequisite
<i>inheritance</i>	Java class	extends
<i>spreadsheet</i>	cell	formula
<i>symbolic links</i>	file name	link

Typical topological-sort applications

Topological sort

**Cycles in digraphs.** If job x must be completed before job y, job y before job z, and job z before job x, then someone has made a mistake, because those three constraints cannot all be satisfied. In general, if a precedence-constrained scheduling problem has a directed cycle, then there is no feasible solution. To check for such errors, we need to be able to solve the following problem:

**Directed cycle detection.** Does a given digraph have a directed cycle? If so, find the vertices on some such cycle, in order from some vertex back to itself.

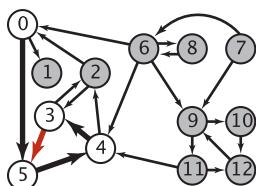
A graph may have an exponential number of cycles (see EXERCISE 4.2.11) so we only ask for one cycle, not all of them. For job scheduling and many other applications it is required that no directed cycle exists, so digraphs where they are absent play a special role:

**Definition.** A *directed acyclic graph* (DAG) is a digraph with no directed cycles.

Solving the directed cycle detection problem thus answers the following question: *Is a given digraph a DAG?* Developing a depth-first-search-based solution to this problem is not difficult, based on the fact that the recursive call stack maintained by the system represents the “current” directed path under consideration (like the string back to the entrance in Tremaux maze exploration). If we ever find a directed edge  $v \rightarrow w$  to a vertex  $w$  that is on that stack, we have found a cycle, since the stack is evidence of a directed path from  $w$  to  $v$ , and the edge  $v \rightarrow w$  completes the cycle. Moreover, the absence of any such *back edges* implies that the graph is acyclic. `DirectedCycle` on the facing page uses this idea to implement the following API:

public class <code>DirectedCycle</code>	
	<i>cycle-finding constructor</i>
<code>DirectedCycle(Digraph G)</code>	<i>does G have a directed cycle?</i>
<code>boolean hasCycle()</code>	<i>vertices on a cycle (if one exists)</i>
<code>Iterable&lt;Integer&gt; cycle()</code>	

**API for directed cycles in digraphs**



	marked[]	edgeTo[]	onStack[]
<code>dfs(0)</code>	0 1 2 3 4 5 ...	0 1 2 3 4 5 ...	0 1 2 3 4 5 ...
<code>dfs(5)</code>	1 0 0 0 0 0 0	- - - - - 0	1 0 0 0 0 0 0
<code>dfs(4)</code>	1 0 0 0 0 0 1	- - - - 5 0	1 0 0 0 0 1 0
<code>dfs(3)</code>	1 0 0 0 0 1 1	- - - 4 5 0	1 0 0 0 1 0 1
<code>check 5</code>	1 0 0 1 1 1	- - - 4 5 0	1 0 0 1 1 0 1

Finding a directed cycle in a digraph

## Finding a directed cycle

```

public class DirectedCycle
{
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle; // vertices on a cycle (if one exists)
    private boolean[] onStack; // vertices on recursive call stack

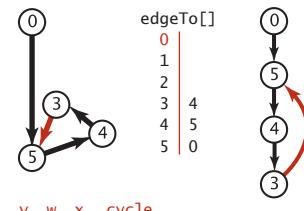
    public DirectedCycle(Digraph G)
    {
        onStack = new boolean[G.V()];
        edgeTo = new int[G.V()];
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        onStack[v] = true;
        marked[v] = true;
        for (int w : G.adj(v))
            if (this.hasCycle()) return;
            else if (!marked[w])
                { edgeTo[w] = v; dfs(G, w); }
            else if (onStack[w])
            {
                cycle = new Stack<Integer>();
                for (int x = v; x != w; x = edgeTo[x])
                    cycle.push(x);
                cycle.push(w);
                cycle.push(v);
            }
        onStack[v] = false;
    }

    public boolean hasCycle()
    { return cycle != null; }

    public Iterable<Integer> cycle()
    { return cycle; }
}

```



v	w	x	cycle
3	5	3	3
3	5	4	4 3
3	5	5	4 3
3	5	5	5 4 3
3	5	5	3 5 4 3

Trace of cycle computation

This class adds to our standard recursive `dfs()` a boolean array `onStack[]` to keep track of the vertices for which the recursive call has not completed. When it finds an edge  $v \rightarrow w$  to a vertex  $w$  that is on the stack, it has discovered a directed cycle, which it can recover by following `edgeTo[]` links.

When executing `dfs(G, v)`, we have followed a directed path from the source to `v`. To keep track of this path, `DirectedCycle` maintains a vertex-indexed array `onStack[]` that marks the vertices on the recursive call stack (by setting `onStack[v]` to `true` on entry to `dfs(G, v)` and to `false` on exit). `DirectedCycle` also maintains an `edgeTo[]` array so that it can return the cycle when it is detected, in the same way as `DepthFirstPaths` (page 536) and `BreadthFirstPaths` (page 540) return paths.

**Depth-first orders and topological sort.** Precedence-constrained scheduling amounts to computing a topological order for the vertices of a DAG, as in this API:

public class <code>Topological</code>	
<code>Topological(Digraph G)</code>	<i>topological-sorting constructor</i>
<code>boolean isDAG()</code>	<i>is G a DAG?</i>
<code>Iterable&lt;Integer&gt; order()</code>	<i>vertices in topological order</i>
API for topological sorting	

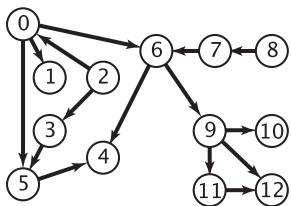
**Proposition E.** A digraph has a topological order if and only if it is a DAG.

**Proof:** If the digraph has a directed cycle, it has no topological order. Conversely, the algorithm that we are about to examine computes a topological order for any given DAG.

Remarkably, it turns out that we have already seen an algorithm for topological sort: a one-line addition to our standard recursive DFS does the job! To convince you of this fact, we begin with the class `DepthFirstOrder` on page 580. It is based on the idea that depth-first search visits each vertex exactly once. If we save the vertex given as argument to the recursive `dfs()` in a data structure, then iterate through that data structure, we see all the graph vertices, in order determined by the nature of the data structure and by whether we do the save before or after the recursive calls. Three vertex orderings are of interest in typical applications:

- *Preorder*: Put the vertex on a queue before the recursive calls.
- *Postorder*: Put the vertex on a queue after the recursive calls.
- *Reverse postorder*: Put the vertex on a stack after the recursive calls.

A trace of `DepthFirstOrder` for our sample DAG is given on the facing page. It is simple to implement and supports `pre()`, `post()`, and `reversePost()` methods that are useful for advanced graph-processing algorithms. For example, `order()` in `Topological` consists of a call on `reversePost()`.



	<i>preorder is order of dfs() calls</i>	<i>postorder is order in which vertices are done</i>	<i>reversePostorder</i>
	pre	post	reversePost
dfs(0)	0		
dfs(5)	0 5		
dfs(4)	0 5 4	queue ↘	
4 done			
5 done			
dfs(1)	0 5 4 1		
1 done		4 5 ↘	
dfs(6)	0 5 4 1 6		
dfs(9)	0 5 4 1 6 9		
dfs(11)	0 5 4 1 6 9 11		
dfs(12)	0 5 4 1 6 9 11 12		
12 done		4 5 1 12 ↘	
11 done		4 5 1 12 11 ↘	
dfs(10)	0 5 4 1 6 9 11 12 10		
10 done		4 5 1 12 11 10 ↘	
check 12			
9 done		4 5 1 12 11 10 9 ↘	
check 4			
6 done		4 5 1 12 11 10 9 6 ↘	
0 done			
check 1		4 5 1 12 11 10 9 6 0 ↘	
dfs(2)	0 5 4 1 6 9 11 12 10 2		
check 0		4 5 1 12 11 10 9 6 0 3 ↘	
dfs(3)	0 5 4 1 6 9 11 12 10 2 3		
check 5		4 5 1 12 11 10 9 6 0 3 2 ↘	
3 done			
2 done		4 5 1 12 11 10 9 6 0 3 2 7 ↘	
check 3			
check 4		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
check 5			
check 6		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
dfs(7)	0 5 4 1 6 9 11 12 10 2 3 7		
check 6		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
7 done			
dfs(8)	0 5 4 1 6 9 11 12 10 2 3 7 8		
check 7		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
8 done			
check 9		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
check 10			
check 11		4 5 1 12 11 10 9 6 0 3 2 7 8 ↘	
check 12			

Computing depth-first orders in a digraph (preorder, postorder, and reverse postorder)

## Depth-first search vertex ordering in a digraph

```
public class DepthFirstOrder
{
    private boolean[] marked;

    private Queue<Integer> pre;           // vertices in preorder
    private Queue<Integer> post;          // vertices in postorder
    private Stack<Integer> reversePost;   // vertices in reverse postorder

    public DepthFirstOrder(Digraph G)
    {
        pre         = new Queue<Integer>();
        post        = new Queue<Integer>();
        reversePost = new Stack<Integer>();
        marked     = new boolean[G.V()];

        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        pre.enqueue(v);

        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);

        post.enqueue(v);
        reversePost.push(v);
    }

    public Iterable<Integer> pre()
    { return pre; }
    public Iterable<Integer> post()
    { return post; }
    public Iterable<Integer> reversePost()
    { return reversePost; }
}
```

This class enables clients to iterate through the vertices in various orders defined by depth-first search. This ability is very useful in the development of advanced digraph-processing algorithms, because the recursive nature of the search enables us to prove properties of the computation (see, for example, PROPOSITION F).

**ALGORITHM 4.5 Topological sort**

```
public class Topological
{
    private Iterable<Integer> order;           // topological order
    public Topological(Digraph G)
    {
        DirectedCycle cyclefinder = new DirectedCycle(G);
        if (!cyclefinder.hasCycle())
        {
            DepthFirstOrder dfs = new DepthFirstOrder(G);
            order = dfs.reversePost();
        }
    }
    public Iterable<Integer> order()
    { return order; }
    public boolean isDAG()
    { return order != null; }
    public static void main(String[] args)
    {
        String filename = args[0];
        String separator = args[1];
        SymbolDigraph sg = new SymbolDigraph(filename, separator);

        Topological top = new Topological(sg.G());
        for (int v : top.order())
            StdOut.println(sg.name(v));
    }
}
```

---

This `DepthFirstOrder` and `DirectedCycle` client returns a topological order for a DAG. The test client solves the precedence-constrained scheduling problem for a `SymbolDigraph`. The instance method `order()` returns `null` if the given digraph is not a DAG and an iterator giving the vertices in topological order otherwise. The code for `SymbolDigraph` is omitted because it is precisely the same as for `SymbolGraph` (page 552), with `Digraph` replacing `Graph` everywhere.

**Proposition F.** Reverse postorder in a DAG is a topological sort.

**Proof:** Consider any edge  $v \rightarrow w$ . One of the following three cases must hold when  $\text{dfs}(v)$  is called (see the diagram on page 583):

- $\text{dfs}(w)$  has already been called and has returned ( $w$  is marked).
- $\text{dfs}(w)$  has not yet been called ( $w$  is unmarked), so  $v \rightarrow w$  will cause  $\text{dfs}(w)$  to be called (and return), either directly or indirectly, before  $\text{dfs}(v)$  returns.
- $\text{dfs}(w)$  has been called and has not yet returned when  $\text{dfs}(v)$  is called. The key to the proof is that this case is impossible in a DAG, because the recursive call chain implies a path from  $w$  to  $v$  and  $v \rightarrow w$  would complete a directed cycle.

In the two possible cases,  $\text{dfs}(w)$  is done before  $\text{dfs}(v)$ , so  $w$  appears *before*  $v$  in postorder and *after*  $v$  in reverse postorder. Thus, each edge  $v \rightarrow w$  points from a vertex earlier in the order to a vertex later in the order, as desired.

```
% more jobs.txt
Algorithms/Theoretical CS/Database/Scientific Computing
Introduction to CS/Advanced Programming/Algorithms
Advanced Programming/Scientific Computing
Scientific Computing/Computational Biology
Theoretical CS/Computational Biology/Artificial Intelligence
Linear Algebra/Theoretical CS
Calculus/Linear Algebra
Artificial Intelligence/Neural Networks/Robotics/Machine Learning
Machine Learning/Neural Networks

% java Topological jobs.txt "/"
Calculus
Linear Algebra
Introduction to CS
Advanced Programming
Algorithms
Theoretical CS
Artificial Intelligence
Robotics
Machine Learning
Neural Networks
Databases
Scientific Computing
Computational Biology
```

**Topological** (ALGORITHM 4.5 on page 581) is an implementation that uses depth-first search to topologically sort a DAG. A trace is given at right.

**Proposition G.** With DFS, we can topologically sort a DAG in time proportional to  $V+E$ .

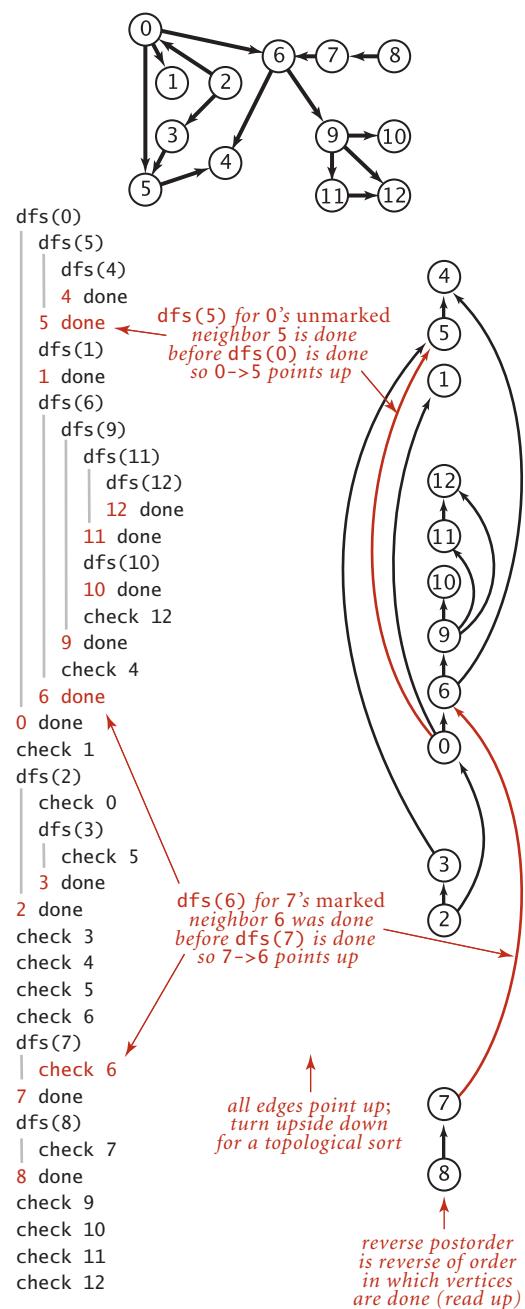
**Proof:** Immediate from the code. It uses one depth-first search to ensure that the graph has no directed cycles, and another to do the reverse postorder ordering. Both involve examining all the edges and all the vertices, and thus take time proportional to  $V+E$ .

Despite the simplicity of this algorithm, it escaped attention for many years, in favor of a more intuitive algorithm based on maintaining a queue of vertices of indegree 0 (see EXERCISE 4.2.39).

IN PRACTICE, topological sorting and cycle detection go hand in hand, with cycle detection playing the role of a debugging tool. For example, in a job-scheduling application, a directed cycle in the underlying digraph represents a mistake that must be corrected, no matter how the schedule was formulated. Thus, a job-scheduling application is typically a three-step process:

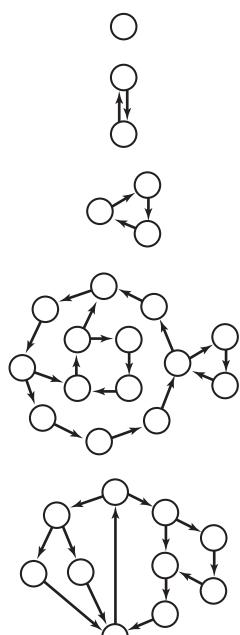
- Specify the tasks and precedence constraints.
- Make sure that a feasible solution exists, by detecting and removing cycles in the underlying digraph until none exist.
- Solve the scheduling problem, using topological sort.

Similarly, any changes in the schedule can be checked for cycles (using `DirectedCycle`), then a new schedule computed (using `Topological`).



Reverse postorder in a DAG is a topological sort

**Strong connectivity in digraphs** We have been careful to maintain a distinction between reachability in digraphs and connectivity in undirected graphs. In an undirected graph, two vertices  $v$  and  $w$  are connected if there is a path connecting them—we can use that path to get from  $v$  to  $w$  or to get from  $w$  to  $v$ . In a digraph, by contrast, a vertex  $w$  is reachable from a vertex  $v$  if there is a directed path from  $v$  to  $w$ , but there may or may not be a directed path back to  $v$  from  $w$ . To complete our study of digraphs, we consider the natural analog of connectivity in undirected graphs.



## Strongly connected digraphs

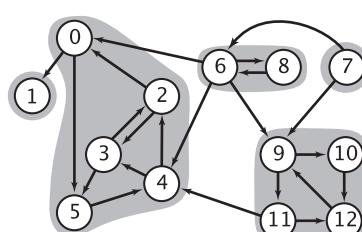
**Definition.** Two vertices  $v$  and  $w$  are *strongly connected* if they are mutually reachable: that is, if there is a directed path from  $v$  to  $w$  and a directed path from  $w$  to  $v$ . A digraph is *strongly connected* if all its vertices are strongly connected to one another.

Several examples of strongly connected graphs are given in the figure at left. As you can see from the examples, cycles play an important role in understanding strong connectivity. Indeed, recalling that a general directed cycle is a directed cycle that may have repeated vertices, it is easy to see that *two vertices are strongly connected if and only if there exists a general directed cycle that contains them both.* (*Proof:* compose the paths from  $v$  to  $w$  and from  $w$  to  $v$ .)

**Strong components.** Like connectivity in undirected graphs, strong connectivity in digraphs is an equivalence relation on the set of vertices, as it has the following properties:

- **Reflexive:** Every vertex  $v$  is strongly connected to itself.
  - **Symmetric:** If  $v$  is strongly connected to  $w$ , then  $w$  is strongly connected to  $v$ .
  - **Transitive:** If  $v$  is strongly connected to  $w$  and  $w$  is strongly connected to  $x$ , then  $v$  is also strongly connected to  $x$ .

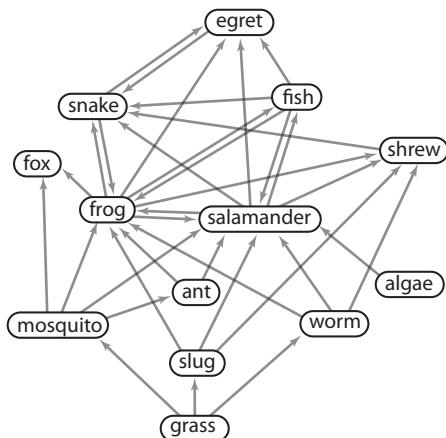
As an equivalence relation, strong connectivity partitions the vertices into equivalence classes. The equivalence classes are maximal subsets of vertices that are strongly connected to one another, with each vertex in exactly one subset. We refer to these subsets as *strongly connected components*, or *strong components* for short. Our sample digraph `tinyDG.txt` has five strong components, as shown in the diagram at right. A digraph with  $V$  vertices has between 1 and  $V$  strong components—a strongly



## A digraph and its strong components

connected digraph has 1 strong component and a DAG has  $V$  strong components. Note that the strong components are defined in terms of the vertices, not the edges. Some edges connect two vertices in the same strong component; some other edges connect vertices in different strong components. The latter are not found on any directed cycle. Just as identifying connected components is typically important in processing undirected graphs, identifying strong components is typically important in processing digraphs.

**Examples of applications.** Strong connectivity is a useful abstraction in understanding the structure of a digraph, highlighting interrelated sets of vertices (strong components). For example, strong components can help textbook authors decide which topics should be grouped together and software developers decide how to organize program modules. The figure below shows an example from ecology. It illustrates a digraph that models the food web connecting living organisms, where vertices represent species and an edge from one vertex to another indicates that an organism of the species indicated by the *point to* vertex consumes organisms of the species indicated by the *point from* vertex for food. Scientific studies on such digraphs (with carefully chosen sets of species and carefully documented relationships) play an important role in helping ecologists answer basic questions about ecological systems. Strong components in such digraphs can help ecologists understand energy flow in the food web. The figure on page 591 shows a digraph model of web content, where vertices represent pages and edges represent hyperlinks from one page to another. Strong components in such a digraph can help network engineers partition the huge number of pages on the web into more manageable sizes for processing. Further properties of these applications and other examples are addressed in the exercises and on the booksite.



Small subset of food web digraph

application	vertex	edge
<i>web</i>	page	hyperlink
<i>textbook</i>	topic	reference
<i>software</i>	module	call
<i>food web</i>	organism	predator-prey relationship

#### Typical strong-component applications

Accordingly, we need the following API, the analog for digraphs of CC (page 543):

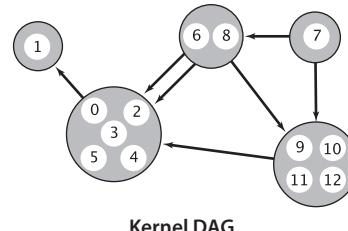
<code>public class SCC</code>	
<code>    SCC(Digraph G)</code>	<i>preprocessing constructor</i>
<code>    boolean stronglyConnected(int v, int w)</code>	<i>are v and w strongly connected?</i>
<code>    int count()</code>	<i>number of strong components</i>
<code>    int id(int v)</code>	<i>component identifier for v (between 0 and count()-1)</i>
<b>API for strong components</b>	

A quadratic algorithm to compute strong components is not difficult to develop (see EXERCISE 4.2.31), but (as usual) quadratic time and space requirements are prohibitive for huge digraphs that arise in practical applications like the ones just described.

**Kosaraju–Sharir algorithm.** We saw in CC (ALGORITHM 4.3 on page 544) that computing connected components in undirected graphs is a simple application of depth-first search. How can we efficiently compute strong components in digraphs? Remarkably, the implementation KosarajuSharirSCC on the facing page does the job with just a few lines of code added to CC, as follows:

- Given a digraph  $G$ , use DepthFirstOrder to compute the reverse postorder of its reverse digraph,  $G^R$ .
- Run standard DFS on  $G$ , but consider the unmarked vertices in the order just computed instead of the standard numerical order.
- All vertices visited on a call to the recursive `dfs()` from the constructor are a *strong component* (!), so identify them as such, in the same manner as in CC.

The Kosaraju–Sharir algorithm is an extreme example of a method that is easy to code but difficult to understand. To persuade yourself that the algorithm is correct, start by considering the *kernel DAG* (or *condensation digraph*) associated with each digraph, formed by collapsing all the vertices in each strong component to a single vertex (and removing any self-loops). The result must be a DAG because any directed cycle would imply a larger strong component. The kernel DAG for the digraph on page 584 has five vertices and seven edges, as shown at right (note the possibility of parallel edges). Since the kernel DAG is a DAG, its vertices can be placed in (reverse) topological order, as shown in the diagram at the top of page 588. This ordering is the key to understanding the Kosaraju–Sharir algorithm.



**ALGORITHM 4.6** Kosaraju–Sharir algorithm for computing strong components

```

public class KosarajuSharirSCC
{
    private boolean[] marked;    // reached vertices
    private int[] id;           // component identifiers
    private int count;          // number of strong components

    public KosarajuSharirSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder order = new DepthFirstOrder(G.reverse());
        for (int s : order.reversePost())
            if (!marked[s])
                { dfs(G, s); count++; }

        % java KosarajuSharirSCC tinyDG.txt
        5 strong components
        1
        0 2 3 4 5
        9 10 11 12
        6 8
        7
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean stronglyConnected(int v, int w)
    { return id[v] == id[w]; }

    public int id(int v)
    { return id[v]; }

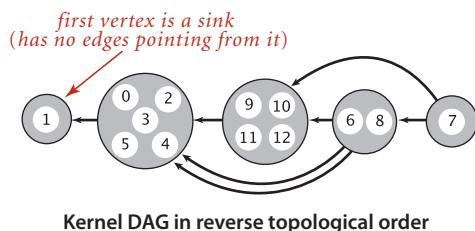
    public int count()
    { return count; }
}

```

This implementation differs from CC (ALGORITHM 4.3) only in the highlighted code (and in the implementation of `main()` where we use the code on page 543, with `Graph` changed to `Digraph`, `CC` changed to `KosarajuSharirSCC`, and “components” changed to “strong components”). To find strong components, it does a depth-first search in the reverse digraph to produce a vertex order (reverse postorder of that search) for use in a depth-first search of the given digraph.

The Kosaraju–Sharir algorithm identifies the strong components in reverse topological order of the kernel DAG. It begins by finding a vertex that is in a sink component of the kernel DAG. When it runs DFS from that vertex, it visits precisely the vertices in that component. The DFS marks those vertices, effectively removing them from the digraph. Next, it finds a vertex that is in a sink component in the remaining kernel DAG, visits precisely the vertices in that component, and so forth.

The postorder of  $G^R$  enables us to examine the strong components in the desired order. The first vertex in a reverse postorder of  $G$  is in a *source* component of the kernel DAG; the first vertex in a reverse postorder of the *reverse* digraph  $G^R$  is in a *sink* component of the kernel DAG (see EXERCISE 4.2.16). More generally, the following lemma relates the reverse postorder of  $G^R$  to the strong components, based on edges in the kernel DAG: it is the key to establishing the correctness of the Kosaraju–Sharir algorithm.



Kernel DAG in reverse topological order

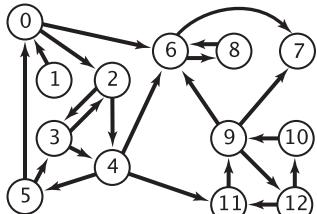
**Postorder lemma.** Let  $C$  be a strong component in a digraph  $G$  and let  $v$  be any vertex not in  $C$ . If there is an edge  $e$  pointing from any vertex in  $C$  to  $v$ , then vertex  $v$  appears before *every* vertex in  $C$  in the reverse postorder of  $G^R$ .

**Proof:** See EXERCISE 4.2.15.

**Proposition H.** The Kosaraju–Sharir algorithm identifies the strong components of a digraph  $G$ .

**Proof:** By induction on the number of strong components identified in the DFS of  $G$ . After the algorithm has identified the first  $i$  components, we assume (by our inductive hypothesis) that the vertices in the first  $i$  components are marked and the vertices in the remaining components are unmarked. Let  $s$  be the unmarked vertex that appears first in the reverse postorder of  $G^R$ . Then, the constructor call `dfs(G, s)` will visit every vertex in the strong component containing  $s$  (which we refer to as component  $i+1$ ) and only those vertices because:

- Vertices in the first  $i$  components will not be visited (because they are already marked).
- Vertices in component  $i+1$  are not yet marked and are reachable from  $s$  using only other vertices in component  $i+1$  (so will be visited and marked).
- Vertices in components after  $i+1$  will not be visited (or marked): Consider (for the sake of contradiction) the first such vertex  $v$  that is visited. Let  $e$  be an edge that goes from a vertex in component  $i+1$  to  $v$ . By the postorder lemma,  $v$  appears in the reverse postorder before every vertex in component  $i+1$  (including  $s$ ). This contradicts the definition of  $s$ .

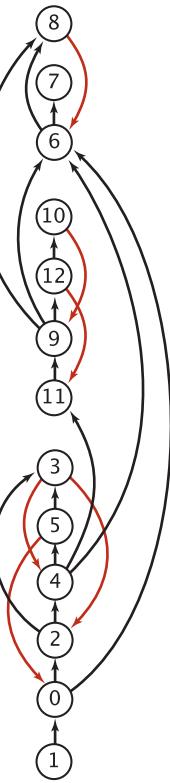
DFS in reverse digraph  $G^r$ 

check unmarked vertices in the order

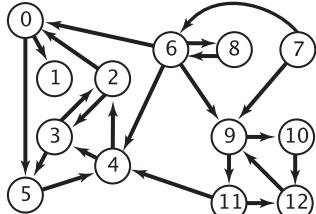
0 1 2 3 4 5 6 7 8 9 10 11 12

```

dfs(0)
  dfs(6)
    dfs(8)
      | check 6
      8 done
      dfs(7)
      7 done
    6 done
    dfs(2)
      dfs(4)
        dfs(11)
        dfs(9)
        dfs(12)
          check 11
          dfs(10)
            | check 9
            10 done
            12 done
            check 7
            check 6
            9 done
            11 done
            check 6
            dfs(5)
              dfs(3)
                | check 4
                | check 2
                3 done
                check 0
                5 done
                4 done
                check 3
                2 done
                0 done
                dfs(1)
                  | check 0
                  1 done
                  check 2
                  check 3
                  check 4
                  check 5
                  check 6
                  check 7
                  check 8
                  check 9
                  check 10
                  check 11
                  check 12
  
```



reverse  
postorder  
for use  
in second  
dfs()  
(read up)

DFS in original digraph  $G$ 

check unmarked vertices in the order

1 0 2 4 5 3 11 9 12 10 6 7 8

```

dfs(1)
  1 done
  dfs(0)
    dfs(5)
      dfs(4)
        dfs(3)
          | check 5
          | check 0
          2 done
          3 done
          check 2
          4 done
          5 done
          check 1
          0 done
          check 2
          check 4
          check 5
          check 3
  
```

dfs(11)
 check 4
 dfs(12)
 dfs(9)
 check 11
 dfs(10)
 | check 12
 10 done
 9 done
 12 done
 11 done
 check 9
 check 12
 check 10

dfs(6)
 | check 9
 | check 4
 | check 6
 8 done
 6 done

dfs(7)
 | check 6
 | check 9
 7 done
 check 8

(1)
 (2)
 (3)
 (4)
 (5)
 (0)
 (11)
 (10)
 (9)
 (12)
 (11)
 (0)
 (1)
 (2)
 (3)
 (4)
 (5)
 (6)
 (7)
 (8)
 (9)
 (10)
 (11)
 (12)

strong components

Kosaraju-Sharir algorithm for finding strong components in a digraph

A trace of the algorithm for `tinyDG.txt` is shown on the preceding page. To the right of each DFS trace is a drawing of the digraph, with vertices appearing in the order they are done. Thus, reading up the reverse digraph drawing on the left gives the reverse postorder in  $G^R$ , the order in which unmarked vertices are checked in the DFS of  $G$ . As you can see from the diagram, the second DFS calls `dfs(1)` (which marks vertex 1) then calls `dfs(0)` (which marks 0, 5, 4, 3, and 2), then checks 2, 4, 5, and 3, then calls `dfs(11)` (which marks 11, 12, 9, and 10), then checks 9, 12, and 10, then calls `dfs(6)` (which marks 6 and 8), and finally `dfs(7)`, which marks 7.

A larger example, a very small subset of a digraph model of the web, is shown on the facing page.

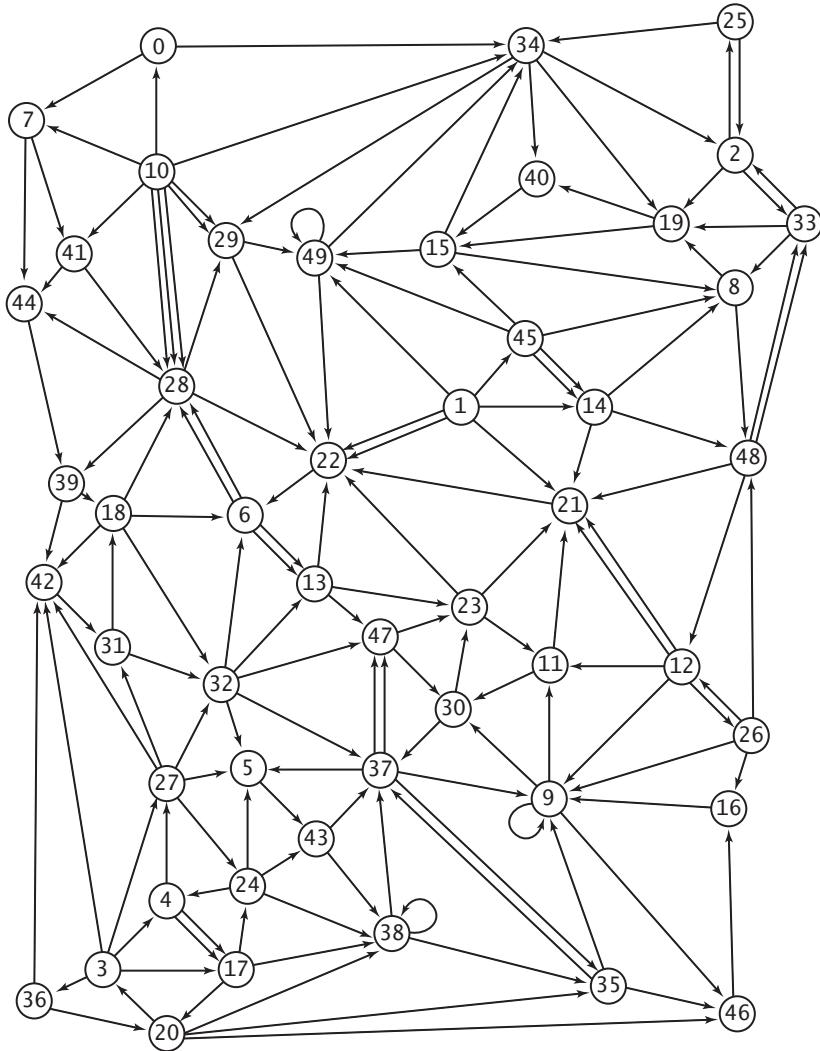
THE KOSARAJU–SHARIR ALGORITHM SOLVES the following analog of the connectivity problem for undirected graphs that we first posed in CHAPTER 1 and reintroduced in SECTION 4.1 (page 534):

**Strong connectivity.** Given a digraph, support queries of the form: *Are two given vertices strongly connected?* and *How many strong components does the digraph have?*

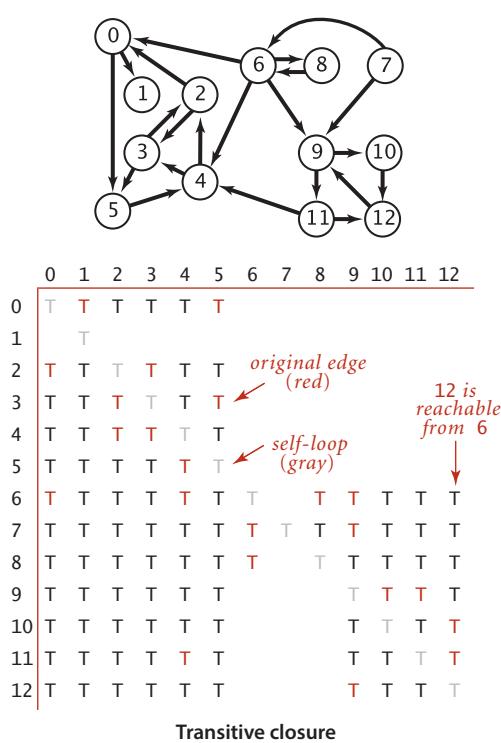
That we can solve this problem in digraphs as efficiently as the corresponding connectivity problem in undirected graphs was an open research problem for some time (resolved by R. E. Tarjan in the early 1970s). That such a simple solution is now available is quite surprising.

**Proposition I.** The Kosaraju–Sharir algorithm uses preprocessing time and space proportional to  $V+E$  to support constant-time strong connectivity queries in a digraph.

**Proof:** The algorithm computes the reverse of the digraph and does two depth-first searches. Each of these three steps takes time proportional to  $V+E$ . The reverse copy of the digraph uses space proportional to  $V+E$ .



How many strong components are there in this digraph?



**Reachability revisited.** With CC for undirected graphs, we can infer from the fact that two vertices  $v$  and  $w$  are connected that there is a path from  $v$  to  $w$  and a path (the same one) from  $w$  to  $v$ . With KosarajuSharirSCC, we can infer from the fact that  $v$  and  $w$  are strongly connected that there is a path from  $v$  to  $w$  and a path (a different one) from  $w$  to  $v$ . But what about pairs of vertices that are not strongly connected? There may be a path from  $v$  to  $w$  or a path from  $w$  to  $v$  or neither, but not both.

**All-pairs reachability.** Given a digraph, support queries of the form *Is there a directed path from a given vertex  $v$  to another given vertex  $w$ ?*

For undirected graphs, the corresponding problem is equivalent to the connectivity problem; for digraphs, it is quite different from the strong connectivity problem. Our CC implementation uses linear preprocessing time to support constant-time answers to such queries for undirected graphs. Can we achieve this performance for digraphs? This seemingly innocuous question has confounded experts for decades. To better understand the challenge,

consider the diagram at left, which illustrates the following fundamental concept:

**Definition.** The *transitive closure* of a digraph  $G$  is another digraph with the same set of vertices, but with an edge from  $v$  to  $w$  in the transitive closure if and only if  $w$  is reachable from  $v$  in  $G$ .

By convention, every vertex is reachable from itself, so the transitive closure has  $V$  self-loops. Our sample digraph has just 22 directed edges, but its transitive closure has 108 out of a possible 169 directed edges. Generally, the transitive closure of a digraph has many more edges than the digraph itself, and it is not at all unusual for a sparse graph to have a dense transitive closure. For example, the transitive closure of a  $V$ -vertex directed cycle, which has  $V$  directed edges, is a complete digraph with  $V^2$  directed edges. Since transitive closures are typically dense, we normally represent them with a matrix of boolean values, where the entry in row  $v$  and column  $w$  is true if and only if  $w$  is

reachable from  $v$ . Instead of explicitly computing the transitive closure, we use depth-first search to implement the following API:

```
public class TransitiveClosure
```

---

```
    TransitiveClosure(Digraph G)
```

*preprocessing constructor*

```
    boolean reachable(int v, int w)
```

*is  $w$  reachable from  $v$ ?*

**API for all-pairs reachability**

The code below is a straightforward implementation that uses `DirectedDFS` (ALGORITHM 4.4). This solution is ideal for small or dense digraphs, but it is not a solution for the large digraphs we might encounter in practice because *the constructor uses space proportional to  $V^2$  and time proportional to  $V(V+E)$* : each of the  $V$  `DirectedDFS` objects takes space proportional to  $V$  (they all have `marked[]` arrays of size  $V$  and examine  $E$  edges to compute the marks). Essentially, `TransitiveClosure` computes and stores the transitive closure of  $G$ , to support constant-time queries—row  $v$  in the transitive closure matrix is the `marked[]` array for the  $v$ th entry in the `DirectedDFS[]` in `TransitiveClosure`. Can we support constant-time queries with substantially less preprocessing time and substantially less space? A general solution that achieves constant-time queries with substantially less than quadratic space is an unsolved research problem, with important practical implications: for example, until it is solved, we cannot hope to have a practical solution to the all-pairs reachability problem for a giant digraph such as the web graph.

```
public class TransitiveClosure
{
    private DirectedDFS[] all;
    TransitiveClosure(Digraph G)
    {
        all = new DirectedDFS[G.V()];
        for (int v = 0; v < G.V(); v++)
            all[v] = new DirectedDFS(G, v);
    }
    boolean reachable(int v, int w)
    {   return all[v].marked(w); }
}
```

All-pairs reachability

**Summary** In this section, we have introduced directed edges and digraphs, emphasizing the relationship between digraph processing and corresponding problems for undirected graphs, as summarized in the following list of topics:

- Digraph nomenclature
- The idea that the representation and approach are essentially the same as for undirected graphs, but some digraph problems are more complicated
- Cycles, DAGs, topological sort, and precedence-constrained scheduling
- Reachability, paths, and strong connectivity in digraphs

The table below summarizes the implementations of digraph algorithms that we have considered (all but one of the algorithms are based on depth-first search). The problems addressed are all simply stated, but the solutions that we have considered range from easy adaptations of corresponding algorithms for undirected graphs to an ingenious and surprising solution. These algorithms are a starting point for several of the more complicated algorithms that we consider in SECTION 4.4, when we consider *edge-weighted* digraphs.

problem	solution	reference
<i>single- and multiple-source reachability</i>	DirectedDFS	page 571
<i>single-source directed paths</i>	DepthFirstDirectedPaths	page 573
<i>single-source shortest directed paths</i>	BreadthFirstDirectedPaths	page 573
<i>directed cycle detection</i>	DirectedCycle	page 577
<i>depth-first vertex orders</i>	DepthFirstOrder	page 580
<i>precedence-constrained scheduling</i>	Topological	page 581
<i>topological sort</i>	Topological	page 581
<i>strong connectivity</i>	KosarajuSharirSCC	page 587
<i>all-pairs reachability</i>	TransitiveClosure	page 593

Digraph-processing problems addressed in this section

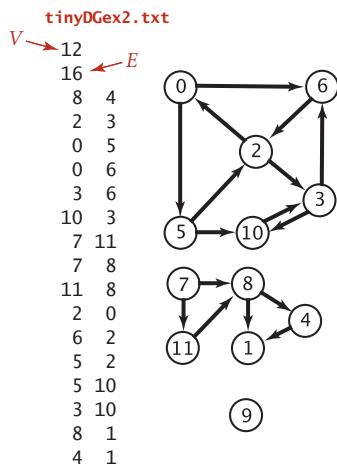
**Q&A**

**Q.** Is a self-loop a cycle?

**A.** Yes, but no self-loop is needed for a vertex to be reachable from itself.

## EXERCISES

**4.2.1** What is the maximum number of edges in a digraph with  $V$  vertices and no parallel edges? What is the minimum number of edges in a digraph with  $V$  vertices, none of which are isolated?



**4.2.2** Draw, in the style of the figure in the text (page 524), the adjacency lists built by `Digraph`'s input stream constructor for the file `tinyDGex2.txt` depicted at left.

**4.2.3** Create a copy constructor for `Digraph` that takes as input a digraph  $G$  and creates and initializes a new copy of the digraph. Any changes a client makes to  $G$  should not affect the newly created digraph.

**4.2.4** Add a method `hasEdge()` to `Digraph` which takes two `int` arguments  $v$  and  $w$  and returns `true` if the graph has an edge  $v \rightarrow w$ , `false` otherwise.

**4.2.5** Modify `Digraph` to disallow parallel edges and self-loops.

**4.2.6** Develop a test client for `Digraph`.

**4.2.7** The *indegree* of a vertex in a digraph is the number of directed edges that point to that vertex. The *outdegree* of a vertex in a digraph is the number of directed edges that emanate from that vertex. No vertex is reachable from a vertex of outdegree 0, which is called a *sink*; a vertex of indegree 0, which is called a *source*, is not reachable from any other vertex. A digraph where self-loops are allowed *and* every vertex has outdegree 1 is called a *map* (a function from the set of integers from 0 to  $V-1$  onto itself). Write a program `Degrees.java` that implements the following API:

---

```
public class Degrees
```

---

<pre>         Degrees(Digraph G)         int indegree(int v)         int outdegree(int v)         Iterable&lt;Integer&gt; sources()         Iterable&lt;Integer&gt; sinks()         boolean isMap()     </pre>	<i>constructor</i> <i>indegree of v</i> <i>outdegree of v</i> <i>sources</i> <i>sinks</i> <i>is G a map?</i>
--	---

**4.2.8** Draw all the nonisomorphic DAGs with two, three, four, and five vertices (see EXERCISE 4.1.28).

**4.2.9** Write a method that checks whether a given permutation of a DAG's vertices is a topological order of that DAG.

**4.2.10** Given a DAG, does there exist a topological order that cannot result from applying a DFS-based algorithm, no matter in what order the vertices adjacent to each vertex are chosen? Prove your answer.

**4.2.11** Describe a family of sparse digraphs whose number of directed cycles grows exponentially in the number of vertices.

**4.2.12** Prove that the strong components in  $G^R$  are the same as in  $G$ .

**4.2.13** Prove that two vertices in a digraph  $G$  are in the same strong component if and only if there is a directed cycle (not necessarily simple) containing both of them.

**4.2.14** Let  $C$  be a strong component in a digraph  $G$  and let  $v$  be any vertex not in  $C$ . Prove that if there is an edge  $e$  pointing from  $v$  to any vertex in  $C$ , then vertex  $v$  appears before *every* vertex in  $C$  in the reverse postorder of  $G$ .

*Solution:* If  $v$  is visited before every vertex in  $C$ , then every vertex in  $C$  will be visited and finished before  $v$  finishes (because every vertex in  $C$  is reachable from  $v$  via edge  $e$ ). If some vertex in  $C$  is visited before  $v$ , then all vertices in  $C$  will be visited and finished before  $v$  is visited (because  $v$  is not reachable from any vertex in  $C$ —if it were, such a path when combined with edge  $e$  would be part of a directed cycle, implying that  $v$  is in  $C$ ).

**4.2.15** Let  $C$  be a strong component in a digraph  $G$  and let  $v$  be any vertex not in  $C$ . Prove that if there is an edge  $e$  pointing from any vertex in  $C$  to  $v$ , then vertex  $v$  appears before *every* vertex in  $C$  in the reverse postorder of  $G^R$ .

*Solution:* Apply EXERCISE 4.2.14 to  $G^R$ .

**4.2.16** Given a digraph  $G$ , prove that the first vertex in the reverse postorder of  $G$  is in a strong component that is a *source* of  $G$ 's kernel DAG. Then, prove that the first vertex in the reverse postorder of  $G^R$  is in a strong component that is a *sink* of  $G$ 's kernel DAG.  
*Hint:* Apply EXERCISES 4.2.14 and 4.2.15.

**EXERCISES (continued)**

**4.2.17** How many strong components are there in the digraph on page 591?

**4.2.18** What are the strong components of a DAG?

**4.2.19** What happens if you run the Kosaraju–Sharir algorithm on a DAG?

**4.2.20** True or false: The reverse postorder of a digraph’s reverse is the same as the postorder of the digraph.

**4.2.21** True or false: If we consider the vertices of a digraph  $G$  (or its reverse  $G^R$ ) in postorder, then vertices in the same strong component will be consecutive in that order.

*Solution:* False. In `tinyDG.txt`, vertices 6 and 8 form a strong component, but they are not consecutive in the postorder of  $G^R$ .

**4.2.22** True or false: If we modify the Kosaraju–Sharir algorithm to run the first depth-first search in the digraph  $G$  (instead of the reverse digraph  $G^R$ ) and the second depth-first search in  $G^R$  (instead of  $G$ ), then it will still find the strong components.

**4.2.23** True or false: If we modify the Kosaraju–Sharir algorithm to replace the second depth-first search with breadth-first search, then it will still find the strong components.

**4.2.24** Compute the memory usage of a Digraph with  $V$  vertices and  $E$  edges, under the memory cost model of SECTION 1.4.

**4.2.25** How many edges are there in the transitive closure of a digraph that is a simple directed path with  $V$  vertices and  $V-1$  edges?

**4.2.26** Give the transitive closure of the digraph with ten vertices and these edges:

3->7 1->4 7->8 0->5 5->2 3->8 2->9 0->6 4->9 2->6 6->4

## CREATIVE PROBLEMS

**4.2.27 Topological sort and BFS.** Explain why the following algorithm does not necessarily produce a topological order: Run BFS, and label the vertices by increasing distance to their respective source.

**4.2.28 Directed Eulerian cycle.** A directed Eulerian cycle is a directed cycle that contains each edge exactly once. Write a `Digraph` client `DirectedEulerianCycle` that finds a directed Eulerian cycle or reports that no such cycle exists. *Hint:* Prove that a digraph  $G$  has a directed Eulerian cycle if and only if  $G$  is strongly connected and each vertex has its indegree equal to its outdegree.

**4.2.29 LCA in a DAG.** Given a DAG and two vertices  $v$  and  $w$ , develop an algorithm to find a *lowest common ancestor* (LCA) of  $v$  and  $w$ . In a tree, the LCA of  $v$  and  $w$  is the (unique) vertex farthest from the root that is an ancestor of both  $v$  and  $w$ . In a DAG, an LCA of  $v$  and  $w$  is an ancestor of  $v$  and  $w$  that has no descendants that are also ancestors of  $v$  and  $w$ . Computing an LCA is useful in multiple inheritance in programming languages, analysis of genealogical data (find degree of inbreeding in a pedigree graph), and other applications. *Hint:* Define the height of a vertex  $v$  in a DAG to be the length of the longest direct path from a source (vertex with indegree 0) to  $v$ . Among vertices that are ancestors of both  $v$  and  $w$ , the one with the greatest height is an LCA of  $v$  and  $w$ .

**4.2.30 Shortest ancestral path.** Given a DAG and two vertices  $v$  and  $w$ , find a *shortest ancestral path* between  $v$  and  $w$ . An ancestral path between  $v$  and  $w$  is a common ancestor  $x$  along with a shortest directed path from  $v$  to  $x$  and a shortest directed path from  $w$  to  $x$ . A shortest ancestral path is the ancestral path whose total length is minimized. *Warmup:* Find a DAG where the shortest ancestral path goes to a common ancestor  $x$  that is not an LCA. *Hint:* Run BFS twice, once from  $v$  and once from  $w$ .

**4.2.31 Strong component.** Describe a linear-time algorithm for computing the strong component containing a given vertex  $v$ . On the basis of that algorithm, describe a simple quadratic-time algorithm for computing the strong components of a digraph.

**4.2.32 Hamiltonian path in DAGs.** Given a DAG, design a linear-time algorithm to determine whether there is a directed path that visits each vertex exactly once.

**CREATIVE PROBLEMS (continued)**

**4.2.33 Unique topological ordering.** Design an algorithm to determine whether a DAG has a unique topological ordering. *Hint:* A DAG has a unique topological ordering if and only if there is a directed edge between each pair of consecutive vertices in a topological order (i.e., the digraph has a Hamiltonian path). If the DAG has multiple topological orderings, then a second topological order can be obtained by swapping any pair of consecutive and nonadjacent vertices.

**4.2.34 2-satisfiability.** Given a boolean formula in conjunctive normal form with  $M$  clauses and  $N$  variables such that each clause has exactly two literals (where a literal is either a variable or its negation), find a satisfying assignment (if one exists). *Hint:* Form the *implication digraph* with  $2N$  vertices (one per literal). For each clause  $x + y$ , include edges from  $y'$  to  $x$  and from  $x'$  to  $y$ . *Claim:* The formula is satisfiable if and only if no literal  $x$  is in the same strong component as its negation  $x'$ . Moreover, a topological sort of the *kernel DAG* (contract each strong component to a single vertex) yields a satisfying assignment.

**4.2.35 Digraph enumeration.** Show that the number of different  $V$ -vertex digraphs with no parallel edges is  $2^{V^2}$ . (How many digraphs are there that contain  $V$  vertices and  $E$  edges?) Then compute an upper bound on the percentage of 20-vertex digraphs that could ever be examined by any computer, under the assumptions that every electron in the universe examines a digraph every nanosecond, that the universe has fewer than  $10^{80}$  electrons, and that the age of the universe will be less than  $10^{20}$  years.

**4.2.36 DAG enumeration.** Give a formula for the number of  $V$ -vertex DAGs with  $E$  edges.

**4.2.37 Arithmetic expressions.** Write a class that evaluates DAGs that represent arithmetic expressions. Use a vertex-indexed array to hold values corresponding to each vertex. Assume that values corresponding to leaves (vertex with outdegree 0) have been established. Describe a family of arithmetic expressions with the property that the size of the expression tree is exponentially larger than the size of the corresponding DAG (so the running time of your program for the DAG is proportional to the logarithm of the running time for the tree).

**4.2.38 Euclidean digraphs.** Modify your solution to EXERCISE 4.1.37 to create an API `EuclideanDigraph` for digraphs whose vertices are points in the plane, so that you can work with graphical representations.

**4.2.39 Queue-based topological sort.** Develop a topological sort implementation that maintains a vertex-indexed array that keeps track of the indegree of each vertex. Initialize the array and a queue of sources in a single pass through all the edges, as in EXERCISE 4.2.7. Then, perform the following operations until the source queue is empty:

- Remove a source from the queue and label it.
- Decrement the entries in the indegree array corresponding to the destination vertex of each of the removed vertex's edges.
- If decrementing any entry causes it to become 0, insert the corresponding vertex onto the source queue.

**4.2.40 Shortest directed cycle.** Given a digraph, design an algorithm to find a directed cycle with the minimum number of edges (or report that the graph is acyclic). The running time of your algorithm should be proportional to  $E V$  in the worst case.

**4.2.41 Odd-length directed cycle.** Design a linear-time algorithm to determine whether a digraph has an odd-length directed cycle.

**4.2.42 Reachable vertex in a DAG.** Design a linear-time algorithm to determine whether a DAG has a vertex that is reachable from every other vertex.

**4.2.43 Reachable vertex in a digraph.** Design a linear-time algorithm to determine whether a digraph has a vertex that is reachable from every other vertex.

**4.2.44 Web crawler.** Write a program that uses breadth-first search to crawl the web digraph, starting from a given web page. Do not explicitly build the web digraph.

## EXPERIMENTS

**4.2.45 Random digraphs.** Write a program `ErdosRenyiDigraph` that takes integer values  $V$  and  $E$  from the command line and builds a digraph by generating  $E$  random pairs of integers between 0 and  $V-1$ . Note: This generator produces self-loops and parallel edges.

**4.2.46 Random simple digraphs.** Write a program `RandomDigraph` that takes integer values  $V$  and  $E$  from the command line and produces, with equal likelihood, each of the possible *simple* digraphs with  $V$  vertices and  $E$  edges.

**4.2.47 Random sparse digraphs.** Modify your solution to EXERCISE 4.1.41 to create a program `RandomSparseDigraph` that generates random sparse digraphs for a well-chosen set of values of  $V$  and  $E$  that you can use it to run meaningful empirical tests.

**4.2.48 Random Euclidean digraphs.** Modify your solution to EXERCISE 4.1.42 to create a `EuclideanDigraph` client `RandomEuclideanDigraph` that assigns a random direction to each edge.

**4.2.49 Random grid digraphs.** Modify your solution to EXERCISE 4.1.43 to create a `EuclideanDiGraph` client `RandomGridDigraph` that assigns a random direction to each edge.

**4.2.50 Real-world digraphs.** Find a large digraph somewhere online—perhaps a transaction graph in some online system, or a digraph defined by links on web pages. Write a program `RandomRealDigraph` that builds a graph by choosing  $V$  vertices at random and  $E$  directed edges at random from the subgraph induced by those vertices.

**4.2.51 Real-world DAG.** Find a large DAG somewhere online—perhaps one defined by class-definition dependencies in a large software system, or by directory links in a large file system. Write a program `RandomRealDAG` that builds a graph by choosing  $V$  vertices at random and  $E$  directed edges at random from the subgraph induced by those vertices.

*Testing all algorithms and studying all parameters against all graph models is unrealistic. For each problem listed below, write a client that addresses the problem for any given input graph, then choose among the generators above to run experiments for that graph model. Use your judgment in selecting experiments, perhaps in response to results of previous experiments. Write a narrative explaining your results and any conclusions that might be drawn.*

**4.2.52 Reachability.** Run experiments to determine empirically the average number of vertices that are reachable from a randomly chosen vertex, for various digraph models.

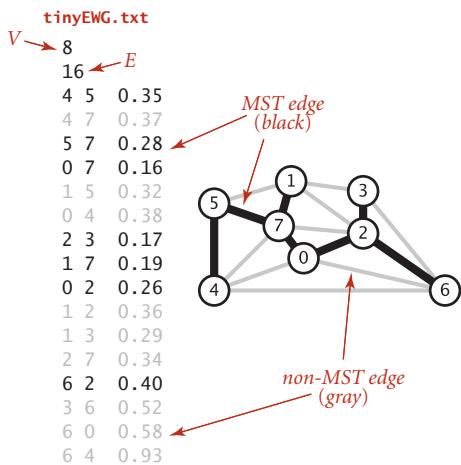
**4.2.53 Path lengths in DFS.** Run experiments to determine empirically the probability that `DepthFirstDirectedPaths` finds a path between two randomly chosen vertices and to calculate the average length of the paths found, for various random digraph models.

**4.2.54 Path lengths in BFS.** Run experiments to determine empirically the probability that `BreadthFirstDirectedPaths` finds a path between two randomly chosen vertices and to calculate the average length of the paths found, for various random digraph models.

**4.2.55 Strong components.** Run experiments to determine empirically the distribution of the number of strong components in random digraphs of various types, by generating large numbers of digraphs and drawing a histogram.

## 4.3 MINIMUM SPANNING TREES

AN *edge-weighted graph* is a graph model where we associate *weights* or *costs* with each edge. Such graphs are natural models for many applications. In an airline map where edges represent flight routes, these weights might represent distances or fares. In an electric circuit where edges represent wires, the weights might represent the length of the wire, its cost, or the time that it takes a signal to propagate through it. Minimizing cost is naturally of interest in such situations. In this section, we consider *undirected* edge-weighted graph models and examine algorithms for one such problem:



An edge-weighted graph and its MST

**Minimum spanning tree.** Given an undirected edge-weighted graph, find an MST.

**Definition.** Recall that a *spanning tree* of a graph is a connected subgraph with no cycles that includes all the vertices. A *minimum spanning tree* (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

In this section, we examine two classical algorithms for computing MSTs: *Prim's algorithm* and *Kruskal's algorithm*. These algorithms are easy to understand and not difficult to implement. They are among the oldest and most well-known algorithms in this book, and they also take good advantage of modern data structures. Since MSTs have numerous important applications, al-

gorithms to solve the problem have been studied at least since the 1920s, at first in the context of power distribution networks, later in the context of telephone networks. MST algorithms are now important in the design of many types of networks (communication, electrical, hydraulic, computer, road, rail, air, and many others) and also in the study of biological, chemical, and physical networks that are found in nature.

application	vertex	edge
circuit	component	wire
airline	airport	flight route
power distribution	power plant	transmission lines
image analysis	feature	proximity relationship

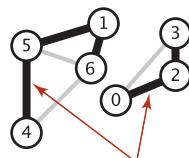
Typical MST applications

**Assumptions.** Various anomalous situations, which are generally easy to handle, can arise when computing minimum spanning trees. To streamline the presentation, we adopt the following conventions:

- *The graph is connected.* The spanning-tree condition in our definition implies that the graph must be connected for an MST to exist. Another way to pose the problem, recalling basic properties of trees from SECTION 4.1, is to find a minimal-weight set of  $V - 1$  edges that connect the graph. If a graph is not connected, we can adapt our algorithms to compute the MSTs of each of its connected components, collectively known as a *minimum spanning forest* (see EXERCISE 4.3.22).
- *The edge weights are not necessarily distances.* Geometric intuition is sometimes beneficial in understanding algorithms, so we use examples where vertices are points in the plane and weights are distances, such as the graph on the facing page. But it is important to remember that the weights might represent time or cost or an entirely different variable and do not need to be proportional to a distance at all.
- *The edge weights may be zero or negative.* If the edge weights are all positive, it suffices to define an MST as a subgraph with minimal total weight that connects all the vertices, as such a subgraph must form a spanning tree. The spanning-tree condition in the definition is included so that it applies to graphs that may have zero or negative edge weights.
- *The edge weights are all different.* If edges can have equal weights, the minimum spanning tree may not be unique (see EXERCISE 4.3.2). The possibility of multiple MSTs complicates the correctness proofs of some of our algorithms, so we rule out that possibility in the presentation. It turns out that this assumption is not restrictive because our algorithms work without modification in the presence of equal weights.

In summary, we assume throughout the presentation that our job is to find the MST of a connected edge-weighted graph with arbitrary (but distinct) weights.

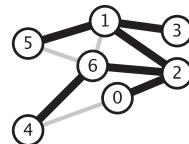
no MST if graph is not connected



4	5	0.61
4	6	0.62
5	6	0.88
1	5	0.11
2	3	0.35
0	3	0.6
1	6	0.10
0	2	0.22

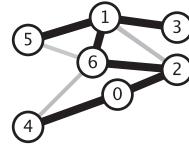
can independently compute  
MSTs of components

weights need not be  
proportional to distance



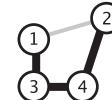
4	6	0.62
5	6	0.88
1	5	0.02
0	4	0.64
1	6	0.90
0	2	0.22
1	2	0.50
1	3	0.97
2	6	0.17

weights can be 0 or negative

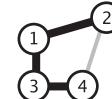


4	6	0.62
5	6	0.88
1	5	0.02
0	4	-0.99
1	6	0
0	2	0.22
1	2	0.50
1	3	0.97
2	6	0.17

MST may not be unique  
when weights have equal values



1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50



1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

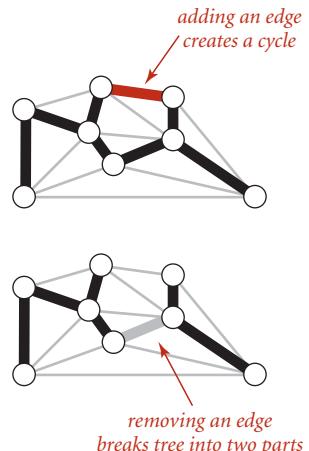
Various MST anomalies

**Underlying principles** To begin, we recall from SECTION 4.1 two of the defining properties of a tree:

- Adding an edge that connects two vertices in a tree creates a unique cycle.
- Removing an edge from a tree breaks it into two separate subtrees.

These properties are the basis for proving a fundamental property of MSTs that leads to the MST algorithms that we consider in this section.

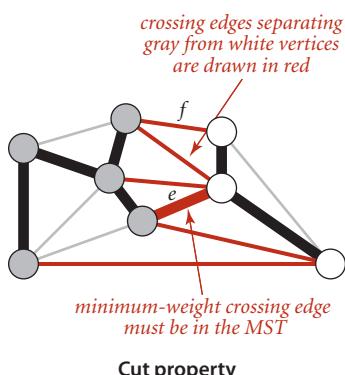
**Cut property.** This property, which we refer to as the *cut property*, has to do with identifying edges that must be in the MST of a given edge-weighted graph, by dividing vertices into two sets and examining edges that cross the division.



Basic properties of a tree

**Definition.** A *cut* of a graph is a partition of its vertices into two nonempty disjoint sets. A *crossing edge* of a cut is an edge that connects a vertex in one set with a vertex in the other.

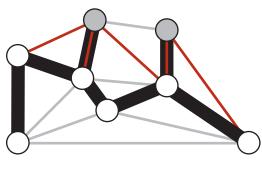
Typically, we specify a cut by specifying a set of vertices, leaving implicit the assumption that the cut comprises the given vertex set and its complement, so that a crossing edge is an edge from a vertex in the set to a vertex not in the set. In figures, we draw vertices on one side of the cut in gray and vertices on the other side in white.



Cut property

**Proposition J. (Cut property)** Given any cut in an edge-weighted graph, the crossing edge of minimum weight is in the MST of the graph.

**Proof:** Let  $e$  be the crossing edge of minimum weight and let  $T$  be the MST. The proof is by contradiction: Suppose that  $T$  does not contain  $e$ . Now consider the graph formed by adding  $e$  to  $T$ . This graph has a cycle that contains  $e$ , and that cycle must contain at least one other crossing edge—say,  $f$ , which has higher weight than  $e$  (since  $e$  is minimal and all edge weights are different). We can get a spanning tree of strictly lower weight by deleting  $f$  and adding  $e$ , contradicting the assumed minimality of  $T$ .



A cut with two MST edges

Under our assumption that edge weights are distinct, every connected graph has a unique MST (see EXERCISE 4.3.3); and the cut property says that the lightest crossing edge for every cut must be in the MST.

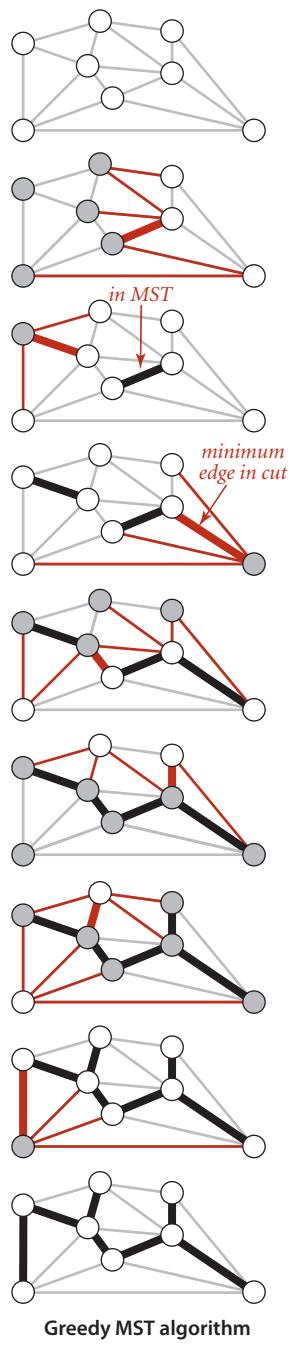
The figure to the left of PROPOSITION J illustrates the cut property. Note that there is no requirement that the minimal edge be the *only* MST edge connecting the two sets; indeed, for typical cuts there are several MST edges that connect a vertex in one set with a vertex in the other, as illustrated in the figure above.

**Greedy algorithm.** The cut property is the basis for the algorithms that we consider for the MST problem. Specifically, they are special cases of a general paradigm known as the *greedy algorithm*: apply the cut property to accept an edge as an MST edge, continuing until finding all of the MST edges. Our algorithms differ in their approaches to maintaining cuts and identifying the crossing edge of minimum weight, but are special cases of the following:

**Proposition K. (Greedy MST algorithm)** The following method colors black all edges in the the MST of any connected edge-weighted graph with  $V$  vertices: starting with all edges colored gray, find a cut with no black edges, color its minimum-weight edge black, and continue until  $V - 1$  edges have been colored black.

**Proof:** For simplicity, we assume in the discussion that the edge weights are all different, though the proposition is still true when that is not the case (see EXERCISE 4.3.5). By the cut property, any edge that is colored black is in the MST. If fewer than  $V - 1$  edges are black, a cut with no black edges exists (recall that we assume the graph to be connected). Once  $V - 1$  edges are black, the black edges form a spanning tree.

The diagram at right is a typical trace of the greedy algorithm. Each drawing depicts a cut and identifies the minimum-weight edge in the cut (thick red) that is added to the MST by the algorithm.



Greedy MST algorithm

**Edge-weighted graph data type** How should we represent edge-weighted graphs? Perhaps the simplest way to proceed is to extend the basic graph representations from SECTION 4.1: in the adjacency-matrix representation, the matrix can contain edge weights rather than boolean values; in the adjacency-lists representation, we can define a node that contains both a vertex and a weight field to put in the adjacency lists. (As usual, we focus on sparse graphs and leave the adjacency-matrix representation for exercises.) This classic approach is appealing, but we will use a different method that is not much more complicated, will make our programs useful in more general settings, and needs a slightly more general API, which allows us to process Edge objects:

---

```
public class Edge implements Comparable<Edge>
    Edge(int v, int w, double weight) initializing constructor
    double weight() weight of this edge
    int either() either of this edge's vertices
    int other(int v) the other vertex
    int compareTo(Edge that) compare this edge to that
    String toString() string representation
```

API for a weighted edge

---

The either() and other() methods for accessing the edge's vertices may be a bit puzzling at first—the need for them will become plain when we examine client code. You can find an implementation of Edge on page 610. It is the basis for this EdgeWeightedGraph API, which refers to Edge objects in a natural manner:

---

```
public class EdgeWeightedGraph
    EdgeWeightedGraph(int V) create an empty V-vertex graph
    EdgeWeightedGraph(In in) read graph from input stream
    int V() number of vertices
    int E() number of edges
    void addEdge(Edge e) add edge e to this graph
    Iterable<Edge> adj(int v) edges incident to v
    Iterable<Edge> edges() all of this graph's edges
    String toString() string representation
```

API for an edge-weighted graph

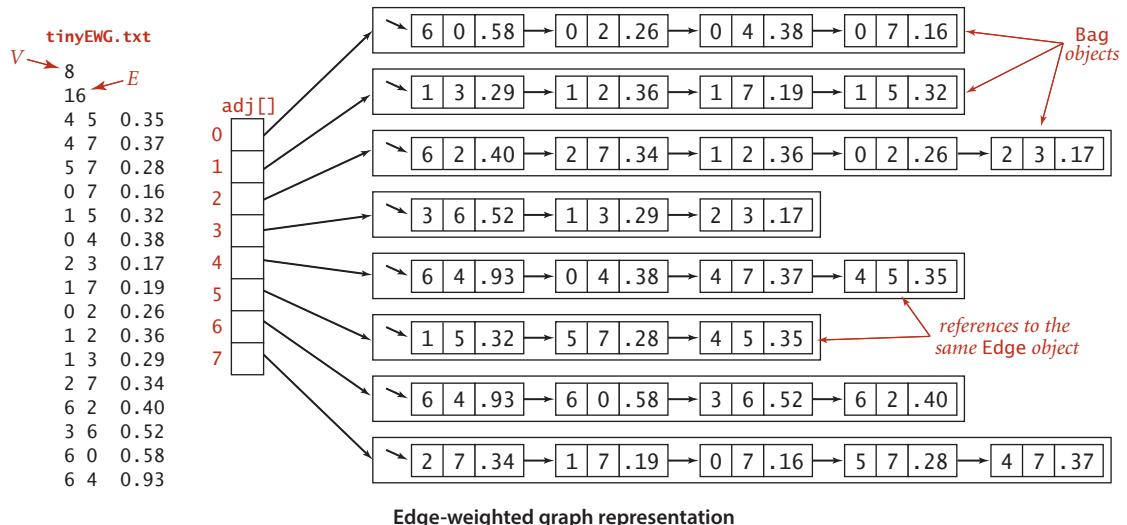
---

This API is very similar to the API for Graph (page 522). The two important differences are that it is based on Edge and that it adds the edges() method at right, which provides clients with the ability to iterate through all the graph's edges (ignoring any self-loops). The rest of the implementation of EdgeWeightedGraph on page 611 is quite similar to the unweighted undirected graph implementation of SECTION 4.1, but instead of the adjacency lists of integers used in Graph, it uses adjacency lists of Edge objects.

The figure at the bottom of this page shows the edge-weighted graph representation that EdgeWeightedGraph builds from the sample file `tinyEWG.txt`, showing the contents of each Bag as a linked list to reflect the standard implementation of SECTION 1.3. To reduce clutter in the figure, we show each Edge as a pair of int values and a double value. The actual data structure is a linked list of links to objects containing those values. In particular, although there are two *references* to each Edge (one in the list for each vertex), there is only one Edge object corresponding to each graph edge. In the figure, the edges appear in each list in reverse order of the order they are processed, because of the stack-like nature of the standard linked-list implementation. As in Graph, by using a Bag we are making clear that our client code makes no assumptions about the order of objects in the lists.

```
public Iterable<Edge> edges()
{
    Bag<Edge> b = new Bag<Edge>();
    for (int v = 0; v < V; v++)
        for (Edge e : adj[v])
            if (e.other(v) > v) b.add(e);
    return b;
}
```

Gathering all the edges in an edge-weighted graph



## Weighted edge data type

```
public class Edge implements Comparable<Edge>
{
    private final int v;                                // one vertex
    private final int w;                                // the other vertex
    private final double weight;                         // edge weight

    public Edge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public double weight()
    { return weight; }

    public int either()
    { return v; }

    public int other(int vertex)
    {
        if      (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new RuntimeException("Inconsistent edge");
    }

    public int compareTo(Edge that)
    {
        if      (this.weight() < that.weight()) return -1;
        else if (this.weight() > that.weight()) return +1;
        else                               return 0;
    }

    public String toString()
    { return String.format("%d-%d %.5f", v, w, weight); }
}
```

This data type provides the methods `either()` and `other()` so that a client can use `other(v)` to find the other vertex when it knows `v`. When neither vertex is known, a client can use the idiomatic code `int v = e.either(), w = e.other(v);` to access an `Edge e`'s two vertices.

## Edge-weighted graph data type

```
public class EdgeWeightedGraph
{
    private final int V;                      // number of vertices
    private int E;                            // number of edges
    private Bag<Edge>[] adj;                // adjacency lists

    public EdgeWeightedGraph(int V)
    {
        this.V = V;
        this.E = 0;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Edge>();
    }

    public EdgeWeightedGraph(In in)
    // See Exercise 4.3.9.

    public int V() { return V; }
    public int E() { return E; }

    public void addEdge(Edge e)
    {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
        E++;
    }

    public Iterable<Edge> adj(int v)
    { return adj[v]; }

    public Iterable<Edge> edges()
    // See page 609.
}
```

---

This implementation maintains a vertex-indexed array of lists of edges. As with `Graph` (see page 526), every edge appears twice: if an edge connects  $v$  and  $w$ , it appears both in  $v$ 's list and in  $w$ 's list. The `edges()` method puts all the edges in a `Bag` (see page 609). The `toString()` implementation is left as an exercise.

**Comparing edges by weight.** The API specifies that the Edge class must implement the Comparable interface and include a `compareTo()` implementation. The natural ordering for edges in an edge-weighted graph is by weight. Accordingly, the implementation of `compareTo()` is straightforward.

**Parallel edges.** As with our undirected-graph implementations, we allow parallel edges. Alternatively, we could develop a more complicated implementation of `EdgeWeightedGraph` that disallows them, perhaps keeping the minimum-weight edge from a set of parallel edges.

**Self-loops.** We allow self-loops. However, our `edges()` implementation in `EdgeWeightedGraph` does not include self-loops even though they might be present in the input or in the data structure. This omission has no effect on our MST algorithms because no MST contains a self-loop. When working with an application where self-loops are significant, you may need to modify our code as appropriate for the application.

OUR CHOICE TO USE explicit `Edge` objects leads to clear and compact client code, as you will see. It carries a small price: each adjacency-list node has a *reference* to an `Edge` object, with redundant information (all the nodes on  $v$ 's adjacency list have a  $v$ ). We also pay object overhead cost. Although we have only one copy of each `Edge`, we do have two references to each `Edge` object. An alternative and widely used approach is to keep two list nodes corresponding to each edge, just as in `Graph`, each with a vertex and the edge weight in each list node. This alternative also carries a price—two nodes, including two copies of the weight for each edge.

**MST API and test client** As usual, for graph processing, we define an API where the constructor takes an edge-weighted graph as argument and supports client query methods that return the MST and its weight. How should we represent the MST itself? The MST of a graph  $G$  is a subgraph of  $G$  that is also a tree, so we have numerous options. Chief among them are

- A list of edges
- An edge-weighted graph
- A vertex-indexed array with parent links

To give clients and our implementations as much flexibility as possible in choosing among these alternatives for various applications, we adopt the following API:

---

<code>public class MST</code>		
	<code>MST(EdgeWeightedGraph G)</code>	<i>constructor</i>
	<code>Iterable&lt;Edge&gt; edges()</code>	<i>all of the MST edges</i>
	<code>double weight()</code>	<i>weight of MST</i>
<code>API for MST implementations</code>		

**Test client.** As usual, we create sample graphs and develop a test client for use in testing our implementations. A sample client is shown below. It reads edges from the input stream, builds an edge-weighted graph, computes the MST of that graph, prints the MST edges, and prints the total weight of the MST.

```
public static void main(String[] args)
{
    In in = new In(args[0]);
    EdgeWeightedGraph G;
    G = new EdgeWeightedGraph(in);

    MST mst = new MST(G);
    for (Edge e : mst.edges())
        StdOut.println(e);
    StdOut.printf("%.5f\n", mst.weight());
}
```

MST test client

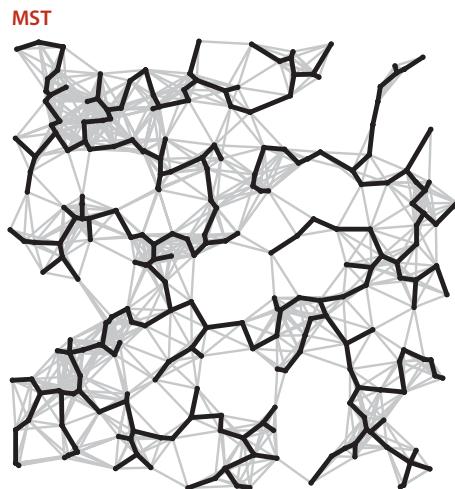
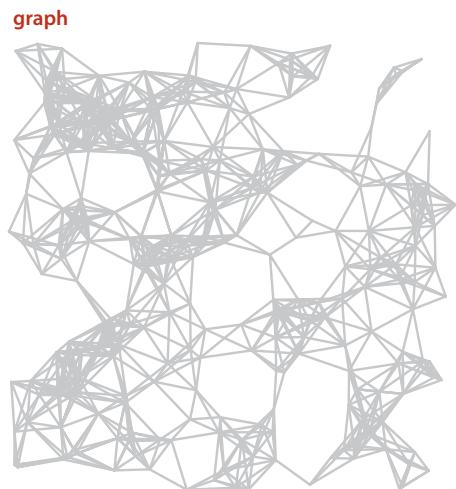
**Test data.** You can find the file `tinyEWG.txt` on the booksite, which defines the small sample graph on page 604 that we use for detailed traces of MST algorithms. You can also find on the booksite the file `mediumEWG.txt`, which defines the weighted graph with 250 vertices that is drawn on bottom of the facing page. It is an example of a *Euclidean graph*, whose vertices are points in the plane and whose edges are lines connecting them with weights equal to their Euclidean distances. Such graphs are useful for gaining insight into the behavior of MST algorithms, and they also model many of the typical practical problems we have mentioned, such as road maps or electric circuits. You can also find on the booksite a larger example `largeEWG.txt` that defines a Euclidean graph with 1 million vertices. Our goal is to be able to find the MST of such a graph in a reasonable amount of time.

```
% more tinyEWG.txt
8 16
4 5 .35
4 7 .37
5 7 .28
0 7 .16
1 5 .32
0 4 .38
2 3 .17
1 7 .19
0 2 .26
1 2 .36
1 3 .29
2 7 .34
6 2 .40
3 6 .52
6 0 .58
6 4 .93

% java MST tinyEWG.txt
0-7 0.16000
2-3 0.17000
1-7 0.19000
0-2 0.26000
5-7 0.28000
4-5 0.35000
6-2 0.40000
1.81000
```

```
% more mediumEWG.txt
250 1273
244 246 0.11712
239 240 0.10616
238 245 0.06142
235 238 0.07048
233 240 0.07634
232 248 0.10223
231 248 0.10699
229 249 0.10098
228 241 0.01473
226 231 0.07638
...
[1263 more edges]

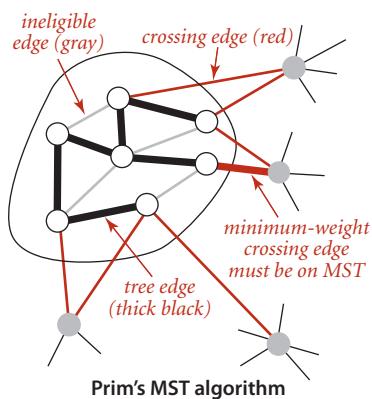
% java MST mediumEWG.txt
0 225 0.02383
49 225 0.03314
44 49 0.02107
44 204 0.01774
49 97 0.03121
202 204 0.04207
176 202 0.04299
176 191 0.02089
68 176 0.04396
58 68 0.04795
...
[239 more edges]
10.46351
```



A 250-vertex Euclidean graph (with 1,273 edges) and its MST

**Prim's algorithm**

Our first MST method, known as *Prim's algorithm*, is to attach a new edge to a single growing tree at each step. Start with any vertex as a single-vertex tree; then add  $V - 1$  edges to it, always taking next (coloring black) the minimum-weight edge that connects a vertex on the tree to a vertex not yet on the tree (a crossing edge for the cut defined by tree vertices).



**Proposition L.** Prim's algorithm computes the MST of any connected edge-weighted graph.

**Proof:** Immediate from PROPOSITION K. The growing tree defines a cut with no black edges; the algorithm takes the crossing edge of minimal weight, so it is successively coloring edges black in accordance with the greedy algorithm.

The one-sentence description of Prim's algorithm just given leaves unanswered a key question: How do we (efficiently) find the crossing edge of minimal weight? Several methods have been proposed—we will discuss some of them after we have developed a full solution based on a particularly simple approach.

**Data structures.** We implement Prim's algorithm with the aid of a few simple and familiar data structures. In particular, we represent the vertices on the tree, the edges on the tree, and the crossing edges, as follows:

- *Vertices on the tree:* We use a vertex-indexed boolean array `marked[]`, where `marked[v]` is `true` if  $v$  is on the tree.
- *Edges in the tree:* We use one of two data structures: either a queue `mst` to collect edges in the MST or a vertex-indexed array `edgeTo[]` of `Edge` objects, where `edgeTo[v]` is the `Edge` that connects  $v$  to the tree.
- *Crossing edges:* We use a `MinPQ<Edge>` priority queue that compares edges by weight (see page 610).

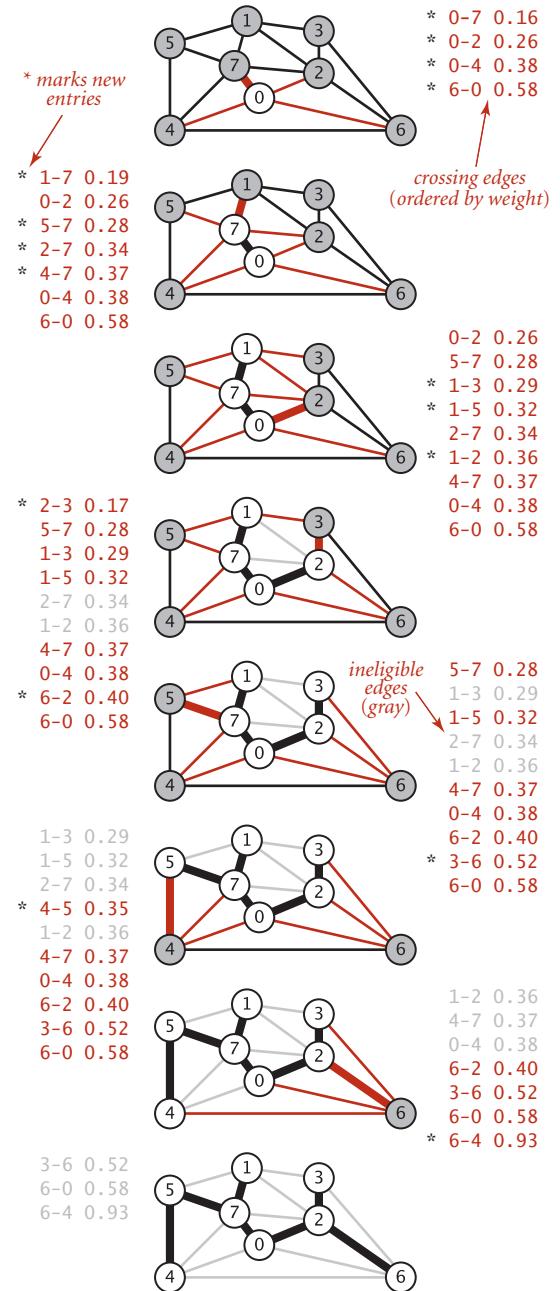
These data structures allow us to directly answer the basic question “Which is the minimal-weight crossing edge?”

**Maintaining the set of crossing edges.** Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex (using `marked[]` to identify such edges). But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes *ineligible* (it is no longer a crossing edge because it connects two tree vertices). An *eager* implementation

of Prim's algorithm would remove such edges from the priority queue; we first consider a simpler *lazy* implementation of the algorithm where we leave such edges on the priority queue, deferring the eligibility test to when we remove them.

The figure at right is a trace for our small sample graph `tinyEWG.txt`. Each drawing depicts the graph and the priority queue just after a vertex is visited (added to the tree and the edges in its adjacency list processed). The contents of the priority queue are shown in order on the side, with new edges marked with asterisks. The algorithm builds the MST as follows:

- Adds 0 to the MST and all edges in its adjacency list to the priority queue.
- Adds 7 and 0-7 to the MST and all edges in its adjacency list to the priority queue.
- Adds 1 and 1-7 to the MST and all edges in its adjacency list to the priority queue.
- Adds 2 and 0-2 to the MST and edges 2-3 and 6-2 to the priority queue. Edges 2-7 and 1-2 become ineligible.
- Adds 3 and 2-3 to the MST and edge 3-6 to the priority queue. Edge 1-3 becomes ineligible.
- Adds 5 and 5-7 to the MST and edge 4-5 to the priority queue. Edge 1-5 becomes ineligible.
- Removes ineligible edges 1-3, 1-5, and 2-7 from the priority queue.
- Adds 4 and 4-5 to the MST and edge 6-4 to the priority queue. Edges 4-7 and 0-4 become ineligible.
- Removes ineligible edges 1-2, 4-7, and 0-4 from the priority queue.
- Adds 6 and 6-2 to the MST. The other edges incident to 6 become ineligible.



Trace of Prim's algorithm (lazy version)

After having added  $V$  vertices (and  $V - 1$  edges), the MST is complete. The remaining edges on the priority queue are ineligible, so we need not examine them again.

**Implementation.** With these preparations, implementing Prim's algorithm is straightforward, as shown in the implementation `LazyPrimMST` on the facing page. As with our depth-first search and breadth-first search implementations in the previous two sections, it computes the MST in the constructor so that client methods can learn properties of the MST with query methods. We use a private method `visit()` that puts a vertex on the tree, by marking it as visited and then putting all of its incident edges that are not ineligible onto the priority queue, thus ensuring that the priority queue contains the crossing edges from tree vertices to non-tree vertices (perhaps also some ineligible edges). The inner loop is a rendition in code of the one-sentence description of the algorithm: we take an edge from the priority queue and (if it is not ineligible) add it to the tree, and also add to the tree the new vertex that it leads to, updating the set of crossing edges by calling `visit()` with that vertex as argument. The `weight()` method requires iterating through the tree edges to add up the edge weights (lazy approach) or keeping a running total in an instance variable (eager approach) and is left as EXERCISE 4.3.31.

**Running time.** How fast is Prim's algorithm? This question is not difficult to answer, given our knowledge of the behavior characteristics of priority queues:

**Proposition M.** The lazy version of Prim's algorithm uses space proportional to  $E$  and time proportional to  $E \log E$  (in the worst case) to compute the MST of a connected edge-weighted graph with  $E$  edges and  $V$  vertices.

**Proof:** The bottleneck in the algorithm is the number of edge-weight comparisons in the priority-queue methods `insert()` and `delMin()`. The number of edges on the priority queue is at most  $E$ , which gives the space bound. In the worst case, the cost of an insertion is  $\sim \lg E$  and the cost to delete the minimum is  $\sim 2\lg E$  (see PROPOSITION Q in CHAPTER 2). Since at most  $E$  edges are inserted and at most  $E$  are deleted, the time bound follows.

In practice, the upper bound on the running time is a bit conservative because the number of edges on the priority queue is typically much less than  $E$ . The existence of such a simple, efficient, and useful algorithm for such a challenging task is quite remarkable. Next, we briefly discuss some improvements. As usual, detailed evaluation of such improvements in performance-critical applications is a job for experts.

## Lazy version of Prim's MST algorithm

```
public class LazyPrimMST
{
    private boolean[] marked;           // MST vertices
    private Queue<Edge> mst;          // MST edges
    private MinPQ<Edge> pq;           // crossing (and ineligible) edges

    public LazyPrimMST(EdgeWeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        marked = new boolean[G.V()];
        mst = new Queue<Edge>();

        visit(G, 0);      // assumes G is connected (see Exercise 4.3.22)
        while (!pq.isEmpty())
        {
            Edge e = pq.delMin();           // Get lowest-weight
            int v = e.either(), w = e.other(v); // edge from pq.
            if (marked[v] && marked[w]) continue; // Skip if ineligible.
            mst.enqueue(e);                // Add edge to tree.
            if (!marked[v]) visit(G, v);    // Add vertex to tree
            if (!marked[w]) visit(G, w);    // (either v or w).
        }
    }

    private void visit(EdgeWeightedGraph G, int v)
    { // Mark v and add to pq all edges from v to unmarked vertices.
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
    }

    public Iterable<Edge> edges()
    { return mst; }

    public double weight() // See Exercise 4.3.31.
    { }
```

---

This implementation of Prim's algorithm uses a priority queue to hold crossing edges, a vertex-indexed array to mark tree vertices, and a queue to hold MST edges. This implementation is a lazy approach where we leave ineligible edges in the priority queue.

**Eager version of Prim's algorithm** To improve the `LazyPrimMST`, we might try to delete ineligible edges from the priority queue, so that the priority queue contains *only* the crossing edges between tree vertices and non-tree vertices. But we can eliminate even more edges. The key is to note that our only interest is in the *minimal* edge

from each non-tree vertex to a tree vertex. When we add a vertex  $v$  to the tree, the only possible change with respect to each non-tree vertex  $w$  is that adding  $v$  brings  $w$  closer than before to the tree. In short, we do not need to keep on the priority queue *all* of the edges from  $w$  to tree vertices—we just need to keep track of the minimum-weight edge and check whether the addition of  $v$  to the tree necessitates that we update that minimum (because of an edge  $v-w$  that has lower weight), which we can do as we process each edge in  $v$ 's adjacency list. In other words, we main-

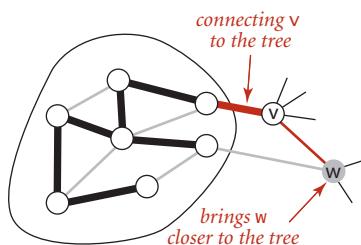
tain on the priority queue just *one* edge for each non-tree vertex  $w$ : the lightest edge that connects it to the tree. Any heavier edge connecting  $w$  to the tree will become ineligible at some point, so there is no need to keep it on the priority queue.

`PrimMST` (ALGORITHM 4.7 on page 622) implements Prim's algorithm using our index priority queue data type from SECTION 2.4 (see page 320). It replaces the data structure `mst[]` in `LazyPrimMST` by two vertex-indexed arrays `edgeTo[]` and `distTo[]`, which have the following properties:

- If  $v$  is not on the tree but has at least one edge connecting it to the tree, then `edgeTo[v]` is the lightest edge connecting  $v$  to the tree, and `distTo[v]` is the weight of that edge.
- All such vertices  $v$  are maintained on the index priority queue, as an index  $v$  associated with the weight of `edgeTo[v]`.

The key implications of these properties is that *the minimum key on the priority queue is the weight of the minimal-weight crossing edge, and its associated vertex v is the next to add to the tree*. To maintain the data structures, `PrimMST` takes a vertex  $v$  from the priority queue, then checks each edge  $v-w$  on its adjacency list. If  $w$  is marked, the edge is ineligible; if it is not on the priority queue or its weight is lower than the current best-known `edgeTo[w]`, the code updates the data structures to establish  $v-w$  as the best-known way to connect  $w$  to the tree.

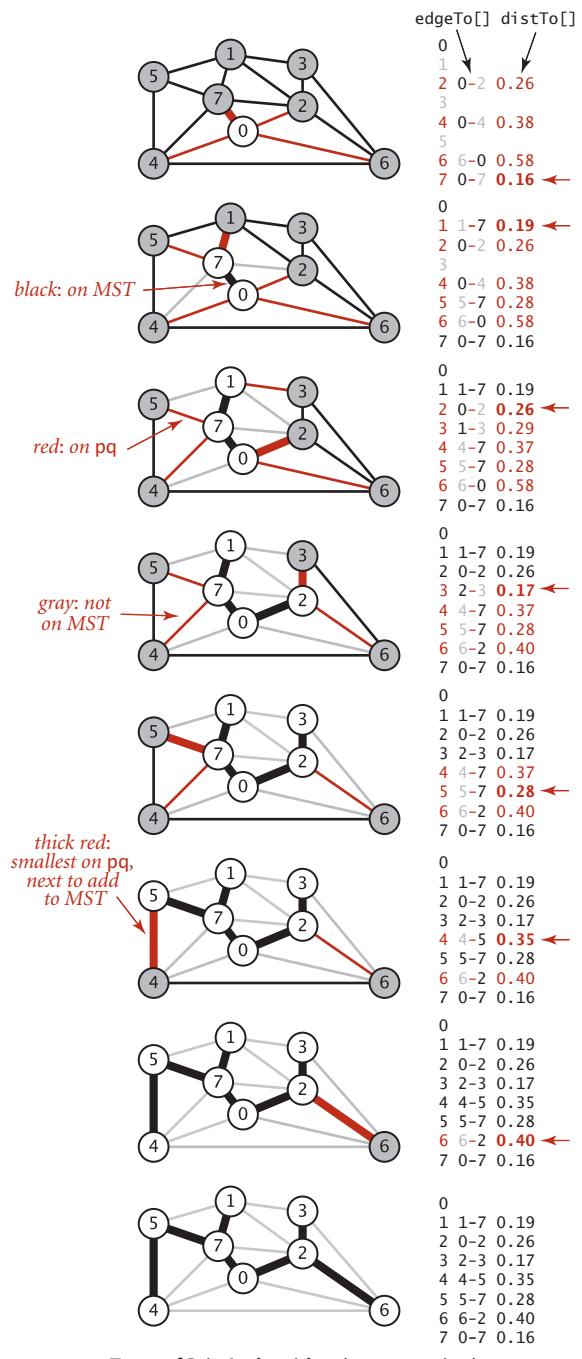
The figure on the facing page is a trace of `PrimMST` for our small sample graph `tinyEWG.txt`. The contents of the `edgeTo[]` and `distTo[]` arrays are depicted after each vertex is added to the MST, color-coded to depict the MST vertices (index in black), the non-MST vertices (index in gray), the MST edges (in black), and the priority-queue index/value pairs (in red). In the drawings, the lightest edge connecting each non-MST vertex to an MST vertex is drawn in red. The algorithm adds edges to



the MST in the same order as the lazy version; the difference is in the priority-queue operations. It builds the MST as follows:

- Adds 0 to the MST and all edges in its adjacency list to the priority queue, since each such edge is the best (only) known connection between a tree vertex and a non-tree vertex.
- Adds 7 and 0-7 to the MST, replaces 0-4 with 4-7 as the lightest edge from a tree vertex to 4, adds 1-7 and 5-7 to the priority queue. Edge 2-7 does not affect the priority queue because its weight is not less than the weight of the known connection from the MST to 2.
- Adds 1 and 1-7 to the MST and 1-3 to the priority queue.
- Adds 2 and 0-2 to the MST, replaces 6-0 with 2-6 as the lightest edge from a tree vertex to 6, and replaces 1-3 with 2-3 as the lightest edge from a tree vertex to 3.
- Adds 3 and 2-3 to the MST.
- Adds 5 and 5-7 to the MST and replaces 4-7 with 4-5 as the lightest edge from a tree vertex to 4.
- Adds 4 and 4-5 to the MST.
- Adds 6 and 6-2 to the MST.

After having added  $V-1$  edges, the MST is complete and the priority queue is empty.



Trace of Prim's algorithm (eager version)

**ALGORITHM 4.7** Prim's MST algorithm (eager version)

```

public class PrimMST
{
    private Edge[] edgeTo;           // shortest edge from tree vertex
    private double[] distTo;         // distTo[w] = edgeTo[w].weight()
    private boolean[] marked;        // true if v on tree
    private IndexMinPQ<Double> pq;  // eligible crossing edges

    public PrimMST(EdgeWeightedGraph G)
    {
        edgeTo = new Edge[G.V()];
        distTo = new double[G.V()];
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        pq = new IndexMinPQ<Double>(G.V());
        distTo[0] = 0.0;                // Initialize pq with 0, weight 0.
        pq.insert(0, 0.0);
        while (!pq.isEmpty())
            visit(G, pq.delMin());    // Add closest vertex to tree.
    }

    private void visit(EdgeWeightedGraph G, int v)
    { // Add v to tree; update data structures.
        marked[v] = true;
        for (Edge e : G.adj(v))
        {
            int w = e.other(v);
            if (marked[w]) continue;    // v-w is ineligible.
            if (e.weight() < distTo[w])
            { // Edge e is new best connection from tree to w.
                edgeTo[w] = e;
                distTo[w] = e.weight();
                if (pq.contains(w)) pq.changeKey(w, distTo[w]);
                else                  pq.insert(w, distTo[w]);
            }
        }
    }

    public Iterable<Edge> edges()      // See Exercise 4.3.21.
    public double weight()             // See Exercise 4.3.31.
}

```

This implementation of Prim's algorithm keeps eligible crossing edges on an index priority queue.

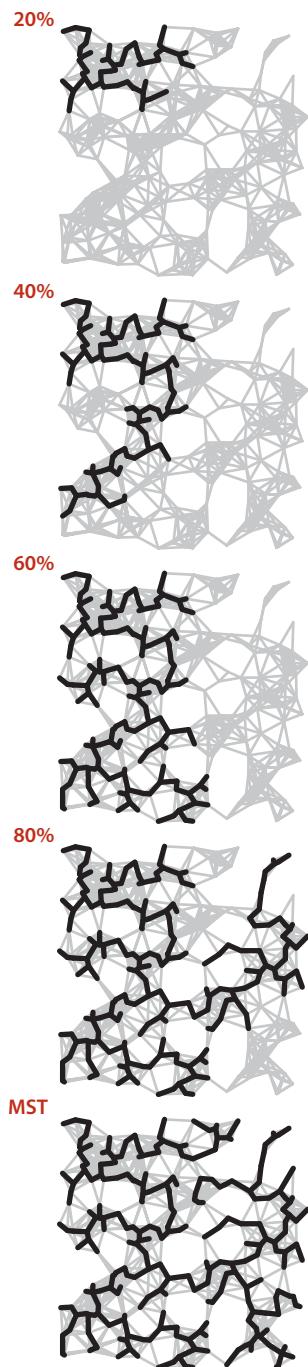
AN ESSENTIALLY IDENTICAL ARGUMENT as in the proof of PROPOSITION M proves that the eager version of Prim's algorithm finds the MST of a connected edge-weighted graph in time proportional to  $E \log V$  and extra space proportional to  $V$ .

**Proposition N.** The eager version of Prim's algorithm uses extra space proportional to  $V$  and time proportional to  $E \log V$  (in the worst case) to compute the MST of a connected edge-weighted graph with  $E$  edges and  $V$  vertices.

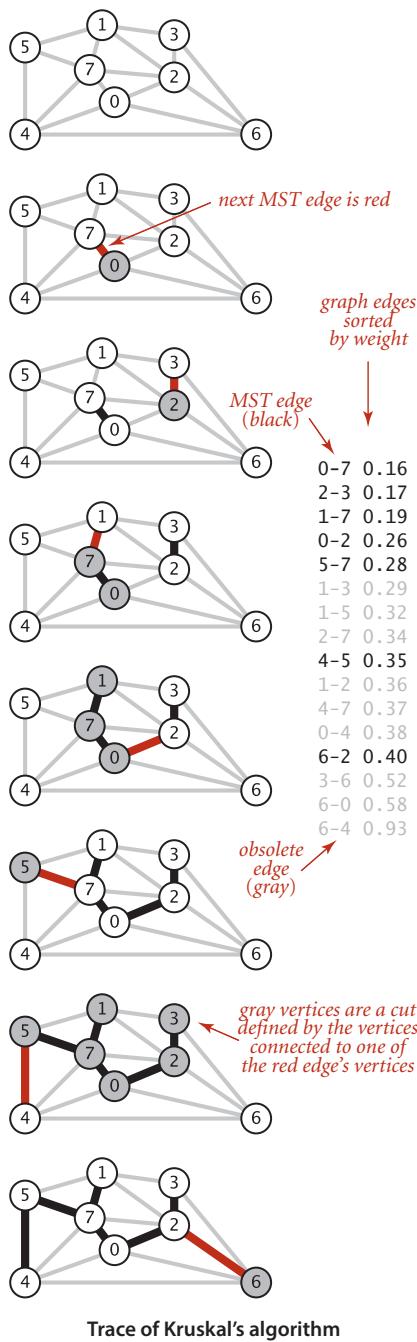
**Proof:** The number of vertices on the priority queue is at most  $V$ , and there are three vertex-indexed arrays, which implies the space bound. The algorithm uses  $V$  *insert* operations,  $V$  *delete the minimum* operations, and (in the worst case)  $E$  *change priority* operations. These counts, coupled with the fact that our heap-based implementation of the index priority queue implements all these operations in time proportional to  $\log V$  (see page 321), imply the time bound.

For the huge sparse graphs that are typical in practice, there is no asymptotic difference in the time bound (because  $\lg E \sim \lg V$  for sparse graphs); the space bound is a constant-factor (but significant) improvement. Further analysis and experimentation are best left for experts facing performance-critical applications, where many factors come into play, including the implementations of MinPQ and IndexMinPQ, the graph representation, properties of the application's graph model, and so forth. As usual, such improvements need to be carefully considered, as the increased code complexity is only justified for applications where constant-factor performance gains are important, and might even be counterproductive on complex modern systems.

The diagram at right shows Prim's algorithm in operation on our 250-vertex Euclidean graph `mediumEWG.txt`. It is a fascinating dynamic process (see also EXERCISE 4.3.27). Most often the tree grows by connecting a new vertex to the vertex just added. When reaching an area with no nearby non-tree vertices, the growth starts from another part of the tree.



Prim's algorithm (250 vertices)



**Kruskal's algorithm** The second MST algorithm that we consider in detail is to process the edges in order of their weight values (smallest to largest), taking for the MST (coloring black) each edge that does not form a cycle with edges previously added, stopping after adding  $V-1$  edges. The black edges form a forest of trees that evolves gradually into a single tree, the MST. This method is known as *Kruskal's algorithm*:

**Proposition O.** Kruskal's algorithm computes the MST of any connected edge-weighted graph.

**Proof:** Immediate from PROPOSITION K. If the next edge to be considered does not form a cycle with black edges, it crosses a cut defined by the set of vertices connected to one of the edge's vertices by black edges (and its complement). Since the edge does not create a cycle, it is the only crossing edge seen so far, and since we consider the edges in sorted order, it is a crossing edge of minimum weight. Thus, the algorithm is successively taking a minimal-weight crossing edge, in accordance with the greedy algorithm.

Prim's algorithm builds the MST one edge at a time, finding a new edge to attach to a single growing tree at each step. Kruskal's algorithm also builds the MST one edge at a time; but, by contrast, it finds an edge that connects two trees in a forest of growing trees. We start with a degenerate forest of  $V$  single-vertex trees and perform the operation of combining two trees (using the lightest edge possible) until there is just one tree left: the MST.

The figure at left shows a step-by-step example of the operation of Kruskal's algorithm on `tinyEWG.txt`. The five lowest-weight edges in the graph are taken for the MST, then 1-3, 1-5, and 2-7 are determined to be ineligible before 4-5 is taken for the MST, and finally 1-2, 4-7, and 0-4 are determined to be ineligible and 6-2 is taken for the MST.

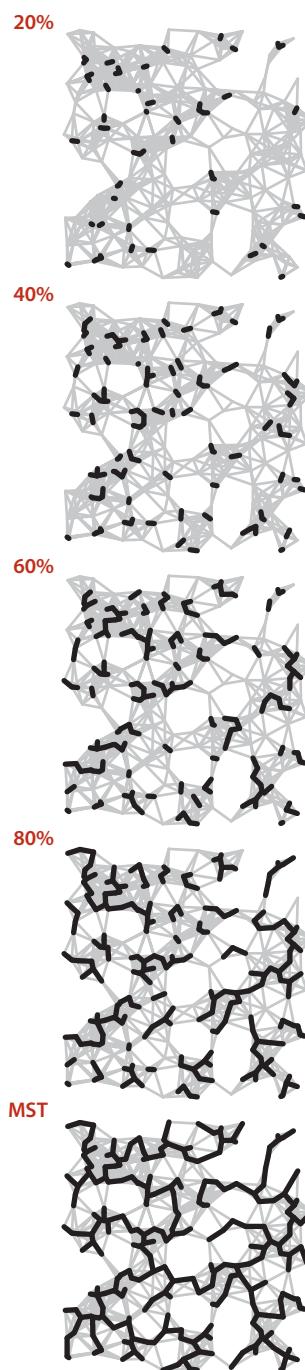
Kruskal's algorithm is also not difficult to implement, given the basic algorithmic tools that we have considered in this book: we use a priority queue (SECTION 2.4) to consider the edges in order by weight, a union-find data structure (SECTION 1.5) to identify those that cause cycles, and a queue (SECTION 1.3) to collect the MST edges. ALGORITHM 4.8 is an implementation along these lines. Note that collecting the MST edges in a Queue means that when a client iterates through the edges it gets them in increasing order of their weight. The `weight()` method requires iterating through the queue to add the edge weights (or keeping a running total in an instance variable) and is left as an exercise (see EXERCISE 4.3.31).

Analyzing the running time of Kruskal's algorithm is a simple matter because we know the running times of its basic operations.

**Proposition N (continued).** Kruskal's algorithm uses space proportional to  $E$  and time proportional to  $E \log E$  (in the worst case) to compute the MST of an edge-weighted connected graph with  $E$  edges and  $V$  vertices.

**Proof:** The implementation uses the priority-queue constructor that initializes the priority queue with all the edges, at a cost of at most  $E$  compares (see SECTION 2.4). After the priority queue is built, the argument is the same as for Prim's algorithm. The number of edges on the priority queue is at most  $E$ , which gives the space bound, and the cost per operation is at most  $2 \lg E$  compares, which gives the time bound. Kruskal's algorithm also performs up to  $E$  `connected()` and  $V$  `union()` operations, but that cost does not contribute to the  $E \log E$  order of growth of the total running time (see SECTION 1.5).

As with Prim's algorithm the cost bound is conservative, since the algorithm terminates after finding the  $V - 1$  MST edges. The order of growth of the actual cost is  $E + E_0 \log E$ , where  $E_0$  is the number of edges whose weight is less than the weight of the MST edge with the highest weight. Despite this advantage, Kruskal's algorithm is generally slower than Prim's algorithm because it has to do a `connected()` operation for each edge, in addition to the priority-queue operations that both algorithms do for each edge processed (see EXERCISE 4.3.39).



The figure at left illustrates the algorithm's dynamic characteristics on the larger example `mediumEWG.txt`. The fact that the edges are added to the forest in order of their length is quite apparent.

---

**ALGORITHM 4.8 Kruskal's MST algorithm**


---

```

public class KruskalMST
{
    private Queue<Edge> mst;
    public KruskalMST(EdgeWeightedGraph G)
    {
        mst = new Queue<Edge>();
        MinPQ<Edge> pq = new MinPQ<Edge>();
        for (Edge e : G.edges())
            pq.insert(e);
        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
            Edge e = pq.delMin();           // Get min weight edge on pq
            int v = e.either(), w = e.other(v); // and its vertices.
            if (uf.connected(v, w)) continue; // Ignore ineligible edges.
            uf.union(v, w);                // Merge components.
            mst.enqueue(e);                // Add edge to mst.
        }
    }
    public Iterable<Edge> edges()
    { return mst; }
    public double weight()           // See Exercise 4.3.31.
}
```

---

This implementation of Kruskal's algorithm uses a queue to hold MST edges, a priority queue to hold edges not yet examined, and a union-find data structure for identifying ineligible edges. The MST edges are returned to the client in increasing order of their weights. The `weight()` method is left as an exercise.

```
% java KruskalMST tinyEWG.txt
0-7 0.16000
2-3 0.17000
1-7 0.19000
0-2 0.26000
5-7 0.28000
4-5 0.35000
6-2 0.40000
1.8100
```

**Perspective** The MST problem is one of the most heavily studied problems that we encounter in this book. Basic approaches to solving it were invented long before the development of modern data structures and modern techniques for analyzing the performance of algorithms, at a time when finding the MST of a graph that contained, say, thousands of edges was a daunting task. The MST algorithms that we have considered differ from these old ones essentially in their use and implementation of modern algorithms and data structures for basic tasks, which (coupled with modern computing power) makes it possible for us to compute MSTs with millions or even billions of edges.

**Historical notes.** An MST implementation for dense graphs (see EXERCISE 4.3.29) was first presented by R. Prim in 1961 and, independently, by E. W. Dijkstra soon thereafter. It is usually referred to as *Prim's algorithm*, although Dijkstra's presentation was more general. But the basic idea was also presented by V. Jarník in 1939, so some authors refer to the method as *Jarník's algorithm*, thus characterizing Prim's (or Dijkstra's) role as finding an efficient implementation of the algorithm for dense graphs. As the priority-queue ADT came into use in the early 1970s, its application to finding MSTs of sparse

graphs was straightforward; the fact that MSTs of sparse graphs could be computed in time proportional to  $E \log E$  became widely known without attribution to any particular researcher. In 1984, M. L. Fredman and R. E. Tarjan developed the *Fibonacci heap* data structure, which improves the theoretical bound on the order of growth of the running time of Prim's algorithm to  $E + V \log V$ . J. Kruskal presented his algorithm in 1956, but, again, the relevant ADT implementations were not carefully studied for many years. Other interesting historical notes are that Kruskal's paper mentioned a version of Prim's algorithm and that a 1926 (!) paper by O. Boruvka mentioned both approaches. Boruvka's paper addressed a power-distribution application and introduced yet another method that is easily implemented with modern data structures (see EXERCISE 4.3.43 and EXERCISE 4.3.44). The method was rediscovered by M. Sollin in 1961;

algorithm	worst-case order of growth for $V$ vertices and $E$ edges	
	space	time
<i>lazy Prim</i>	$E$	$E \log E$
<i>eager Prim</i>	$V$	$E \log V$
<i>Kruskal</i>	$E$	$E \log E$
<i>Fredman-Tarjan</i>	$V$	$E + V \log V$
<i>Chazelle</i>	$V$	<i>very, very nearly, but not quite <math>E</math></i>
<i>impossible?</i>	$V$	$E?$

**Performance characteristics of MST algorithms**

it later attracted attention as the basis for MST algorithms with efficient asymptotic performance and as the basis for parallel MST algorithms.

**A linear-time algorithm?** On the one hand, no theoretical results have been developed that deny the existence of an MST algorithm that is guaranteed to run in linear time for all graphs. On the other hand, the goal of developing algorithms for computing the MST of sparse graphs in linear time remains elusive. Since the 1970s the applicability of the union-find abstraction to Kruskal's algorithm and the applicability of the priority-queue abstraction to Prim's algorithm have been prime motivations for many researchers to seek better implementations of those ADTs. Many researchers have concentrated on finding efficient priority-queue implementations as the key to finding efficient MST algorithms for sparse graphs; many other researchers have studied variations of Boruvka's algorithm as the basis for nearly linear-time MST algorithms for sparse graphs. Such research still holds the potential to lead us eventually to a practical linear-time MST algorithm and has even shown the existence of a randomized linear-time algorithm. Also, researchers are getting quite close to the linear-time goal: B. Chazelle exhibited an algorithm in 1997 that certainly could never be distinguished from a linear-time algorithm in any conceivable practical situation (even though it is provably nonlinear), but is so complicated that no one would use it in practice. While the algorithms that have emerged from such research are generally quite complicated, simplified versions of some of them may yet be shown to be useful in practice. In the meantime, we can use the basic algorithms that we have considered here to compute the MST in linear time in most practical situations, perhaps paying an extra factor of  $\log V$  for some sparse graphs.

IN SUMMARY, we can consider the MST problem to be “solved” for practical purposes. For most graphs, the cost of finding the MST is only slightly higher than the cost of extracting the graph's edges. This rule holds except for huge graphs that are extremely sparse, but the available performance improvement over the best-known algorithms even in this case is a small constant factor, perhaps a factor of 10 at best. These conclusions are borne out for many graph models, and practitioners have been using Prim's and Kruskal's algorithms to find MSTs in huge graphs for decades.

**Q&A**

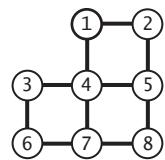
**Q.** Do Prim's and Kruskal's algorithms work for *directed* graphs?

**A.** No, not at all. That is a more difficult graph-processing problem known as the *minimum cost arborescence* problem.

**EXERCISES**

**4.3.1** Prove that you can rescale the weights by adding a positive constant to all of them or by multiplying them all by a positive constant without affecting the MST.

**4.3.2** Draw all of the MSTs of the graph depicted at right (all edge weights are equal).



**4.3.3** Show that if a graph's edges all have distinct weights, the MST is unique.

**4.3.4** Consider the assertion that an edge-weighted graph has a unique MST *only* if its edge weights are distinct. Give a proof or a counterexample.

**4.3.5** Show that the greedy algorithm is valid even when edge weights are not distinct.

**4.3.6** Give the MST of the weighted graph obtained by deleting vertex 7 from `tinyEWG.txt` (see page 604).

**4.3.7** How would you find a *maximum* spanning tree of an edge-weighted graph?

**4.3.8** Prove the following, known as the *cycle property*: Given any cycle in an edge-weighted graph (all edge weights distinct), the edge of maximum weight in the cycle does not belong to the MST of the graph.

**4.3.9** Implement the constructor for `EdgeWeightedGraph` that reads an edge-weighted graph from the input stream, by suitably modifying the constructor from `Graph` (see page 526).

**4.3.10** Develop an `EdgeWeightedGraph` implementation for dense graphs that uses an adjacency-matrix (two-dimensional array of weights) representation. Disallow parallel edges.

**4.3.11** Determine the amount of memory used by `EdgeWeightedGraph` to represent a graph with  $V$  vertices and  $E$  edges, using the memory-cost model of SECTION 1.4.

**4.3.12** Suppose that a graph has distinct edge weights. Does its lightest edge have to belong to the MST? Can its heaviest edge belong to the MST? Does a min-weight edge on every cycle have to belong to the MST? Prove your answer to each question or give a counterexample.

**4.3.13** Give a counterexample that shows why the following strategy does not necessarily find the MST: 'Start with any vertex as a single-vertex MST, then add  $V-1$  edges

**EXERCISES (continued)**

to it, always taking next a min-weight edge incident to the vertex most recently added to the MST.

**4.3.14** Given an MST for an edge-weighted graph  $G$ , suppose that an edge in  $G$  that does not disconnect  $G$  is deleted. Describe how to find an MST of the new graph in time proportional to  $E$ .

**4.3.15** Given an MST for an edge-weighted graph  $G$  and a new edge  $e$  with weight  $w$ , describe how to find an MST of the new graph in time proportional to  $V$ .

**4.3.16** Given an MST for an edge-weighted graph  $G$  and a new edge  $e$ , write a program that determines the range of weights for which  $e$  is in an MST.

**4.3.17** Implement `toString()` for `EdgeWeightedGraph`.

**4.3.18** Give traces that show the process of computing the MST of the graph defined in EXERCISE 4.3.6 with the lazy version of Prim's algorithm, the eager version of Prim's algorithm, and Kruskal's algorithm.

**4.3.19** Suppose that you implement `PrimMST` but instead of using a priority queue to find the next vertex to add to the tree, you scan through all  $V$  entries in the `distTo[]` array to find the non-tree vertex with the smallest weight. What would be the order of growth of the running time for graphs with  $V$  vertices and  $E$  edges? When would this method be appropriate, if ever? Defend your answer.

**4.3.20** True or false: At any point during the execution of Kruskal's algorithm, each vertex is closer to some vertex in its subtree than to any vertex not in its subtree. Prove your answer.

**4.3.21** Provide an implementation of `edges()` for `PrimMST` (page 622).

*Solution:*

```
public Iterable<Edge> edges()
{
    Queue<Edge> mst = new Queue<Edge>();
    for (int v = 1; v < edgeTo.length; v++)
        mst.enqueue(edgeTo[v]);
    return mst;
}
```

**CREATIVE PROBLEMS**

**4.3.22** *Minimum spanning forest.* Develop versions of Prim’s and Kruskal’s algorithms that compute the minimum spanning *forest* of an edge-weighted graph that is not necessarily connected. Use the connected-components API of SECTION 4.1 and find MSTs in each component.

**4.3.23** *Vyssotsky’s algorithm.* Develop an implementation that computes the MST by applying the cycle property (see EXERCISE 4.3.8) repeatedly: Add edges one at a time to a putative tree, deleting a maximum-weight edge on the cycle if one is formed. *Note:* This method has received less attention than the others that we consider because of the comparative difficulty of maintaining a data structure that supports efficient implementation of the “delete the maximum-weight edge on the cycle” operation.

**4.3.24** *Reverse-delete algorithm.* Develop an implementation that computes the MST as follows: Start with a graph containing all of the edges. Then repeatedly go through the edges in decreasing order of weight. For each edge, check if deleting that edge will disconnect the graph; if not, delete it. Prove that this algorithm computes the MST. What is the order of growth of the number of edge-weight compares performed by your implementation?

**4.3.25** *Worst-case generator.* Develop a reasonable generator for edge-weighted graphs with  $V$  vertices and  $E$  edges such that the running time of the lazy version of Prim’s algorithm is nonlinear. Answer the same question for the eager version.

**4.3.26** *Critical edges.* An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. Show how to find all critical edges in a graph in time proportional to  $E \log E$ . *Note:* This question assumes that edge weights are not necessarily distinct (otherwise all edges in the MST are critical).

**4.3.27** *Animations.* Write a client program that does dynamic graphical animations of MST algorithms. Run your program for `mediumEWG.txt` to produce images like the figures on page 621 and page 624.

**4.3.28** *Space-efficient data structures.* Develop an implementation of the lazy version of Prim’s algorithm that saves space by using lower-level data structures for `EdgeWeightedGraph` and for `MinPQ` instead of `Bag` and `Edge`. Estimate the amount of memory saved as a function of  $V$  and  $E$ , using the memory-cost model of SECTION 1.4 (see EXERCISE 4.3.11).

**CREATIVE PROBLEMS (continued)**

**4.3.29 Dense graphs.** Develop an implementation of Prim's algorithm that uses an *eager* approach (but not a priority queue) and computes the MST using  $V^2$  edge-weight comparisons.

**4.3.30 Euclidean weighted graphs.** Modify your solution to EXERCISE 4.1.37 to create an API `EuclideanEdgeWeightedGraph` for graphs whose vertices are points in the plane, so that you can work with graphical representations.

**4.3.31 MST weights.** Develop implementations of `weight()` for `LazyPrimMST`, `PrimMST`, and `KruskalMST`, using a *lazy* strategy that iterates through the MST edges when the client calls `weight()`. Then develop alternate implementations that use an *eager* strategy that maintains a running total as the MST is computed.

**4.3.32 Specified set.** Given a connected edge-weighted graph  $G$  and a specified set of edges  $S$  (having no cycles), describe a way to find a minimum-weight spanning tree of  $G$  among those spanning trees that contain all the edges in  $S$ .

**4.3.33 Certification.** Write an `MST` and `EdgeWeightedGraph` client `check()` that uses the following *cut optimality conditions* implied by PROPOSITION J to verify that a proposed set of edges is in fact an MST: A set of edges is an MST if it is a spanning tree and every edge is a minimum-weight edge in the cut defined by removing that edge from the tree. What is the order of growth of the running time of your method?

## EXPERIMENTS

**4.3.34 Random sparse edge-weighted graphs.** Write a random-sparse-edge-weighted-graph generator based on your solution to EXERCISE 4.1.41. To assign edge weights, define a random-edge-weighted graph ADT and write two implementations: one that generates uniformly distributed weights, another that generates weights according to a Gaussian distribution. Write client programs to generate sparse random edge-weighted graphs for both weight distributions with a well-chosen set of values of  $V$  and  $E$  so that you can use them to run empirical tests on graphs drawn from various distributions of edge weights.

**4.3.35 Random Euclidean edge-weighted graphs.** Modify your solution to EXERCISE 4.1.42 to assign the distance between vertices as each edge's weight.

**4.3.36 Random grid edge-weighted graphs.** Modify your solution to EXERCISE 4.1.43 to assign a random weight (between 0 and 1) to each edge.

**4.3.37 Real edge-weighted graphs.** Find a large weighted graph somewhere online—perhaps a map with distances, telephone connections with costs, or an airline rate schedule. Write a program `RandomRealEdgeWeightedGraph` that builds a weighted graph by choosing  $V$  vertices at random and  $E$  weighted edges at random from the subgraph induced by those vertices.

*Testing all algorithms and studying all parameters against all graph models is unrealistic. For each problem listed below, write a client that addresses the problem for any given input graph, then choose among the generators above to run experiments for that graph model. Use your judgment in selecting experiments, perhaps in response to results of previous experiments. Write a narrative explaining your results and any conclusions that might be drawn.*

**4.3.38 Cost of laziness.** Run empirical studies to compare the performance of the lazy version of Prim's algorithm with the eager version, for various types of graphs.

**4.3.39 Prim versus Kruskal.** Run empirical studies to compare the performance of the lazy and eager versions of Prim's algorithm with Kruskal's algorithm.

**4.3.40 Reduced overhead.** Run empirical studies to determine the effect of using primitive types instead of `Edge` values in `EdgeWeightedGraph`, as described in EXERCISE 4.3.28.

**EXPERIMENTS (continued)**

**4.3.41 Heaviest MST edge.** Run empirical studies to analyze the weight of the heaviest edge in the MST and the number of graph edges that are not heavier than that one.

**4.3.42 Partitioning.** Develop an implementation based on integrating Kruskal's algorithm with quicksort partitioning (instead of using a priority queue) so as to check MST membership of each edge as soon as all smaller edges have been checked.

**4.3.43 Boruvka's algorithm.** Develop an implementation of Boruvka's algorithm: Build an MST by adding edges to a growing forest of trees, as in Kruskal's algorithm, but in stages. At each stage, find the minimum-weight edge that connects each tree to a different one, then add all such edges to the MST. Assume that the edge weights are all different, to avoid cycles. *Hint:* Maintain a vertex-indexed array to identify the edge that connects each tree to its nearest neighbor, and use the union-find data structure.

**4.3.44 Improved Boruvka.** Develop an implementation of Boruvka's algorithm that uses doubly-linked circular lists to represent MST subtrees so that subtrees can be merged and renamed in time bounded by  $E$  during each stage (and the union-find data type is therefore not needed).

**4.3.45 External MST.** Describe how you would find the MST of a graph so large that only  $V$  edges can fit into main memory at once.

**4.3.46 Johnson's algorithm.** Develop a priority-queue implementation that uses a  $d$ -way heap (see EXERCISE 2.4.41). Find the best value of  $d$  for various weighted graph models.

*This page intentionally left blank*

## 4.4 SHORTEST PATHS

PERHAPS THE MOST INTUITIVE graph-processing problem is one that you encounter regularly, when using a map application or a navigation system to get directions from one place to another. A graph model is immediate: vertices correspond to intersections and edges correspond to roads, with weights on the edges that model the cost, perhaps distance or travel time. The possibility of one-way roads means that we will need to consider edge-weighted *digraphs*. In this model, the problem is easy to formulate:

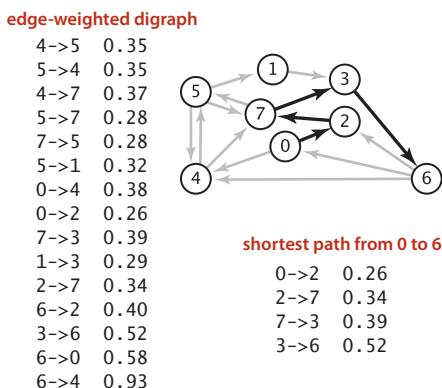
*Find a lowest-cost way to get from one vertex to another.*

Beyond direct applications of this sort, the shortest-paths model is appropriate for a range of other problems, some of which do not seem to be at all related to graph processing. As one example, we shall consider the *arbitrage* problem from computational finance at the end of this section.

We adopt a general model where we work with *edge-weighted digraphs* (combining the models of SECTION 4.2 and SECTION 4.3). In SECTION 4.2 we wished to know whether it is *possible* to get from one vertex to another; in this section, we take weights into consideration, as we did for undirected edge-weighted graphs in SECTION 4.3. Every directed path in

application	vertex	edge
<i>map</i>	intersection	road
<i>network</i>	router	connection
<i>schedule</i>	job	precedence constraint
<i>arbitrage</i>	currency	exchange rate

Typical shortest-paths applications



an edge-weighted digraph has an associated *path weight*, the value of which is the sum of the weights of that path's edges. This essential measure allows us to formulate such problems as “find the lowest-weight directed path from one vertex to another,” the topic of this section. The figure at left shows an example.

**Definition.** A *shortest path* from vertex  $s$  to vertex  $t$  in an edge-weighted digraph is a directed path from  $s$  to  $t$  with the property that no other such path has a lower weight.

An edge-weighted digraph and a shortest path

Thus, in this section, we consider classic algorithms for the following problem:

**Single-source shortest paths.** Given an edge-weighted digraph and a source vertex  $s$ , support queries of the form *Is there a directed path from  $s$  to a given target vertex  $t$ ?* If so, find a *shortest* such path (one whose total weight is minimal).

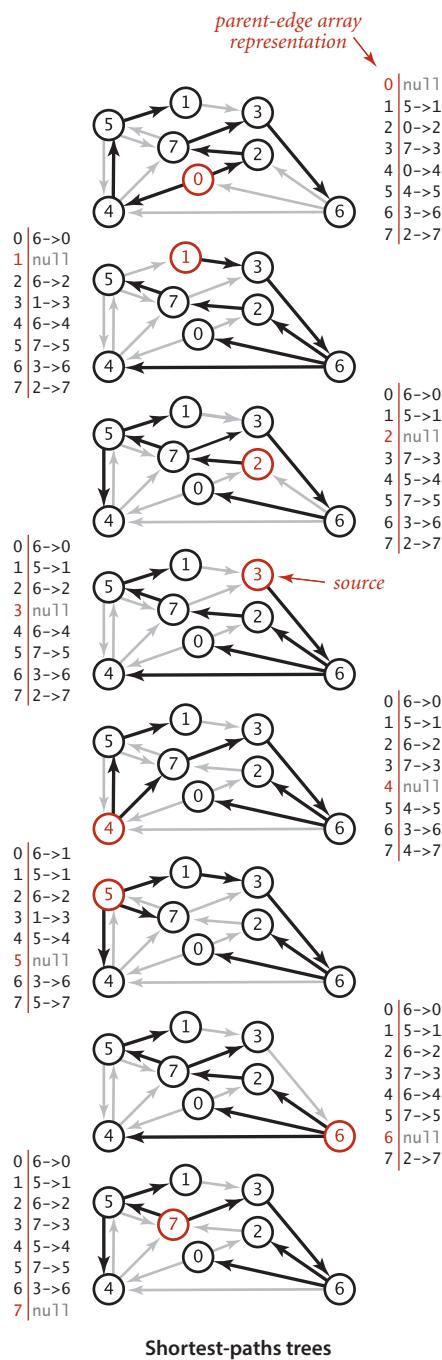
The plan of the section is to cover the following list of topics:

- Our APIs and implementations for edge-weighted digraphs, and a single-source shortest-paths API
- The classic Dijkstra's algorithm for the problem when weights are nonnegative
- A faster algorithm for acyclic edge-weighted digraphs (edge-weighted DAGs) that works even when edge weights can be negative
- The classic Bellman-Ford algorithm for use in the general case, when cycles may be present, edge weights may be negative, and we need algorithms for finding negative-weight cycles and shortest paths in edge-weighted digraphs with no such cycles

In the context of the algorithms, we also consider applications.

**Properties of shortest paths** The basic definition of the shortest-paths problem is succinct, but its brevity masks several points worth examining before we begin to formulate algorithms and data structures for solving it:

- *Paths are directed.* A shortest path must respect the direction of its edges.
- *The weights are not necessarily distances.* Geometric intuition can be helpful in understanding algorithms, so we use examples where vertices are points in the plane and weights are Euclidean distances, such as the digraph on the facing page. But the weights might represent time or cost or an entirely different variable and do not need to be proportional to a distance at all. We are emphasizing this point by using mixed-metaphor terminology where we refer to a *shortest* path of minimal *weight* or *cost*.
- *Not all vertices need be reachable.* If  $t$  is not reachable from  $s$ , there is no path at all, and therefore there is no shortest path from  $s$  to  $t$ . For simplicity, our small running example is strongly connected (every vertex is reachable from every other vertex).
- *Negative weights introduce complications.* For the moment, we assume that edge weights are positive (or zero). The surprising impact of negative weights is a major focus of the last part of this section.
- *Shortest paths are normally simple.* Our algorithms ignore zero-weight edges that form cycles, so that the shortest paths they find have no cycles.
- *Shortest paths are not necessarily unique.* There may be multiple paths of the low-



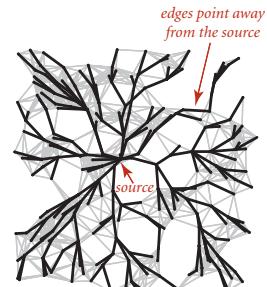
shortest weight from one vertex to another; we are content to find any one of them.

■ *Parallel edges and self-loops may be present.* Only the lowest-weight among a set of parallel edges will play a role, and no shortest path contains a self-loop (except possibly one of zero weight, which we ignore). In the text, we implicitly assume that parallel edges are not present for convenience in using the notation  $v \rightarrow w$  to refer unambiguously to the edge from  $v$  to  $w$ , but our code handles them without difficulty.

**Shortest-paths tree** We focus on the *single-source shortest-paths problem*, where we are given a source vertex  $s$ . The result of the computation is a tree known as the *shortest-paths tree* (SPT), which gives a shortest path from  $s$  to every vertex reachable from  $s$ .

**Definition.** Given an edge-weighted digraph and a designated source vertex  $s$ , a *shortest-paths tree* for vertex  $s$  is a subgraph containing  $s$  and all the vertices reachable from  $s$  that forms a directed tree rooted at  $s$  such that every tree path is a shortest path in the digraph.

Such a tree always exists: in general there may be two paths of the same length connecting  $s$  to a vertex; if that is the case, we can delete the final edge on one of them, continuing until we have only one path connecting the source to each vertex (a rooted tree). By building a shortest-paths tree, we can provide clients with the shortest path from  $s$  to any vertex in the graph, using a parent-link representation, in precisely the same manner as for paths in graphs in SECTION 4.1.



**Edge-weighted digraph data types** Our data type for directed edges is simpler than for undirected edges because we follow directed edges in just one direction. Instead of the `either()` and `other()` methods in `Edge`, we have `from()` and `to()` methods:

---

<code>public class DirectedEdge</code>	
	<code>DirectedEdge(int v, int w, double weight)</code>
	<code>double weight()</code>
	<i>weight of this edge</i>
	<code>int from()</code>
	<i>vertex this edge points from</i>
	<code>int to()</code>
	<i>vertex this edge points to</i>
	<code>String toString()</code>
	<i>string representation</i>
	<b>Weighted directed-edge API</b>

---

As with our transition from `Graph` to `EdgeWeightedGraph` from SECTION 4.1 to SECTION 4.3, we include an `edges()` method and use `DirectedEdge` instead of integers:

---

<code>public class EdgeWeightedDigraph</code>	
	<code>EdgeWeightedDigraph(int V)</code> <i>empty V-vertex digraph</i>
	<code>EdgeWeightedDigraph(In in)</code> <i>construct from in</i>
	<code>int V()</code> <i>number of vertices</i>
	<code>int E()</code> <i>number of edges</i>
	<code>void addEdge(DirectedEdge e)</code> <i>add e to this digraph</i>
<code>Iterable&lt;DirectedEdge&gt;</code>	<code>adj(int v)</code> <i>edges pointing from v</i>
<code>Iterable&lt;DirectedEdge&gt;</code>	<code>edges()</code> <i>all edges in this digraph</i>
	<code>String toString()</code> <i>string representation</i>
	<b>Edge-weighted digraph API</b>

---

You can find implementations of these two APIs on the following two pages. These are natural extensions of the implementations of SECTION 4.2 and SECTION 4.3. Instead of the adjacency lists of integers used in `Digraph`, we have adjacency lists of `DirectedEdge` objects in `EdgeWeightedDigraph`. As with the transition from `Graph` to `Digraph` from SECTION 4.1 to SECTION 4.2, the transition from `EdgeWeightedGraph` in SECTION 4.3 to `EdgeWeightedDigraph` in this section simplifies the code, since each edge appears only once in the data structure.

## Directed weighted edge data type

---

```
public class DirectedEdge
{
    private final int v;                      // edge tail
    private final int w;                      // edge head
    private final double weight;              // edge weight

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public double weight()
    {   return weight;   }

    public int from()
    {   return v;   }

    public int to()
    {   return w;   }

    public String toString()
    {   return String.format("%d->%d %.2f", v, w, weight);   }

}
```

---

This `DirectedEdge` implementation is simpler than the undirected weighted `Edge` implementation of SECTION 4.3 (see page 610) because the two vertices are distinguished. Our clients use the idiomatic code `int v = e.to(), w = e.from();` to access a `DirectedEdge` `e`'s two vertices.

## Edge-weighted digraph data type

```
public class EdgeWeightedDigraph
{
    private final int V;                      // number of vertices
    private int E;                            // number of edges
    private Bag<DirectedEdge>[] adj;        // adjacency lists

    public EdgeWeightedDigraph(int V)
    {
        this.V = V;
        this.E = 0;
        adj = (Bag<DirectedEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<DirectedEdge>();
    }

    public EdgeWeightedDigraph(In in)
    // See Exercise 4.4.2.

    public int V() { return V; }
    public int E() { return E; }

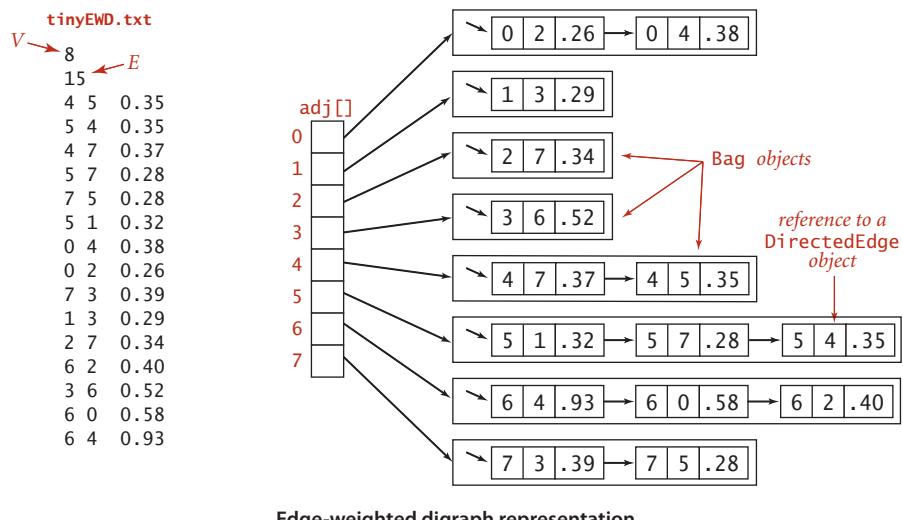
    public void addEdge(DirectedEdge e)
    {
        adj[e.from()].add(e);
        E++;
    }

    public Iterable<DirectedEdge> adj(int v)
    { return adj[v]; }

    public Iterable<DirectedEdge> edges()
    {
        Bag<DirectedEdge> bag = new Bag<DirectedEdge>();
        for (int v = 0; v < V; v++)
            for (DirectedEdge e : adj[v])
                bag.add(e);
        return bag;
    }

}
```

This `EdgeWeightedDigraph` implementation is an amalgam of `EdgeWeightedGraph` and `Digraph` that maintains a vertex-indexed array of bags of `DirectedEdge` objects. As with `Digraph`, every edge appears just once: if an edge connects  $v$  to  $w$ , it appears in  $v$ 's adjacency list. Self-loops and parallel edges are allowed. The `toString()` implementation is left as EXERCISE 4.4.2.



The figure above shows the data structure that `EdgeWeightedDigraph` builds to represent the digraph defined by the edges at left when they are added in the order they appear. As usual, we use `Bag` to represent adjacency lists and depict them as linked lists, the standard representation. As with the unweighted digraphs of SECTION 4.2, only one representation of each edge appears in the data structure.

**Shortest-paths API.** For shortest paths, we use the same design paradigm as for the `DepthFirstPaths` and `BreadthFirstPaths` APIs in SECTION 4.1. Our algorithms implement the following API to provide clients with shortest paths and their lengths:

---

```
public class SP
    SP(EdgeWeightedDigraph G, int s)   constructor
        double distTo(int v)           distance from s
                                        to v,  $\infty$  if no path
        boolean hasPathTo(int v)       path from s to v?
        Iterable<DirectedEdge> pathTo(int v)   path from s to v,
                                                null if none
```

**API for shortest-paths implementations**

The constructor builds a shortest-paths tree and computes shortest-paths distances; the client query methods use these data structures to provide distances and iterable paths to the client.

**Test client.** A sample client is shown below. It takes an input stream and source vertex index as command-line arguments, reads the edge-weighted digraph from the input stream, computes the SPT of that digraph for the source, and prints the shortest path from the source to each of the other vertices. We assume that all of our shortest-paths implementations include this test client. Our examples use the file `tinyEWD.txt` shown on the facing page, which defines the edges and weights that are used in the small sample digraph that we use for detailed traces of shortest-paths algorithms. It uses the same file format that we used for MST algorithms: the number of vertices  $V$  and the number of edges  $E$  followed by  $E$  lines, each with two vertex indices and a weight. You can also find on the booksite files that define several larger edge-weighted digraphs,

including the file `mediumEWD.txt` which defines the 250-vertex graph drawn on page 640. In the drawing of the graph, every line represents edges in both directions, so this file has twice as many lines as the corresponding file `mediumEWG.txt` that we examined for MSTs. In the drawing of the SPT, each line represents a directed edge pointing away from the source.

```
public static void main(String[] args)
{
    EdgeWeightedDigraph G;
    G = new EdgeWeightedDigraph(new In(args[0]));
    int s = Integer.parseInt(args[1]);
    SP sp = new SP(G, s);

    for (int t = 0; t < G.V(); t++)
    {
        StdOut.print(s + " to " + t);
        StdOut.printf(" (%4.2f): ", sp.distTo(t));
        if (sp.hasPathTo(t))
            for (DirectedEdge e : sp.pathTo(t))
                StdOut.print(e + " ");
        StdOut.println();
    }
}
```

Shortest paths test client

```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38 4->5 0.35 5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26 2->7 0.34 7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38 4->5 0.35
0 to 6 (1.51): 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52
0 to 7 (0.60): 0->2 0.26 2->7 0.34
```

**Data structures for shortest paths.** The data structures that we need to represent shortest paths are straightforward:

- *Edges on the shortest-paths tree:* As for DFS, BFS, and Prim's algorithm, we use a parent-edge representation in the form of a vertex-indexed array `edgeTo[]` of `DirectedEdge` objects, where `edgeTo[v]` is the edge that connects `v` to its parent in the tree (the last edge on a shortest path from `s` to `v`).
- *Distance to the source:* We use a vertex-indexed array `distTo[]` such that `distTo[v]` is the length of the shortest known path from `s` to `v`.

By convention, `edgeTo[s]` is `null` and `distTo[s]` is 0. We also adopt the convention that distances to vertices that are not reachable from the source are all `Double.POSITIVE_INFINITY`. As usual, we will develop data types that build these data structures in the constructor and then support instance methods that use them to support client queries for shortest paths and shortest-path distances.

**Edge relaxation.** Our shortest-paths implementations are based on a simple operation known as *relaxation*. We start knowing only the graph's edges and weights, with the `distTo[]` entry for the source initialized to 0 and all of the other `distTo[]` entries initialized to `Double.POSITIVE_INFINITY`. As an algorithm proceeds, it gathers information about the shortest paths that connect the source to each vertex encountered in our `edgeTo[]` and `distTo[]` data structures. By updating this information when we encounter edges, we can make new inferences about shortest paths. Specifically, we use *edge relaxation*, defined as follows: to *relax* an edge `v->w` means to test whether the best known way from `s` to `w` is to go from `s` to `v`, then take the edge from `v` to `w`, and, if so, update our data structures to indicate that to be the case. The code at the right implements this operation. The best known distance to `w` through `v` is the sum of `distTo[v]` and `e.weight()`—if that value is not smaller than `distTo[w]`, we say the edge is *ineligible*, and we ignore it; if it is smaller, we update the data

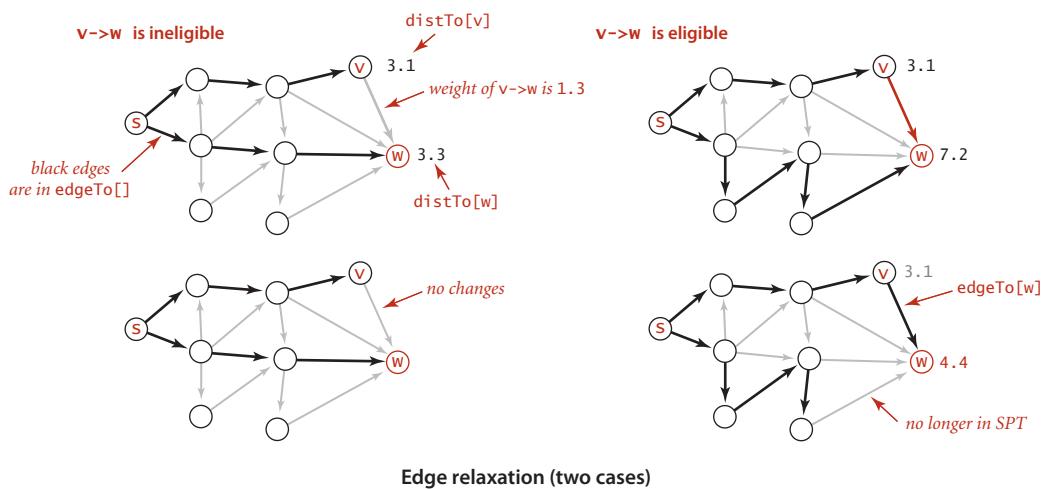
	edgeTo[]	distTo[]
0	null	0
1	5->1	0.32
2	0->2	0.26
3	7->3	0.39
4	0->4	0.38
5	4->5	0.35
6	3->6	0.52
7	2->7	0.34

Shortest-paths data structures

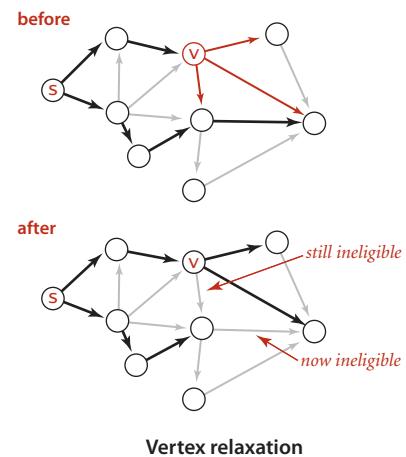
```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Edge relaxation

structures. The figure at the bottom of this page illustrates the two possible outcomes of an edge-relaxation operation. Either the edge is ineligible (as in the example at left) and no changes are made, or the edge  $v \rightarrow w$  leads to a shorter path to  $w$  (as in the example at right) and we update `edgeTo[w]` and `distTo[w]` (which might render some other edges ineligible and might create some new eligible edges). The term *relaxation* follows from the idea of a rubber band stretched tight on a path connecting two vertices: relaxing an edge is akin to relaxing the tension on the rubber band along a shorter path, if possible. We say that an edge  $e$  can be *successfully relaxed* if `relax(e)` would change the values of `distTo[e.to()]` and `edgeTo[e.to()]`.



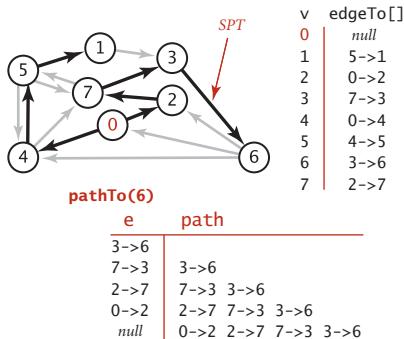
**Vertex relaxation.** All of our implementations actually relax *all* the edges pointing from a given vertex as shown in the (overloaded) implementation of `relax()` below. Note that any edge  $v \rightarrow w$  from a vertex whose `distTo[v]` entry is finite to a vertex whose `distTo[w]` entry is infinite is eligible and will be added to `edgeTo[w]` if relaxed. In particular, some edge leaving the source is the first to be added to `edgeTo[]`. Our algorithms choose vertices judiciously, so that each vertex relaxation finds a shorter path than the best known so far to some vertex, incrementally progressing toward the goal of finding shortest paths to every vertex.



```
private void relax(EdgeWeightedDigraph G, int v)
{
    for (DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

Vertex relaxation

**Client query methods.** In a manner similar to our implementations for pathfinding APIs in SECTION 4.1 (and EXERCISE 4.1.13), the `edgeTo[]` and `distTo[]` data structures directly support the `pathTo()`, `hasPathTo()`, and `distTo()` client query methods, as shown below. This code is included in all of our shortest-paths implementations. As we have noted already, `distTo[v]` is only meaningful when  $v$  is reachable from  $s$  and we adopt the convention that `distTo()` should return infinity for vertices that are not reachable from  $s$ . To implement this convention, we initialize all `distTo[]` entries to `Double.POSITIVE_INFINITY` and `distTo[s]` to 0; then our shortest-paths implementations will set `distTo[v]` to a finite value for all vertices  $v$  that are reachable from the source. Thus, we can dispense with the `marked[]` array that we normally use to mark reachable vertices in a graph search and implement `hasPathTo(v)` by testing whether `distTo[v]` equals `Double.POSITIVE_INFINITY`. For `pathTo()`, we use the convention that `pathTo(v)` returns `null` if  $v$  is not reachable from the source and a path with no edges if  $v$  is the source. For reachable vertices, we travel up the tree, pushing the edges that we find on a stack, in the same manner as we did for `DepthFirstPaths` and `BreadthFirstPaths`. The figure at right shows the discovery of the path  $0 \rightarrow 2 \rightarrow 7 \rightarrow 3 \rightarrow 6$  for our example.



```

public double distTo(int v)
{   return distTo[v];   }

public boolean hasPathTo(int v)
{   return distTo[v] < Double.POSITIVE_INFINITY;   }

public Iterable<DirectedEdge> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}

```

Client query methods for shortest paths

**Theoretical basis for shortest-paths algorithms** Edge relaxation is an easy-to-implement fundamental operation that provides a practical basis for our shortest-paths implementations. It also provides a theoretical basis for understanding the algorithms and an opportunity for us to do our algorithm correctness proofs at the outset.

**Optimality conditions.** The following proposition shows an equivalence between the *global* condition that the distances are shortest-paths distances, and the *local* condition that we test to relax an edge.

**Proposition P. (Shortest-paths optimality conditions)** Let  $G$  be an edge-weighted digraph, with  $s$  a source vertex in  $G$  and  $\text{distTo}[]$  a vertex-indexed array of path lengths in  $G$  such that, for all  $v$  reachable from  $s$ , the value of  $\text{distTo}[v]$  is the length of *some* path from  $s$  to  $v$  with  $\text{distTo}[v]$  equal to infinity for all  $v$  not reachable from  $s$ . These values are the lengths of *shortest* paths if and only if they satisfy  $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$  for each edge  $e$  from  $v$  to  $w$  (or, in other words, no edge is eligible).

**Proof:** Suppose that  $\text{distTo}[w]$  is the length of a shortest path from  $s$  to  $w$ . If  $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$  for some edge  $e$  from  $v$  to  $w$ , then  $e$  would give a path from  $s$  to  $w$  (through  $v$ ) of length less than  $\text{distTo}[w]$ , a contradiction. Thus the optimality conditions are necessary.

To prove that the optimality conditions are sufficient, suppose that  $w$  is reachable from  $s$  and that  $s = v_0 \rightarrow v_1 \rightarrow v_2 \dots \rightarrow v_k = w$  is a shortest path from  $s$  to  $w$ , of weight  $\text{OPT}_{sw}$ . For  $i$  from 1 to  $k$ , denote the edge from  $v_{i-1}$  to  $v_i$  by  $e_i$ . By the optimality conditions, we have the following sequence of inequalities:

$$\begin{aligned}\text{distTo}[w] &= \text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}() \\ \text{distTo}[v_{k-1}] &\leq \text{distTo}[v_{k-2}] + e_{k-1}.\text{weight}() \\ &\dots \\ \text{distTo}[v_2] &\leq \text{distTo}[v_1] + e_2.\text{weight}() \\ \text{distTo}[v_1] &\leq \text{distTo}[s] + e_1.\text{weight}()\end{aligned}$$

Collapsing these inequalities and eliminating  $\text{distTo}[s] = 0.0$ , we have

$$\text{distTo}[w] \leq e_1.\text{weight}() + \dots + e_k.\text{weight}() = \text{OPT}_{sw}.$$

Now,  $\text{distTo}[w]$  is the length of *some* path from  $s$  to  $w$ , so it cannot be smaller than the length of a *shortest* path. Thus, we have shown that

$$\text{OPT}_{sw} \leq \text{distTo}[w] \leq \text{OPT}_{sw}$$

and equality must hold.

**Certification.** An important practical consequence of PROPOSITION P is its applicability to certification. However an algorithm computes `distTo[]`, we can check whether it contains shortest-path lengths in a single pass through the edges of the graph, checking whether the optimality conditions are satisfied. Shortest-paths algorithms can be complicated, and this ability to efficiently test their outcome is crucial. We include a method `check()` in our implementations on the booksite for this purpose. This method also checks that `edgeTo[]` specifies paths from the source and is consistent with `distTo[]`.

**Generic algorithm.** The optimality conditions lead immediately to a generic algorithm that encompasses all of the shortest-paths algorithms that we consider. For the moment, we restrict attention to nonnegative weights.

**Proposition Q. (Generic shortest-paths algorithm)** Initialize `distTo[s]` to 0 and all other `distTo[]` values to infinity, and proceed as follows:

*Relax any edge in G, continuing until no edge is eligible.*

For all vertices  $w$  reachable from  $s$ , the value of `distTo[w]` after this computation is the length of a shortest path from  $s$  to  $w$  (and `edgeTo[w]` is the last edge on such a path).

**Proof:** Relaxing an edge  $v \rightarrow w$  always sets `distTo[w]` to the length of some path from  $s$  (and `edgeTo[w]` to the last edge on that path). For any vertex  $w$  reachable from  $s$ , some edge on the shortest path to  $w$  is eligible as long as `distTo[w]` remains infinite, so the algorithm continues until the `distTo[]` value of each vertex reachable from  $s$  is the length of some path to that vertex. For any vertex  $v$  for which the shortest path is well-defined, throughout the algorithm `distTo[v]` is the length of some simple path from  $s$  to  $v$  and is strictly monotonically decreasing. Thus, it can decrease at most a finite number of times (once for each simple path from  $s$  to  $v$ ). When no edge is eligible, PROPOSITION P applies.

The key reason for considering the optimality conditions and the generic algorithm is that the generic algorithm *does not specify in which order the edges are to be relaxed*. Thus, all that we need to do to prove that any algorithm computes shortest paths is to prove that it relaxes edges until no edge is eligible.

**Dijkstra's algorithm** In SECTION 4.3, we discussed Prim's algorithm for finding the minimum spanning tree (MST) of an edge-weighted undirected graph: we build the MST by attaching a new edge to a single growing tree at each step. *Dijkstra's algorithm* is an analogous scheme to compute an SPT. We begin by initializing  $\text{dist}[s]$  to 0 and all other  $\text{distTo}[]$  entries to positive infinity, then we *relax and add to the tree a non-tree vertex with the lowest  $\text{distTo}[]$  value, continuing until all vertices are on the tree or no non-tree vertex has a finite  $\text{distTo}[]$  value.*

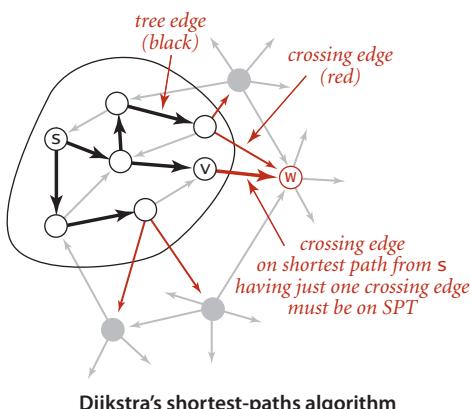
**Proposition R.** Dijkstra's algorithm solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights.

**Proof:** If  $v$  is reachable from the source, every edge  $v \rightarrow w$  is relaxed exactly once, when  $v$  is relaxed, leaving  $\text{distTo}[w] \leftarrow \text{distTo}[v] + e.\text{weight}()$ . This inequality holds until the algorithm completes, since  $\text{distTo}[w]$  can only decrease (any relaxation can only decrease a  $\text{distTo}[]$  value) and  $\text{distTo}[v]$  never changes (because edge weights are nonnegative and we choose the lowest  $\text{distTo}[]$  value at each step, no subsequent relaxation can set any  $\text{distTo}[]$  entry to a lower value than  $\text{distTo}[v]$ ). Thus, after all vertices reachable from  $s$  have been added to the tree, the shortest-paths optimality conditions hold, and PROPOSITION P applies.

**Data structures.** To implement Dijkstra's algorithm we add to our  $\text{distTo}[]$  and  $\text{edgeTo}[]$  data structures an index priority queue  $\text{pq}$  to keep track of vertices that are candidates for being the next to be relaxed. Recall that an `IndexMinPQ` allows us to associate indices with keys (priorities) and to remove and return the index corresponding

to the lowest key. For this application, we always associate a vertex  $v$  with  $\text{distTo}[v]$ , and we have a direct and immediate implementation of Dijkstra's algorithm as stated. Moreover, it is immediate by induction that the  $\text{edgeTo}[]$  entries corresponding to reachable vertices form a tree, the SPT.

**Alternative viewpoint.** Another way to understand the dynamics of the algorithm derives from the proof, diagrammed at left: we have the invariant that  $\text{distTo}[]$  entries for tree vertices are shortest-paths distances and for each vertex  $w$  on the priority queue,  $\text{distTo}[w]$  is the weight of a shortest path from  $s$  to  $w$  that uses only

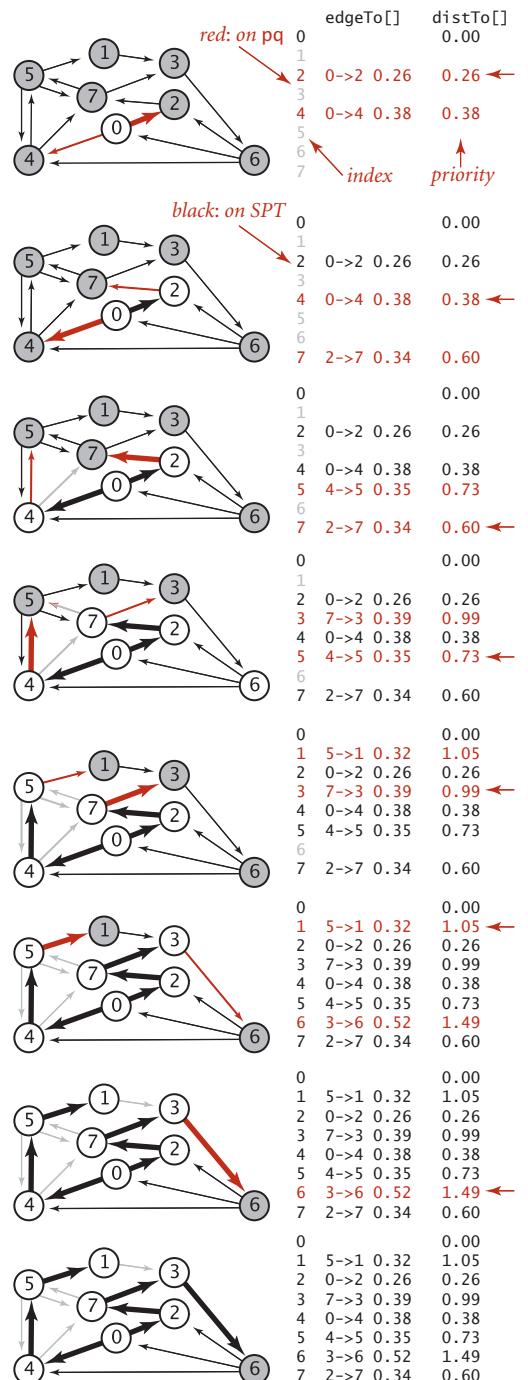


intermediate vertices in the tree and ends in the crossing edge  $\text{edgeTo}[w]$ . The  $\text{distTo}[]$  entry for the vertex with the *smallest* priority is a shortest-path weight, not smaller than the shortest-path weight to any vertex already relaxed, and not larger than the shortest-path weight to any vertex not yet relaxed. That vertex is next to be relaxed. Reachable vertices are relaxed in order of the weight of their shortest path from  $s$ .

The figure at right is a trace for our small sample graph `tinyEWD.txt`. For this example, the algorithm builds the SPT as follows:

- Adds 0 to the tree and its adjacent vertices 2 and 4 to the priority queue.
- Removes 2 from the priority queue, adds  $0 \rightarrow 2$  to the tree, and adds 7 to the priority queue.
- Removes 4 from the priority queue, adds  $0 \rightarrow 4$  to the tree, and adds 5 to the priority queue. Edge  $4 \rightarrow 7$  is ineligible.
- Removes 7 from the priority queue, adds  $2 \rightarrow 7$  to the tree, and adds 3 to the priority queue. Edge  $7 \rightarrow 5$  is ineligible.
- Removes 5 from the priority queue, adds  $4 \rightarrow 5$  to the tree, and adds 1 to the priority queue. Edge  $5 \rightarrow 7$  is ineligible.
- Removes 3 from the priority queue, adds  $7 \rightarrow 3$  to the tree, and adds 6 to the priority queue.
- Removes 1 from the priority queue and adds  $5 \rightarrow 1$  to the tree. Edge  $1 \rightarrow 3$  is ineligible.
- Removes 6 from the priority queue and adds  $3 \rightarrow 6$  to the tree.

Vertices are added to the SPT in increasing order of their distance from the source, as indicated by the red arrows at the right edge of the diagram.



Trace of Dijkstra's algorithm

The implementation of Dijkstra's algorithm in `DijkstraSP` (ALGORITHM 4.9) is a rendition in code of the one-sentence description of the algorithm, enabled by adding one statement to `relax()` to handle two cases: either the `to()` vertex on an edge is not yet on the priority queue, in which case we use `insert()` to add it to the priority queue, or it is already on the priority queue and its priority lowered, in which case `changeKey()` does so.

**Proposition R (continued).** Dijkstra's algorithm uses extra space proportional to  $V$  and time proportional to  $E \log V$  (in the worst case) to solve the single-source shortest paths problem in an edge-weighted digraph with  $E$  edges and  $V$  vertices.

**Proof:** Same as for Prim's algorithm (see PROPOSITION N).

AS WE HAVE INDICATED, ANOTHER WAY TO THINK ABOUT Dijkstra's algorithm is to compare it to Prim's MST algorithm from SECTION 4.3 (see page 622). Both algorithms build a rooted tree by adding an edge to a growing tree: Prim's adds next the non-tree vertex that is closest to the *tree*; Dijkstra's adds next the non-tree vertex that is closest to the *source*. The `marked[]` array is not needed, because the condition `!marked[w]` is equivalent to the condition that `distTo[w]` is infinite. In other words, switching to undirected graphs and edges and omitting the references to `distTo[v]` in the `relax()` code in ALGORITHM 4.9 gives an implementation of ALGORITHM 4.7, the eager version of Prim's algorithm (!). Also, a lazy version of Dijkstra's algorithm along the lines of `LazyPrimMST` (page 619) is not difficult to develop.

**Variants.** Our implementation of Dijkstra's algorithm, with suitable modifications, is effective for solving other versions of the problem, such as the following:

**Single-source shortest paths in undirected graphs.** Given an edge-weighted *undirected* graph and a source vertex  $s$ , support queries of the form *Is there a path from  $s$  to a given target vertex  $v$ ?* If so, find a *shortest* such path (one whose total weight is minimal).

The solution to this problem is immediate if we view the undirected graph as a digraph. That is, given an undirected graph, build an edge-weighted digraph with the same vertices and with two directed edges (one in each direction) corresponding to each edge in the graph. There is a one-to-one correspondence between paths in the digraph and paths in the graph, and the costs of the paths are the same—the shortest-paths problems are equivalent.

---

**ALGORITHM 4.9 Dijkstra's shortest-paths algorithm**


---

```

public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;
    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        pq.insert(s, 0.0);
        while (!pq.isEmpty())
            relax(G, pq.delMin());
    }
    private void relax(EdgeWeightedDigraph G, int v)
    {
        for (DirectedEdge e : G.adj(v))
        {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight())
            {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.changeKey(w, distTo[w]);
                else                  pq.insert(w, distTo[w]);
            }
        }
    }
    public double distTo(int v)          // standard client query methods
    public boolean hasPathTo(int v)      // for SPT implementations
    public Iterable<Edge> pathTo(int v) // (See page 649.)
}

```

---

This implementation of Dijkstra's algorithm grows the SPT by adding an edge at a time, always choosing the edge from a tree vertex to a non-tree vertex whose destination  $w$  is closest to  $s$ .

**Source-sink shortest paths.** Given an edge-weighted digraph, a source vertex  $s$ , and a target vertex  $t$ , find the shortest path from  $s$  to  $t$ .

To solve this problem, use Dijkstra's algorithm, but terminate the search as soon as  $t$  comes off the priority queue.

**All-pairs shortest paths.** Given an edge-weighted digraph, support queries of the form *Given a source vertex  $s$  and a target vertex  $t$ , is there a path from  $s$  to  $t$ ?* If so, find a *shortest* such path (one whose total weight is minimal).

The surprisingly compact implementation at left below solves the all-pairs shortest paths problem, using time and space proportional to  $EV\log V$ . It builds an array of `DijkstraSP` objects, one for each vertex as the source. To answer a client query, it uses the source to access the corresponding single-source shortest-paths object and then passes the target as argument to the query.

**Shortest paths in Euclidean graphs.** Solve the single-source, source-sink, and all-pairs shortest-paths problems in graphs where vertices are points in the plane and edge weights are proportional to Euclidean distances between vertices.

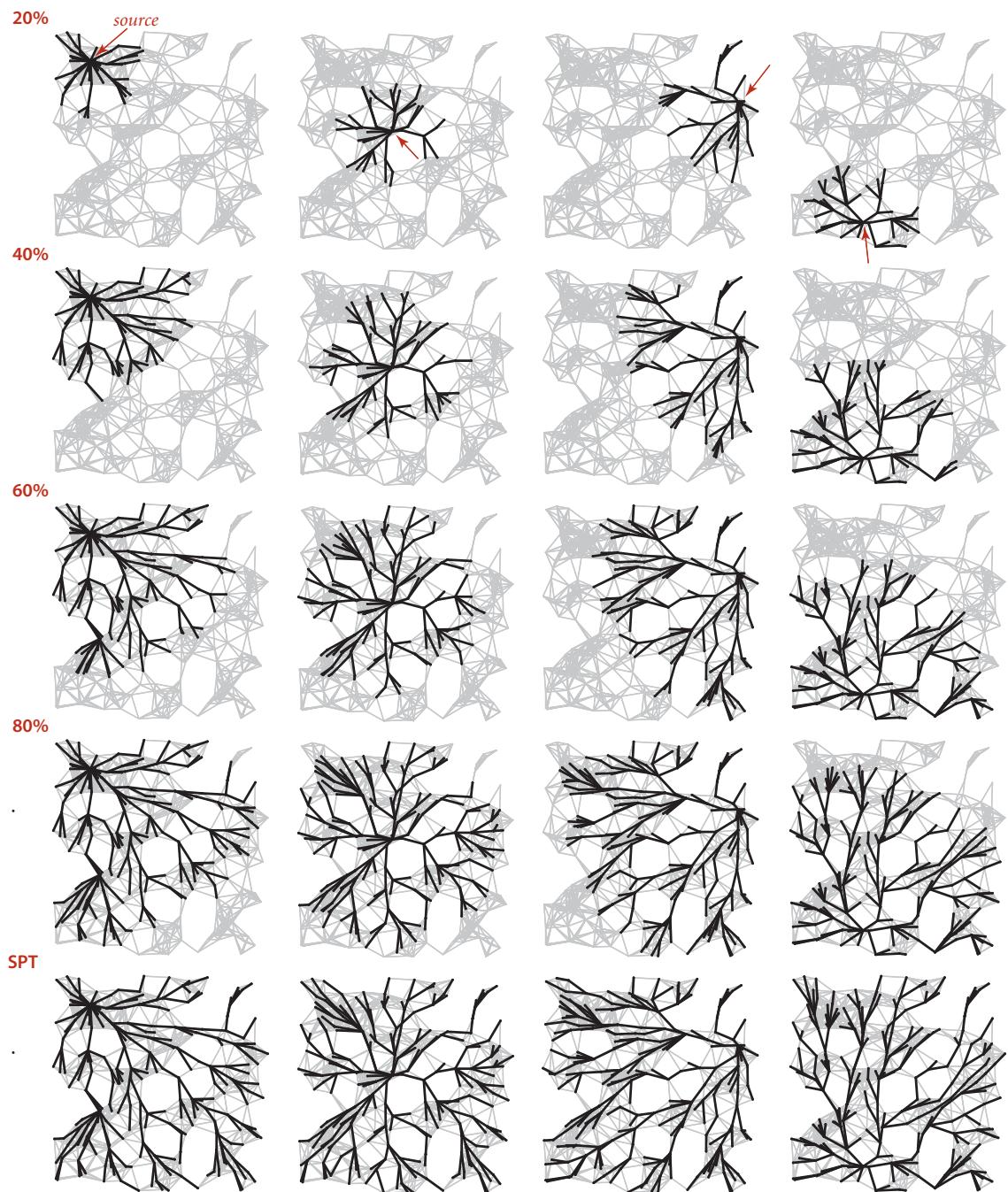
A simple modification considerably speeds up Dijkstra's algorithm in this case (see EXERCISE 4.4.27).

THE FIGURES ON THE FACING PAGE show the emergence of the SPT as computed by Dijkstra's algorithm for the Euclidean graph defined by our test file `mediumEWD.txt` (see page 645) for several different sources. Recall that line segments in this graph represent directed edges in both directions. Again, these figures illustrate a fascinating dynamic process.

```
public class DijkstraAllPairsSP
{
    private DijkstraSP[] all;
    DijkstraAllPairsSP(EdgeWeightedDigraph G)
    {
        all = new DijkstraSP[G.V()]
        for (int v = 0; v < G.V(); v++)
            all[v] = new DijkstraSP(G, v);
    }
    Iterable<DirectedEdge> path(int s, int t)
    { return all[s].pathTo(t); }
    double dist(int s, int t)
    { return all[s].distTo(t); }
}
```

All-pairs shortest paths

Next, we consider shortest-paths algorithms for acyclic edge-weighted graphs, where we can solve the problem in linear time (faster than Dijkstra's algorithm) and then for edge-weighted digraphs with negative weights, where Dijkstra's algorithm does not apply.



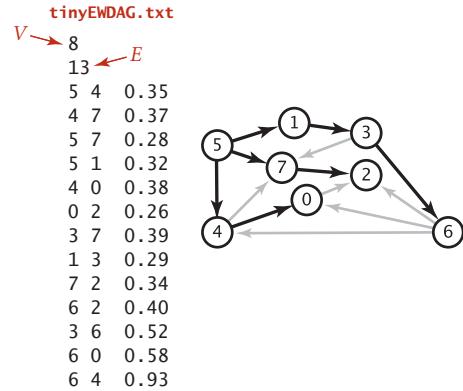
Dijkstra's algorithm (250 vertices, various sources)

**Acyclic edge-weighted digraphs** For many natural applications, edge-weighted digraphs are known to have no directed cycles. For economy, we use the equivalent term *edge-weighted DAG* to refer to an acyclic edge-weighted digraph. We now consider an algorithm for finding shortest paths that is simpler and faster than Dijkstra's algorithm for edge-weighted DAGs. Specifically, it

- Solves the single-source problem in linear time
- Handles negative edge weights
- Solves related problems, such as finding *longest* paths.

These algorithms are straightforward extensions to the algorithm for topological sort in DAGs that we considered in SECTION 4.2.

Specifically, vertex relaxation, in combination with topological sorting, immediately presents a solution to the single-source shortest-paths problem for edge-weighted DAGs. We initialize `distTo[s]` to 0 and all other `distTo[]` values to infinity, then relax the vertices, one by one, *taking the vertices in topological order*. An argument similar to (but simpler than) the argument that we used for Dijkstra's algorithm on page 652 establishes the effectiveness of this method:



An acyclic edge-weighted digraph with an SPT

**Proposition S.** By relaxing vertices in topological order, we can solve the single-source shortest-paths problem for edge-weighted DAGs in time proportional to  $E + V$ .

**Proof:** Every edge  $v \rightarrow w$  is relaxed exactly once, when  $v$  is relaxed, leaving `distTo[w] <= distTo[v] + e.weight()`. This inequality holds until the algorithm completes, since `distTo[v]` never changes (because of the topological order, no edge pointing to  $v$  will be processed after  $v$  is relaxed) and `distTo[w]` can only decrease (any relaxation can only decrease a `distTo[]` value). Thus, after all vertices reachable from  $s$  have been added to the tree, the shortest-paths optimality conditions hold, and PROPOSITION Q applies. The time bound is immediate: PROPOSITION G on page 583 tells us that the topological sort takes time proportional to  $E + V$ , and the second relaxation pass completes the job by relaxing each edge once, again in time proportional to  $E + V$ .

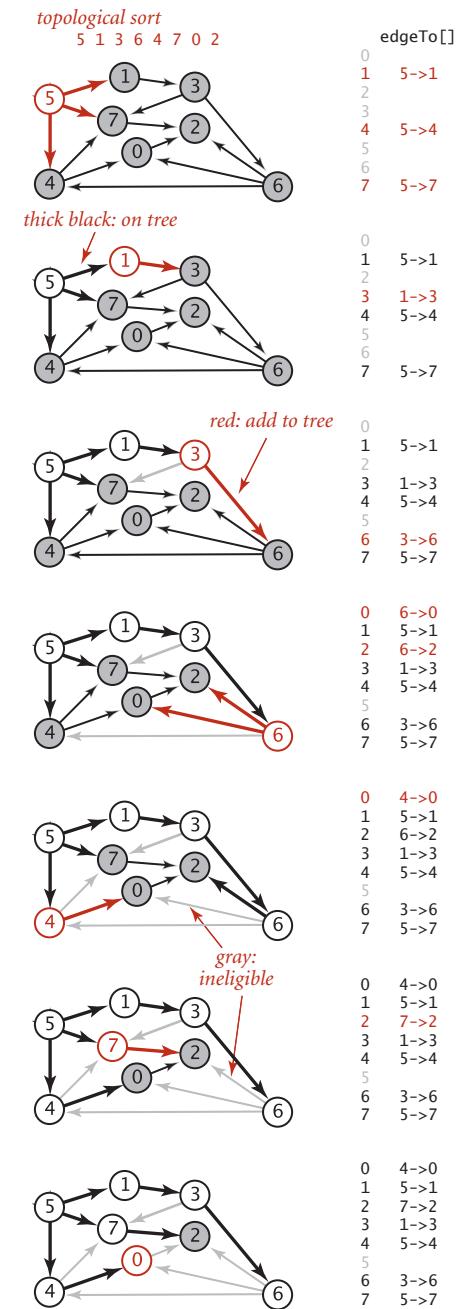
The figure at right is a trace for a sample acyclic edge-weighted digraph `tinyEWdag.txt`. For this example, the algorithm builds the shortest-paths tree from vertex 5 as follows:

- Does a DFS to discover the topological order  
5 1 3 6 4 7 0 2.
- Adds to the tree 5 and all edges leaving it.
- Adds to the tree 1 and 1->3.
- Adds to the tree 3 and 3->6, but not 3->7, which is ineligible.
- Adds to the tree 6 and edges 6->2 and 6->0, but not 6->4, which is ineligible.
- Adds to the tree 4 and 4->0, but not 4->7, which is ineligible. Edge 6->0 becomes ineligible.
- Adds to the tree 7 and 7->2. Edge 6->2 becomes ineligible.
- Adds 0 to the tree, but not its incident edge 0->2, which is ineligible.
- Adds 2 to the tree.

The addition of 2 to the tree is not depicted; the last vertex in a topological sort has no edges leaving it.

The implementation, shown in ALGORITHM 4.10, is a straightforward application of code we have already considered. It assumes that `Topological` has overloaded methods for the topological sort, using the `EdgeWeightedDigraph` and `DirectedEdge` APIs of this section (see EXERCISE 4.4.12). Note that our boolean array `marked[]` is not needed in this implementation: since we are processing vertices in an acyclic digraph in topological order, we never re-encounter a vertex that we have already relaxed. ALGORITHM 4.10 could hardly be more efficient: after the topological sort, the constructor scans the graph, relaxing each edge exactly once. It is the method of choice for finding shortest paths in edge-weighted graphs that are known to be acyclic.

PROPOSITION s is significant because it provides a concrete example where the absence of cycles



Trace for shortest paths in an edge-weighted DAG

**ALGORITHM 4.10** Shortest paths in edge-weighted DAGs

```

public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        Topological top = new Topological(G);
        for (int v : top.order())
            relax(G, v);
    }

    private void relax(EdgeWeightedDigraph G, int v)
    // See page 648.

    public double distTo(int v)           // standard client query methods
    public boolean hasPathTo(int v)       // for SPT implementations
    public Iterable<DirectedEdge> pathTo(int v) // (See page 649.)
}

```

This shortest-paths algorithm for edge-weighted DAGs uses a topological sort (ALGORITHM 4.5, adapted to use `EdgeWeightedDigraph` and `DirectedEdge`) to enable it to relax the vertices in topological order, which is all that is needed to compute shortest paths.

```

% java AcyclicSP tinyEWDAG.txt 5
5 to 0 (0.73): 5->4 0.35  4->0 0.38
5 to 1 (0.32): 5->1 0.32
5 to 2 (0.62): 5->7 0.28  7->2 0.34
5 to 3 (0.62): 5->1 0.32  1->3 0.29
5 to 4 (0.35): 5->4 0.35
5 to 5 (0.00):
5 to 6 (1.13): 5->1 0.32  1->3 0.29  3->6 0.52
5 to 7 (0.28): 5->7 0.28

```

considerably simplifies a problem. For shortest paths, the topological-sort-based method is faster than Dijkstra's algorithm by a factor proportional to the cost of the priority-queue operations in Dijkstra's algorithm. Moreover, the proof of PROPOSITION S does not depend on the edge weights being nonnegative, so we can remove that restriction for edge-weighted DAGs. Next, we consider implications of this ability to allow negative edge weights, by considering the use of the shortest-paths model to solve two other problems, one of which seems at first blush to be quite removed from graph processing.

**Longest paths.** Consider the problem of finding the *longest* path in an edge-weighted DAG with edge weights that may be positive or negative.

**Single-source longest paths in edge-weighted DAGs.** Given an edge-weighted DAG (with negative weights allowed) and a source vertex  $s$ , support queries of the form: *Is there a directed path from  $s$  to a given target vertex  $v$ ?* If so, find a *longest* such path (one whose total weight is *maximal*).

The algorithm just considered provides a quick solution to this problem:

**Proposition T.** We can solve the longest-paths problem in edge-weighted DAGs in time proportional to  $E + V$ .

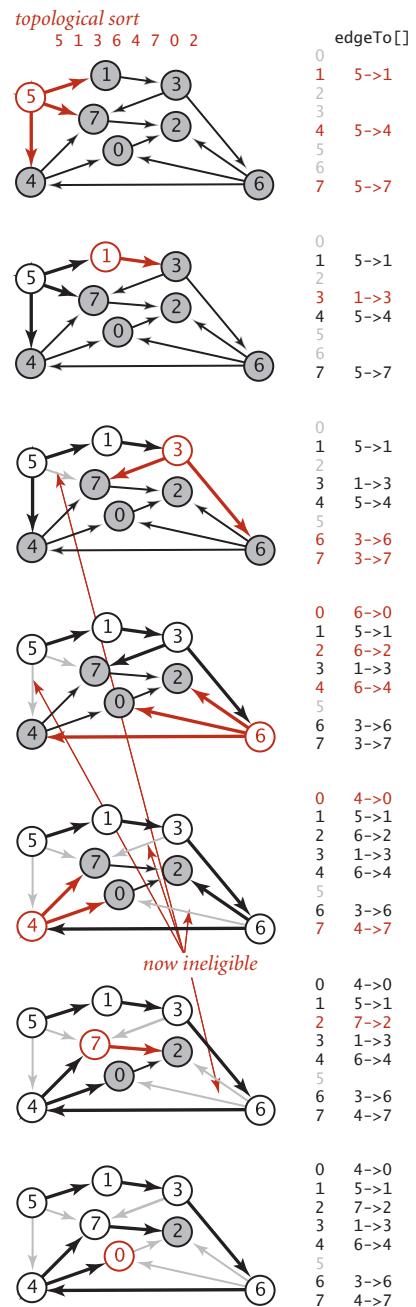
**Proof:** Given a longest-paths problem, create a copy of the given edge-weighted DAG that is identical to the original, except that all edge weights are negated. Then the *shortest* path in this copy is the *longest* path in the original. To transform the solution of the shortest-paths problem to a solution of the longest-paths problem, negate the weights in the solution. The running time follows immediately from PROPOSITION S.

Using this transformation to develop a class `AcyclicLP` that finds longest paths in edge-weighted DAGs is straightforward. An even simpler way to implement such a class is to copy `AcyclicSP`, then switch the `distTo[]` initialization to `Double.NEGATIVE_INFINITY` and switch the sense of the inequality in `relax()`. Either way, we get an efficient solution to the longest-paths problem in edge-weighted DAGs. This result is to be compared with the fact that the best known algorithm for finding longest simple paths in general edge-weighted digraphs (where edge weights may be negative) requires *exponential* time in the worst case (see CHAPTER 6)! The possibility of cycles seems to make the problem exponentially more difficult.

The figure at right is a trace of the process of finding longest paths in our sample edge-weighted DAG `tinyEWDAG.txt`, for comparison with the shortest-paths trace for the same DAG on page 659. For this example, the algorithm builds the longest-paths tree (LPT) from vertex 5 as follows:

- Does a DFS to discover the topological order 5 1 3 6 4 7 0 2.
- Adds to the tree 5 and all edges leaving it.
- Adds to the tree 1 and 1->3.
- Adds to the tree 3 and edges 3->6 and 3->7. Edge 5->7 becomes ineligible.
- Adds to the tree 6 and edges 6->2, 6->4, and 6->0.
- Adds to the tree 4 and edges 4->0 and 4->7. Edges 6->0 and 3->7 become ineligible.
- Adds to the tree 7 and 7->2. Edge 6->2 becomes ineligible.
- Adds 0 to the tree, but not 0->2, which is ineligible.
- Adds 2 to the tree (not depicted).

The longest-paths algorithm processes the vertices in the same order as the shortest-paths algorithm but produces a completely different result.



Trace for longest paths in an acyclic network

**Parallel job scheduling.** As an example application, we revisit the class of *scheduling* problems that we first considered in SECTION 4.2 (page 574). Specifically, consider the following scheduling problem (differences from the problem on page 575 are italicized):

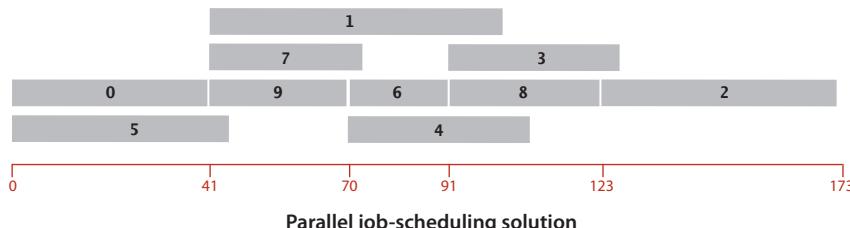
**Parallel precedence-constrained scheduling.** Given a set of jobs of *specified duration* to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs *on identical processors (as many as needed)* such that they are all completed *in the minimum amount of time* while still respecting the constraints?

Implicit in the model of SECTION 4.2 is a single processor: we schedule the jobs in topological order and the total time required is the total duration of the jobs. Now, we assume that we have sufficient processors to perform as many jobs as possible, limited only by precedence constraints. Again, thousands or even millions of jobs might be involved, so we require an efficient algorithm. Remarkably, a *linear-time* algorithm is available—an approach known as the *critical path method* demonstrates that the problem is equivalent to a longest-paths problem in an edge-weighted DAG. This method has been used successfully in countless industrial applications.

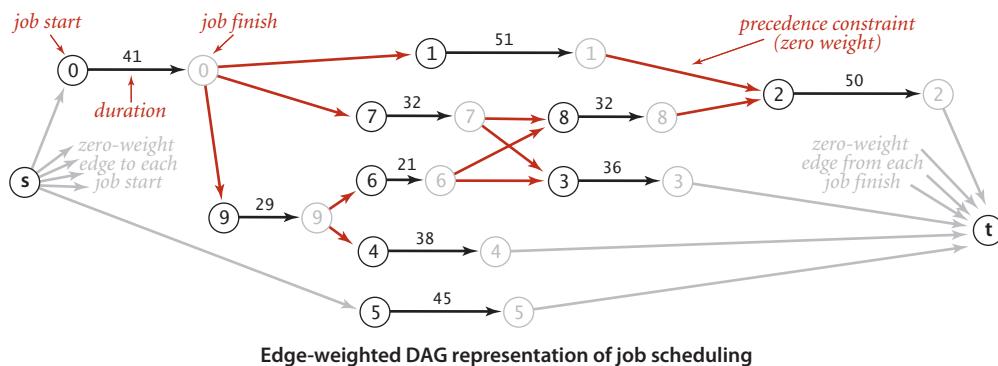
We focus on the earliest possible time that we can schedule each job, assuming that any available processor can handle the job for its duration. For example, consider the problem instance specified in the table at right. The solution below shows that 173.0 is the minimum possible completion time for any schedule for this problem: the schedule satisfies all the constraints, and no schedule can complete before time 173.0 because of the job sequence 0->9->6->8->2. This sequence is known as a *critical path* for this problem. Every sequence of jobs, each constrained to follow the job just preceding it in the sequence, represents a lower bound on the length of the schedule. If we define the length of such a sequence to be its earliest possible completion time (total of the durations of its jobs), the longest sequence is known as a critical path because any delay in the starting time of any job delays the best achievable completion time of the entire project.

job	duration	must complete before		
0	41.0	1	7	9
1	51.0		2	
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0		2	
9	29.0	4	6	

A job-scheduling problem

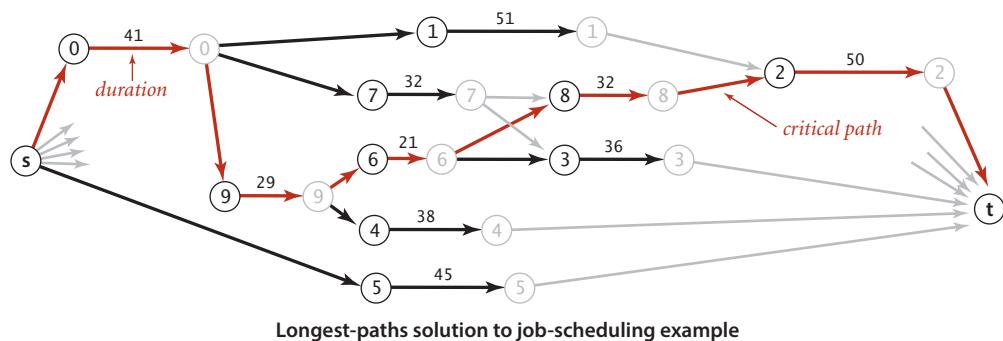


Parallel job-scheduling solution



**Definition.** The *critical path method* for parallel scheduling is to proceed as follows: Create an edge-weighted DAG with a source  $s$ , a sink  $t$ , and two vertices for each job (a *start* vertex and an *end* vertex). For each job, add an edge from its start vertex to its end vertex with weight equal to its duration. For each precedence constraint  $v \rightarrow w$ , add a zero-weight edge from the end vertex corresponding to  $v$  to the beginning vertex corresponding to  $w$ . Also add zero-weight edges from the source to each job's start vertex and from each job's end vertex to the sink. Now, schedule each job at the time given by the length of its longest path from the source.

The figure at the top of this page depicts this correspondence for our sample problem, and the figure at the bottom of the page gives the longest-paths solution. As specified, the graph has three edges for each job (zero-weight edges from the source to the start and from the finish to the sink, and an edge from start to finish) and one edge for each precedence constraint. The class `CPM` on the facing page is a straightforward implementation of the critical path method. It transforms any instance of the job-scheduling problem into an instance of the longest-paths problem in an edge-weighted DAG, uses `AcyclicLP` to solve it, then prints the job start times and schedule finish time.



## Critical path method for parallel precedence-constrained job scheduling

```

public class CPM
{
    public static void main(String[] args)
    {
        int N = StdIn.readInt(); StdIn.readLine();
        EdgeWeightedDigraph G;
        G = new EdgeWeightedDigraph(2*N+2);

        int s = 2*N, t = 2*N+1;
        for (int i = 0; i < N; i++)
        {
            String[] a = StdIn.readLine().split("\s+");
            double duration = Double.parseDouble(a[0]);
            G.addEdge(new DirectedEdge(i, i+N, duration));
            G.addEdge(new DirectedEdge(s, i, 0.0));
            G.addEdge(new DirectedEdge(i+N, t, 0.0));
            for (int j = 1; j < a.length; j++)
            {
                int successor = Integer.parseInt(a[j]);
                G.addEdge(new DirectedEdge(i+N, successor, 0.0));
            }
        }

        AcyclicLP lp = new AcyclicLP(G, s);

        StdOut.println("Start times:");
        for (int i = 0; i < N; i++)
            StdOut.printf("%4d: %5.1f\n", i, lp.distTo(i));
        StdOut.printf("Finish time: %5.1f\n", lp.distTo(t));
    }
}

```

This implementation of the critical path method for job scheduling reduces the problem directly to the longest-paths problem in edge-weighted DAGs. It builds an edge-weighted digraph (which must be a DAG) from the job-scheduling problem specification, as prescribed by the critical path method, then uses `AcyclicLP` (see PROPOSITION T) to find the longest-paths tree and to print the longest-paths lengths, which are precisely the start times for each job.

```
% more jobsPC.txt
10
41.0 1 7 9
51.0 2
50.0
36.0
38.0
45.0
21.0 3 8
32.0 3 8
32.0 2
29.0 4 6
```

```
% java CPM < jobsPC.txt
Start times:
0: 0.0
1: 41.0
2: 123.0
3: 91.0
4: 70.0
5: 0.0
6: 70.0
7: 41.0
8: 91.0
9: 41.0
Finish time: 173.0
```

**original**

<i>job</i>	<i>start</i>
0	0.0
1	41.0
2	123.0
3	91.0
4	70.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

**2 by 12.0 after 4**

<i>job</i>	<i>start</i>
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	41.0
8	91.0
9	41.0

**2 by 70.0 after 7**

<i>job</i>	<i>start</i>
0	0.0
1	41.0
2	123.0
3	91.0
4	111.0
5	0.0
6	70.0
7	53.0
8	91.0
9	41.0

**4 by 80.0 after 0**

infeasible!

Relative deadlines  
in job scheduling**Proposition U.** The critical path method solves the parallel precedence-constrained scheduling problem in linear time.

**Proof:** Why does the CPM approach work? The correctness of the algorithm rests on two facts. First, every path in the DAG is a sequence of job starts and job finishes, separated by zero-weight precedence constraints—the length of any path from the source  $s$  to any vertex  $v$  in the graph is a lower bound on the start/finish time represented by  $v$ , because we could not do better than scheduling those jobs one after another on the same machine. In particular, the length of the longest path from  $s$  to the sink  $t$  is a lower bound on the finish time of all the jobs. Second, all the start and finish times implied by longest paths are *feasible*—every job starts after the finish of all the jobs where it appears as a successor in a precedence constraint, because the start time is the length of the *longest* path from the source to it. In particular, the length of the longest path from  $s$  to  $t$  is an *upper* bound on the finish time of all the jobs. The linear-time performance is immediate from PROPOSITION T.

<i>job</i>	<i>time</i>	<i>relative to</i>
2	12.0	4
2	70.0	7
4	80.0	0

Added deadlines  
for job scheduling

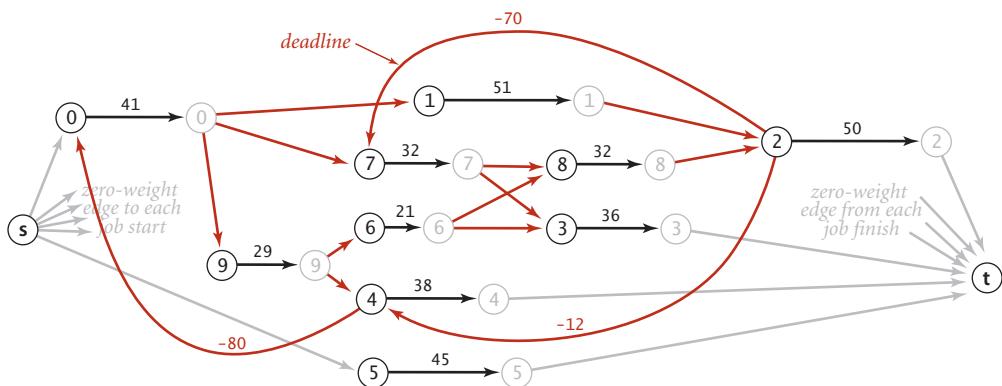
**Parallel job scheduling with relative deadlines.** Conventional deadlines are relative to the start time of the first job. Suppose that we allow an additional type of constraint in the job-scheduling problem to specify that a job must begin before a specified amount of time has elapsed, relative to the start time of another job. Such constraints are commonly needed in time-critical manufacturing processes and in many other applications, but they can make the job-scheduling problem considerably more difficult to solve. For example, as shown at left, suppose that we need to add a constraint to our example that job 2 must start no later than 12 time units after job 4 starts. This deadline is actually a constraint on the start time of job 4: it must be no earlier than 12 time units before the start time of job 2. In our example, there is room in the schedule to meet the deadline: we can move the start time of job 4 to 111, 12 time units before the scheduled start time of job 2. Note that, if job 4 were a long job, this change would increase the finish time of the whole schedule. Similarly, if we add to the schedule a deadline that job 2 must start no later than 70 time units after job 7 starts, there is room in the schedule to change the start time of job 7 to 53, without having to reschedule jobs 3 and 8. But if we add a deadline that job 4 must start no later

than 80 time units after job 0, the schedule becomes *infeasible*: the constraints that 4 must start no more than 80 time units after job 0 and that job 2 must start no more than 12 units after job 4 imply that job 2 must start no more than 92 time units after job 0, but job 2 must start at least 123 time units after job 0 because of the chain 0 (41 time units) precedes 9 (29 time units) precedes 6 (21 time units) precedes 8 (32 time units) precedes 2. Adding more deadlines of course multiplies the possibilities and turns an easy problem into a difficult one.

**Proposition V.** Parallel job scheduling with relative deadlines is a shortest-paths problem in edge-weighted digraphs (with cycles and negative weights allowed).

**Proof:** Use the same construction as in PROPOSITION U, adding an edge for each deadline: if job  $v$  has to start within  $d$  time units of the start of job  $w$ , add an edge from  $v$  to  $w$  with *negative* weight  $d$ . Then convert to a shortest-paths problem by negating all the weights in the digraph. The proof of correctness applies, *provided that the schedule is feasible*. Determining whether a schedule is feasible is part of the computational burden, as you will see.

This example illustrates that negative weights can play a critical role in practical application models. It says that if we can find an efficient solution to the shortest-paths problem with negative weights, then we can find an efficient solution to the parallel job scheduling problem with relative deadlines. Neither of the algorithms we have considered can do the job: Dijkstra's algorithm requires that weights be positive (or zero), and ALGORITHM 4.10 requires that the digraph be acyclic. Next, we consider the problem of coping with negative edge weights in digraphs that are not necessarily acyclic.



Edge-weighted digraph representation of parallel precedence-constrained scheduling with relative deadlines

**Shortest paths in general edge-weighted digraphs** Our job-scheduling-with-deadlines example just discussed demonstrates that negative weights are not merely a mathematical curiosity; on the contrary, they significantly extend the applicability of the shortest-paths problem as a problem-solving model. Accordingly we now consider algorithms for edge-weighted digraphs that may have *both* cycles and negative

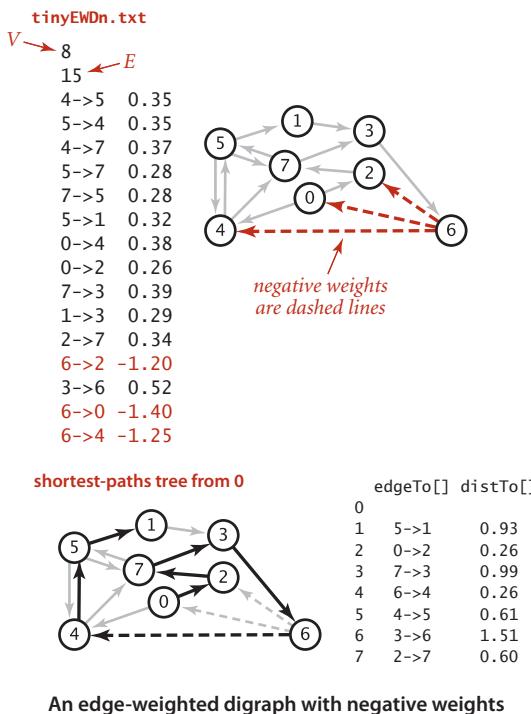
weights. Before doing so, we consider some basic properties of such digraphs to reset our intuition about shortest paths. The figure at left is a small example that illustrates the effects of introducing negative weights on a digraph's shortest paths. Perhaps the most important effect is that when negative weights are present, low-weight shortest paths tend to have *more* edges than higher-weight paths. For positive weights, our emphasis was on looking for shortcuts; but when negative weights are present, we seek *detours* that use negative-weight edges. This effect turns our intuition in seeking “short” paths into a liability in understanding the algorithms, so we need to suppress that line of intuition and consider the problem on a basic abstract level.

**Strawman I.** The first idea that suggests itself is to find the smallest (most negative) edge weight, then to add the absolute value of that number to all the edge weights to transform the digraph into one with no negative weights. This naive approach does not work at all, because shortest

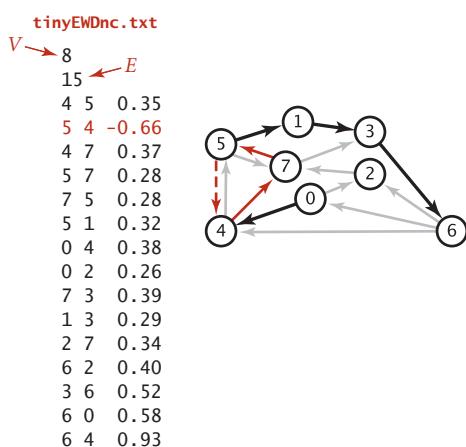
paths in the new digraph bear little relation to shortest paths in the old one. The more edges a path has, the more it is penalized by this transformation (see EXERCISE 4.4.14).

**Strawman II.** The second idea that suggests itself is to try to adapt Dijkstra's algorithm in some way. The fundamental difficulty with this approach is that the algorithm depends on examining vertices in increasing order of their distance from the source. The proof in PROPOSITION R that the algorithm is correct assumes that adding an edge to a path makes that path longer. But any edge with negative weight makes the path *shorter*, so that assumption is unfounded (see EXERCISE 4.4.14).

**Negative cycles.** When we consider digraphs that could have negative edge weights, the concept of a shortest path is meaningless if there is a cycle in the digraph that



An edge-weighted digraph with negative weights



shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

An edge-weighted digraph with a negative cycle

has negative weight. For example, consider the digraph at left, which is identical to our first example except that edge  $5 \rightarrow 4$  has weight  $-.66$ . Then, the weight of the cycle  $4 \rightarrow 7 \rightarrow 5 \rightarrow 4$  is

$$.37 + .28 - .66 = -.01$$

We can spin around that cycle to generate arbitrarily short paths! Note that it is not necessary for all the edges on a directed cycle to be of negative weight; what matters is the *sum* of the edge weights.

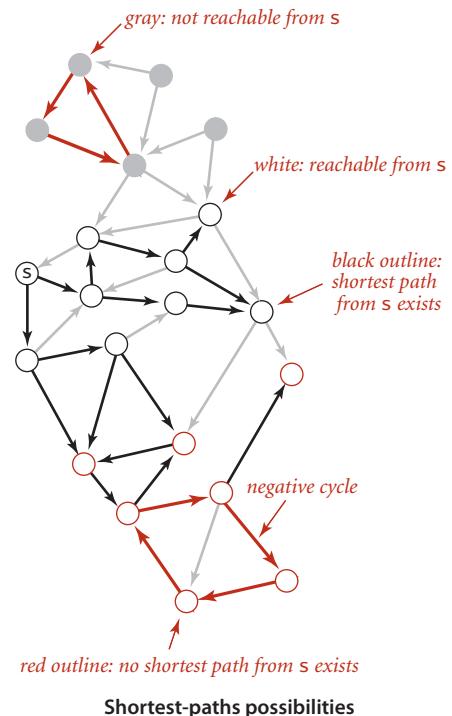
**Definition.** A *negative cycle* in an edge-weighted digraph is a directed cycle whose total weight (sum of the weights of its edges) is negative.

Now, suppose that some vertex on a path from  $s$  to a reachable vertex  $v$  is also on a negative cycle. In this case, the existence of a shortest path from  $s$  to  $v$  would be a contradiction, because we could use the cycle to construct a path with weight lower than any given value. In other words, shortest paths can be an ill-posed problem if negative cycles are present.

**Proposition W.** There exists a shortest path from  $s$  to  $v$  in an edge-weighted digraph if and only if there exists at least one directed path from  $s$  to  $v$  and no vertex on any directed path from  $s$  to  $v$  is on a negative cycle.

**Proof:** See discussion above and EXERCISE 4.4.29.

Note that the requirement that shortest paths have no vertices on negative cycles implies that shortest paths are simple and that we can compute a shortest-paths tree for such vertices, as we have done for positive edge weights.



**Strawman III.** Whether or not there are negative cycles, there exists a shortest *simple* path connecting the source to each vertex reachable from the source. Why not define shortest paths so that we seek such paths? Unfortunately, the best known algorithm for solving this problem takes exponential time in the worst case (see CHAPTER 6). Generally, we consider such problems “too difficult to solve” and study simpler versions.

THUS, A WELL-POSED AND TRACTABLE VERSION of the shortest paths problem in edge-weighted digraphs is to require algorithms to

- Assign a shortest-path weight of  $+\infty$  to vertices that are not reachable from the source
- Assign a shortest-path weight of  $-\infty$  to vertices that are on a path from the source that has a vertex that is on a negative cycle
- Compute the shortest-path weight (and tree) for all other vertices

Throughout this section, we have been placing restrictions on the shortest-paths problem so that we can develop algorithms to solve it. First, we disallowed negative weights, then we disallowed directed cycles. We now adopt these less stringent restrictions and focus on the following problems in general digraphs:

**Negative cycle detection.** Does a given edge-weighted digraph have a negative cycle? If it does, find one such cycle.

**Single-source shortest paths when negative cycles are not reachable.** Given an edge-weighted digraph and a source  $s$  with no negative cycles reachable from  $s$ , support queries of the form *Is there a directed path from  $s$  to a given target vertex  $v$ ?* If so, find a *shortest* such path (one whose total weight is minimal).

TO SUMMARIZE: while shortest paths in digraphs with negative cycles is an ill-posed problem and we *cannot* efficiently solve the problem of finding simple shortest paths in such digraphs, we *can* identify negative cycles in practical situations. For example, in a job-scheduling-with-deadlines problem, we might expect negative cycles to be relatively rare: constraints and deadlines derive from logical real-world constraints, so any negative cycles are likely to stem from an error in the problem statement. Finding negative cycles, correcting errors, and then finding the schedule in a problem with no negative cycles is a reasonable way to proceed. In other cases, finding a negative cycle is the goal of the computation. The following approach, developed by R. Bellman and L. Ford in the late 1950s, provides a simple and effective basis for attacking both of these problems and is also effective for digraphs with positive weights:

**Proposition X. (Bellman-Ford algorithm)** The following method solves the single-source shortest-paths problem from a given source  $s$  for any edge-weighted digraph with  $V$  vertices and no negative cycles reachable from  $s$ : Initialize  $\text{distTo}[s]$  to 0 and all other  $\text{distTo}[]$  values to infinity. Then, considering the digraph's edges in any order, relax all edges. Make  $V$  such passes.

**Proof:** For any vertex  $t$  that is reachable from  $s$  consider a specific shortest path from  $s$  to  $t$ :  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k$ , where  $v_0$  is  $s$  and  $v_k$  is  $t$ . Since there are no negative cycles, such a path exists and  $k$  can be no larger than  $V-1$ . We show by induction on  $i$  that after the  $i$ th pass the algorithm computes a shortest path from  $s$  to  $v_i$ . The base case ( $i = 0$ ) is trivial. Assuming the claim to be true for  $i$ ,  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i$  is a shortest path from  $s$  to  $v_i$ , and  $\text{distTo}[v_i]$  is its length. Now, we relax every edge in the  $i$ th pass, including  $v_i \rightarrow v_{i+1}$ , so  $\text{distTo}[v_{i+1}]$  is no greater than  $\text{distTo}[v_i]$  plus the weight of  $v_i \rightarrow v_{i+1}$ . Now, after the  $i$ th pass,  $\text{distTo}[v_{i+1}]$  must be equal to  $\text{distTo}[v_i]$  plus the weight of  $v_i \rightarrow v_{i+1}$ . It cannot be greater because we relax every edge in the  $i$ th pass, in particular  $v_i \rightarrow v_{i+1}$ , and it cannot be less because that is the length of  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_{i+1}$ , a shortest path. Thus the algorithm computes a shortest path from  $s$  to  $v_{i+1}$  after the  $(i+1)$ st pass.

**Proposition W (continued).** The Bellman-Ford algorithm takes time proportional to  $EV$  and extra space proportional to  $V$ .

**Proof:** Each of the  $V$  passes relaxes  $E$  edges.

This method is very general, since it does not specify the order in which the edges are relaxed. We now restrict attention to a less general method where we always relax all the edges leaving any vertex (in any order). The following code exhibits the simplicity of the approach:

```
for (int pass = 0; pass < G.V(); pass++)
    for (v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

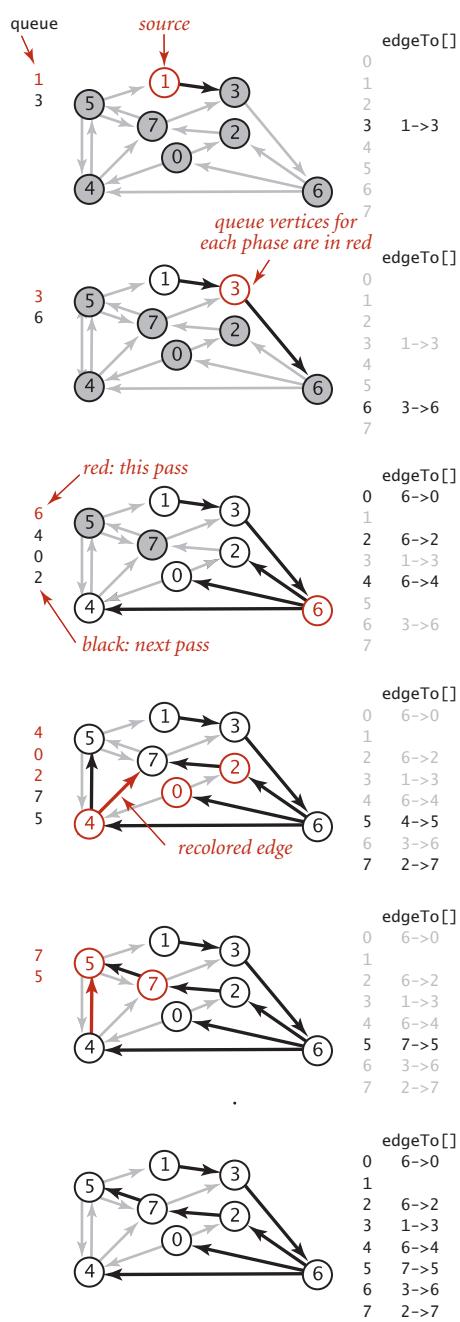
We do not consider this version in detail because it *always* relaxes  $VE$  edges, and a simple modification makes the algorithm much more efficient for typical applications.

**Queue-based Bellman-Ford.** Specifically, we can easily determine *a priori* that numerous edges are not going to lead to a successful relaxation in any given pass: the only edges that could lead to a change in `distTo[]` are those leaving a vertex whose `distTo[]` value changed in the previous pass. To keep track of such vertices, we use a FIFO queue. The operation of the algorithm for our standard example with positive weights is shown at right. Shown at the left of the figure are the queue entries for each pass (in red), followed by the queue entries for the next pass (in black). We start with the source on the queue and then compute the SPT as follows:

- Relax  $1 \rightarrow 3$  and put 3 on the queue.
  - Relax  $3 \rightarrow 6$  and put 6 on the queue.
  - Relax  $6 \rightarrow 4$ ,  $6 \rightarrow 0$ , and  $6 \rightarrow 2$  and put 4, 0, and 2 on the queue.
  - Relax  $4 \rightarrow 7$  and  $4 \rightarrow 5$  and put 7 and 5 on the queue. Then relax  $0 \rightarrow 4$  and  $0 \rightarrow 2$ , which are ineligible. Then relax  $2 \rightarrow 7$  (and recolor  $4 \rightarrow 7$ ).
  - Relax  $7 \rightarrow 5$  (and recolor  $4 \rightarrow 5$ ) but do not put 5 on the queue (it is already there). Then relax  $7 \rightarrow 3$ , which is ineligible. Then relax  $5 \rightarrow 1$ ,  $5 \rightarrow 4$ , and  $5 \rightarrow 7$ , which are ineligible, leaving the queue empty.

**Implementation.** Implementing the Bellman-Ford algorithm along these lines requires remarkably little code, as shown in ALGORITHM 4.11. It is based on two additional data structures:

- A queue `queue` of vertices to be relaxed
  - A vertex-indexed `boolean` array `onQ[]` that indicates which vertices are on the queue, to avoid duplicates



## Trace of the Bellman-Ford algorithm

We start by putting the source  $s$  on the queue, then enter a loop where we take a vertex off the queue and relax it. To add vertices to the queue, we augment our `relax()` implementation from page 648 to put the vertex pointed to by any edge that successfully relaxes onto the queue, as shown in the code below. The data structures ensure that

- Only one copy of each vertex appears on the queue
- Every vertex whose `edgeTo[]` and `distTo[]` values change in some pass is processed in the next pass

To complete the implementation, we need to ensure that the algorithm terminates after  $V$  passes. One way to achieve this end is to explicitly keep track of the passes. Our implementation `BellmanFordSP` (ALGORITHM 4.11) uses a different approach that we will consider in detail on page 677: it checks for negative cycles in the subset of digraph edges in `edgeTo[]` and terminates if it finds one.

```
private void relax(EdgeWeightedDigraph G, int v)
{
    for (DirectedEdge e : G.adj(v))
    {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (!onQ[w])
            {
                queue.enqueue(w);
                onQ[w] = true;
            }
        }
        if (cost++ % G.V() == 0)
            findNegativeCycle();
    }
}
```

Relaxation for Bellman-Ford

**Proposition Y.** The queue-based implementation of the Bellman-Ford algorithm solves the single-source shortest-paths problem from a given source  $s$  (or finds a negative cycle reachable from  $s$ ) for any edge-weighted digraph with  $E$  edges and  $V$  vertices, in time proportional to  $EV$  and extra space proportional to  $V$ , in the worst case.

**Proof:** If there is no negative cycle reachable from  $s$ , the algorithm terminates after relaxations corresponding to the  $(V-1)$ st pass of the generic algorithm described in PROPOSITION X (since all shortest paths have fewer than  $V$  edges). If there does exist a negative cycle reachable from  $s$ , the queue never empties (see EXERCISE 4.4.46). If any edge is relaxed during the  $V$ th pass of the generic algorithm described in PROPOSITION X, then the `edgeTo[]` array has a directed cycle and any such cycle is a negative cycle (see EXERCISE 4.4.47). In the worst case, the algorithm mimics the generic algorithm and relaxes all  $E$  edges in each of  $V$  passes.

**ALGORITHM 4.11 Bellman-Ford algorithm (queue-based)**


---

```

public class BellmanFordSP
{
    private double[] distTo;                      // length of path to v
    private DirectedEdge[] edgeTo;                 // last edge on path to v
    private boolean[] onQ;                         // Is this vertex on the queue?
    private Queue<Integer> queue;                // vertices being relaxed
    private int cost;                            // number of calls to relax()
    private Iterable<DirectedEdge> cycle;        // negative cycle in edgeTo[]?

    public BellmanFordSP(EdgeWeightedDigraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        onQ = new boolean[G.V()];
        queue = new Queue<Integer>();
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        queue.enqueue(s);
        onQ[s] = true;
        while (!queue.isEmpty() && !hasNegativeCycle())
        {
            int v = queue.dequeue();
            onQ[v] = false;
            relax(G, v);
        }
    }

    private void relax(EdgeWeightedDigraph G, int v)
    // See page 673.

    public double distTo(int v)                  // standard client query methods
    public boolean hasPathTo(int v)             // for SPT implementations
    public Iterable<Edge> pathTo(int v)         // (See page 649.)

    private void findNegativeCycle()
    public boolean hasNegativeCycle()
    public Iterable<DirectedEdge> negativeCycle()
    // See page 677.
}

```

---

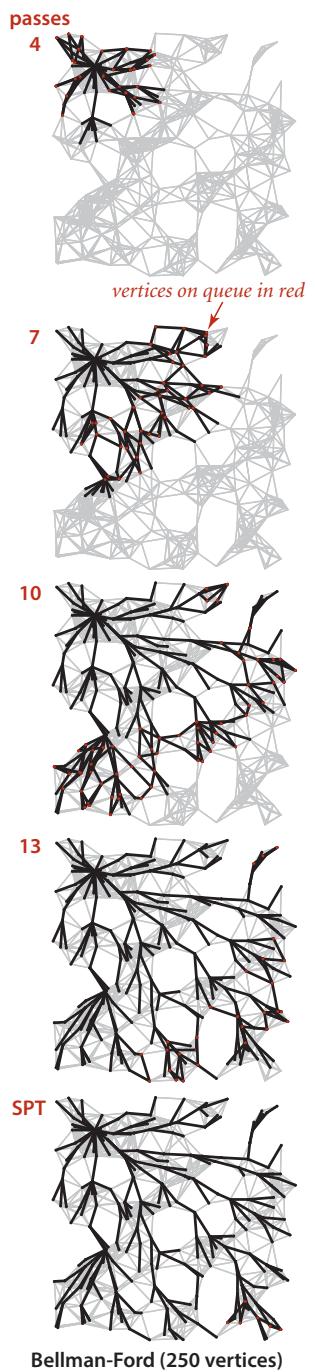
This implementation of the Bellman-Ford algorithm uses a version of `relax()` that puts vertices pointed to by edges that successfully relax on a FIFO queue (avoiding duplicates) and periodically checks for a negative cycle in `edgeTo[]` (see text).

The queue-based Bellman-Ford algorithm is an effective and efficient method for solving the shortest-paths problem that is widely used in practice, even for the case when edge weights are positive. For example, as shown in the diagram at right, our 250-vertex example is complete in 14 passes and requires fewer path-length compares than Dijkstra's algorithm for the same problem.

**Negative weights.** The example on the next page traces the progress of the Bellman-Ford algorithm in a digraph with negative weights. We start with the source  $s$  on queue and then compute the SPT as follows:

- Relax  $0 \rightarrow 2$  and  $0 \rightarrow 4$  and put 2 and 4 on the queue.
- Relax  $2 \rightarrow 7$  and put 7 on the queue. Then relax  $4 \rightarrow 5$  and put 5 on the queue. Then relax  $4 \rightarrow 7$ , which is ineligible.
- Relax  $7 \rightarrow 3$  and  $5 \rightarrow 1$  and put 3 and 1 on the queue. Then relax  $5 \rightarrow 4$  and  $5 \rightarrow 7$ , which are ineligible.
- Relax  $3 \rightarrow 6$  and put 6 on the queue. Then relax  $1 \rightarrow 3$ , which is ineligible.
- Relax  $6 \rightarrow 4$  and put 4 on the queue. This negative-weight edge gives a shorter path to 4, so its edges must be relaxed again (they were first relaxed in pass 2). The distances to 5 and to 1 are no longer valid but will be corrected in later passes.
- Relax  $4 \rightarrow 5$  and put 5 on the queue. Then relax  $4 \rightarrow 7$ , which is still ineligible.
- Relax  $5 \rightarrow 1$  and put 1 on the queue. Then relax  $5 \rightarrow 4$  and  $5 \rightarrow 7$ , which are both still ineligible.
- Relax  $1 \rightarrow 3$ , which is still ineligible, leaving the queue empty.

The shortest-paths tree for this example is a single long path from 0 to 1. The edges from 4, 5, and 1 are all relaxed twice for this example. Rereading the proof of PROPOSITION X in the context of this example is a good way to better understand it.



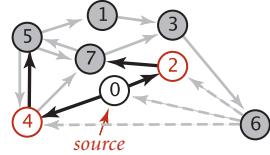
## tinyEDDn.txt

```

4->5 0.35
5->4 0.35
4->7 0.37
5->7 0.28
7->5 0.28
5->1 0.32
0->4 0.38
0->2 0.26
7->3 0.39
1->3 0.29
2->7 0.34
6->2 -1.20
3->6 0.52
6->0 -1.40
6->4 -1.25

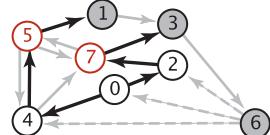
```

queue  
2  
4  
7  
5



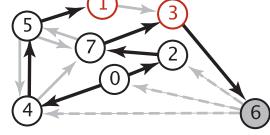
edgeTo[] distTo[]

0		
1		
2	0->2	0.26
3		
4	0->4	0.38
5	4->5	0.73
6		
7	2->7	0.60

7  
5  
3  
1

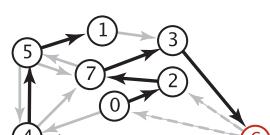
edgeTo[] distTo[]

0		
1	5->1	1.05
2	0->2	0.26
3	7->3	0.99
4	0->4	0.38
5	4->5	0.73
6	2->7	0.60
7		

3  
1  
6

edgeTo[] distTo[]

0		
1	5->1	1.05
2	0->2	0.26
3	7->3	0.99
4	0->4	0.38
5	4->5	0.73
6	3->6	1.51
7	2->7	0.60

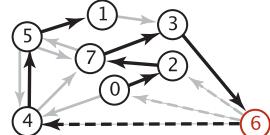
6  
4

edgeTo[] distTo[]

0		
1	5->1	1.05
2	0->2	0.26
3	7->3	0.99
4	6->4	0.26
5	4->5	0.73
6	3->6	1.51
7	2->7	0.60

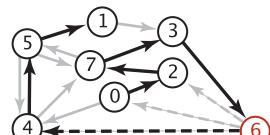
edgeTo[] distTo[]

0		
1	5->1	1.05
2	0->2	0.26
3	7->3	0.99
4	6->4	0.26
5	4->5	0.61
6	3->6	1.51
7	2->7	0.60

4  
5

edgeTo[] distTo[]

0		
1	5->1	0.93
2	0->2	0.26
3	7->3	0.99
4	6->4	0.26
5	4->5	0.61
6	3->6	1.51
7	2->7	0.60

5  
1

Trace of the Bellman-Ford algorithm (negative weights)

**Negative cycle detection.** Our implementation BellmanFordSP checks for negative cycles to avoid an infinite loop. We can apply the code that does this check to provide clients with the capability to check for and extract negative cycles, as well. We do so by adding the following methods to the SP API on page 644:

<pre>boolean hasNegativeCycle() Iterable&lt;DirectedEdge&gt; negativeCycle()</pre>	<i>has a negative cycle?</i> <i>a negative cycle</i> <i>(null if no negative cycles)</i>
--	--

Shortest -paths API extensions for handling negative cycles

Implementing these methods is not difficult, as shown in the code below. After running the constructor in BellmanFordSP, the proof of PROPOSITION Y tells us that the digraph has a negative cycle reachable from the source if and only if the queue is nonempty after the  $V$ th pass through all the edges. Moreover, the subgraph of edges in our `edgeTo[]` array must contain a negative cycle. Accordingly, to implement `negativeCycle()` we build an edge-weighted digraph from the edges in `edgeTo[]` and look for a cycle in that digraph. To find the cycle, we use a version of `DirectedCycle` from SECTION 4.2, adapted to work for edge-weighted digraphs (see EXERCISE 4.4.12). We amortize the cost of this check by

- Adding an instance variable `cycle` and a private method `findNegativeCycle()` that sets `cycle` to an iterator for the edges of a negative cycle if one is found (and to `null` if none is found)
- Calling `findNegativeCycle()` after every  $V$  edge relaxations

This approach ensures that the loop in the constructor terminates. Moreover, clients can call `hasNegativeCycle()` to learn whether there is a negative cycle reachable from the source and `negativeCycle()` to get one such cycle. Adding the capability to detect any negative cycle in the digraph is also a simple extension (see EXERCISE 4.4.43).

```
private void findNegativeCycle()
{
    int V = edgeTo.length;
    EdgeWeightedDigraph spt;
    spt = new EdgeWeightedDigraph(V);
    for (int v = 0; v < V; v++)
        if (edgeTo[v] != null)
            spt.addEdge(edgeTo[v]);

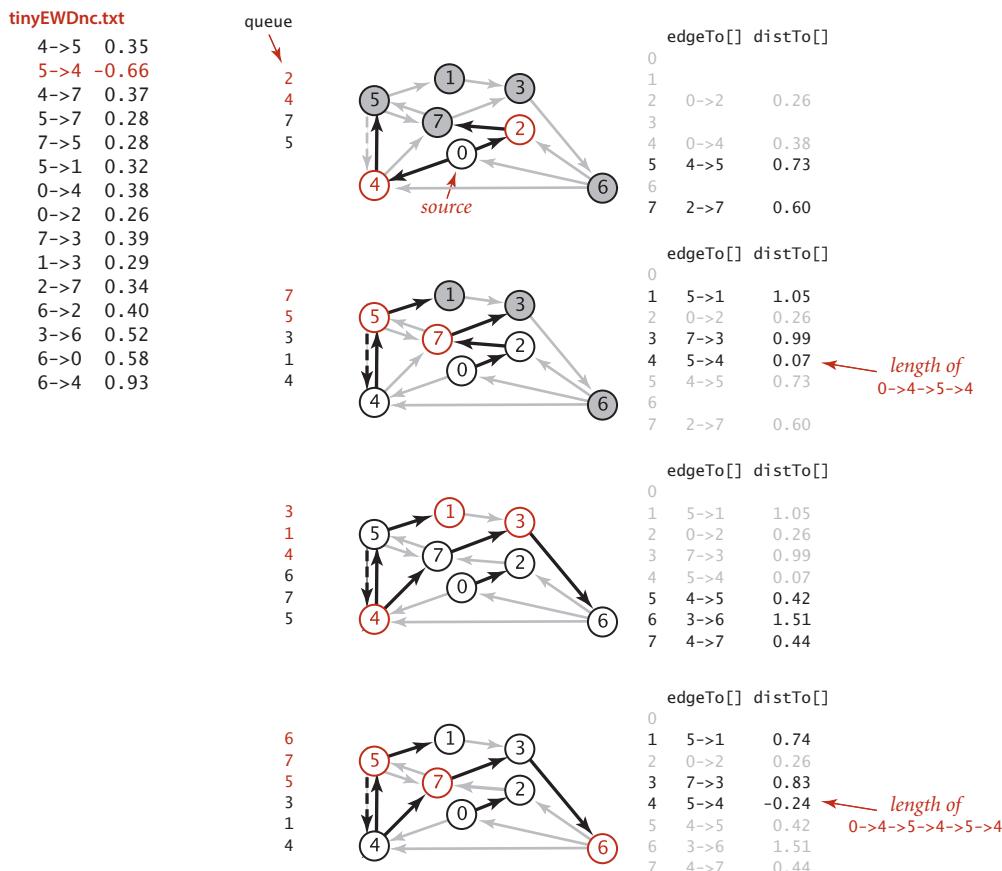
    EdgeWeightedCycleFinder cf;
    cf = new EdgeWeightedCycleFinder(spt);
    cycle = cf.cycle();
}

public boolean hasNegativeCycle()
{   return cycle != null; }

public Iterable<DirectedEdge> negativeCycle()
{   return cycle; }
```

Negative cycle detection methods for Bellman-Ford algorithm

The example below traces the progress of the Bellman-Ford algorithm in a digraph with a negative cycle. Passes 0 (not shown) and 1 are the same as for `tinyEWDn.txt`. In pass 2, after relaxing  $7 \rightarrow 3$  and  $5 \rightarrow 1$  and putting 3 and 1 on queue, it relaxes the negative-weight edge  $5 \rightarrow 4$ . This relaxation sets `edgeTo[4]` to  $5 \rightarrow 4$ , which cuts off vertex 4 from the source 0 in `edgeTo[]`, thereby *creating a cycle*  $4 \rightarrow 5 \rightarrow 4$ . From that point on, the algorithm spins through the cycle, lowering the distances to all the vertices touched, until finishing when the cycle is detected, with the queue not empty. The cycle is in the `edgeTo[]` array, for discovery by `findNegativeCycle()`. Using the cycle detection strategy described on the previous page, the algorithm terminates when vertex 6 is relaxed during pass 4.



Trace of the Bellman-Ford algorithm (negative cycle)

**Arbitrage.** Consider a market for financial transactions that is based on trading commodities. You can find a familiar example in tables that show conversion rates among currencies, such as the one in our sample file `rates.txt` shown here. The first line in the file is the number  $V$  of currencies;

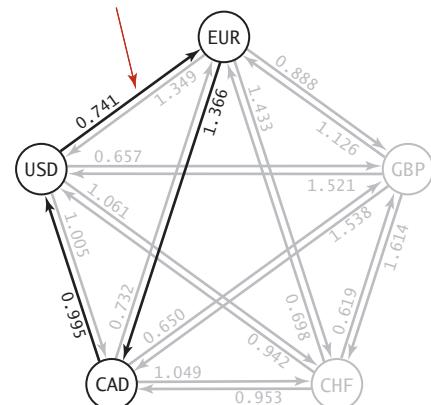
then the file has one line per currency, giving its name followed by the conversion rates to the other currencies. For brevity, this example includes just five of the hundreds of currencies that are traded on modern markets: U.S. dollars (USD), Euros (EUR), British pounds (GBP), Swiss francs (CHF), and Canadian dollars (CAD). The  $t$ th number on line  $s$  represents a conversion rate: the number of units of the currency named on row  $t$  that can be bought with 1 unit of the currency named on row  $s$ . For example, our table says that 1,000 U.S. dollars will buy 741 euros. This table is equivalent to a *complete edge-weighted digraph* with a vertex corresponding to each currency and an edge corresponding to each conversion rate. An edge  $s \rightarrow t$  with weight  $x$  corresponds to a conversion from  $s$  to  $t$  at exchange rate  $x$ . Paths in the di-

graph specify multistep conversions. For example, combining the conversion just mentioned with an edge  $t \rightarrow u$  with weight  $y$  gives a path  $s \rightarrow t \rightarrow u$  that represents a way to convert 1 unit of currency  $s$  into  $xy$  units of currency  $u$ . For example, we might buy  $1,012.206 = 741 \times 1.366$  Canadian dollars with our euros. Note that this gives a better rate than directly converting from U.S. dollars to Canadian dollars. You might expect  $xy$  to be equal to the weight of  $s \rightarrow u$  in all such cases, but such tables represent a complex financial system where such consistency cannot be guaranteed. Thus, finding the path from  $s$  to  $u$  such that the product of the weights is maximal is certainly of interest. Even more interesting is a case where the product of the edge weights in a directed cycle is greater than 1. In our example, suppose that the weight of  $u \rightarrow s$  is  $z$  and  $xyz > 1$ . Then cycle  $s \rightarrow t \rightarrow u \rightarrow s$  gives a way to convert 1 unit of currency  $s$  into more than 1 unit ( $xyz$ ) of currency  $s$ . In other words, we can make a  $100(xyz - 1)$  percent profit by converting from  $s$  to  $t$  to  $u$  back to  $s$ . For example, if we convert our 1,012.206 Canadian dollars back to US dollars, we get  $1,012.206 \times .995 = 1,007.14497$  dollars, a 7.14497-dollar profit. That might not seem like much, but a

% more rates.txt

	5	1	0.741	0.657	1.061	1.005
USD	1	1.349	1	0.888	1.433	1.366
EUR	1.349	1	1.126	1	1.614	1.538
GBP	1.521	1.126	1			
CHF	0.942	0.698	0.619	1		0.953
CAD	0.995	0.732	0.650	1.049	1	

$$0.741 * 1.366 * .995 = 1.00714497$$



An arbitrage opportunity

## Arbitrage in currency exchange

```
public class Arbitrage
{
    public static void main(String[] args)
    {
        int V = StdIn.readInt();
        String[] name = new String[V];
        EdgeWeightedDigraph G = new EdgeWeightedDigraph(V);
        for (int v = 0; v < V; v++)
        {
            name[v] = StdIn.readString();
            for (int w = 0; w < V; w++)
            {
                double rate = StdIn.readDouble();
                DirectedEdge e = new DirectedEdge(v, w, -Math.log(rate));
                G.addEdge(e);
            }
        }
        BellmanFordSP spt = new BellmanFordSP(G, 0);
        if (spt.hasNegativeCycle())
        {
            double stake = 1000.0;
            for (DirectedEdge e : spt.negativeCycle())
            {
                StdOut.printf("%10.5f %s ", stake, name[e.from()]);
                stake *= Math.exp(-e.weight());
                StdOut.printf("= %10.5f %s\n", stake, name[e.to()]);
            }
        }
        else StdOut.println("No arbitrage opportunity");
    }
}
```

This `BellmanFordSP` client finds an arbitrage opportunity in a currency exchange table by constructing a complete-graph representation of the exchange table and then using the Bellman-Ford algorithm to find a negative cycle in the graph.

```
% java Arbitrage < rates.txt
1000.00000 USD = 741.00000 EUR
741.00000 EUR = 1012.20600 CAD
1012.20600 CAD = 1007.14497 USD
```

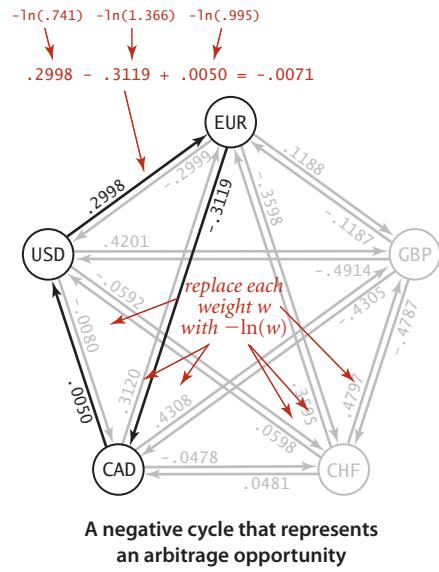
currency trader might have 1 million dollars and be able to execute these transactions every minute, which would lead to profits of over \$7,000 per minute, or \$420,000 per hour! This situation is an example of an *arbitrage* opportunity that would allow traders to make unlimited profits were it not for forces outside the model, such as transaction fees or limitations on the size of transactions. Even with these forces, arbitrage is plenty profitable in the real world. What does this problem have to do with shortest paths? The answer to this question is remarkably simple:

**Proposition Z.** The arbitrage problem is a negative-cycle-detection problem in edge-weighted digraphs.

**Proof:** Replace each weight by its *logarithm*, negated. With this change, computing path weights by multiplying edge weights in the original problem corresponds to adding them in the transformed problem. Specifically, any product  $w_1 w_2 \dots w_k$  corresponds to a sum  $-\ln(w_1) - \ln(w_2) - \dots - \ln(w_k)$ . The transformed edge weights might be negative or positive, a path from  $v$  to  $w$  gives a way of converting from currency  $v$  to currency  $w$ , and any negative cycle is an arbitrage opportunity.

In our example, where all transactions are possible, the digraph is a complete graph, so any negative cycle is reachable from any vertex. In general commodity exchanges, some edges may be absent, so the one-argument constructor described in EXERCISE 4.4.43 is needed. No efficient algorithm for finding the *best* arbitrage opportunity (the most negative cycle in a digraph) is known (and the graph does not have to be very big for this computational burden to be overwhelming), but the fastest algorithm to find *any* arbitrage opportunity is crucial—a trader with that algorithm is likely to systematically wipe out numerous opportunities before the second-fastest algorithm finds any.

THE TRANSFORMATION IN THE PROOF OF PROPOSITION Z is useful even in the absence of arbitrage, because it reduces currency conversion to a shortest-paths problem. Since the logarithm function is monotonic (and we negated the logarithms), the product is maximized precisely when the sum is minimized. The edge weights might be negative or positive, and a shortest path from  $v$  to  $w$  gives a best way of converting from currency  $v$  to currency  $w$ .



**Perspective** The table below summarizes the important characteristics of the shortest-paths algorithms that we have considered in this section. The first reason to choose among the algorithms has to do with basic properties of the digraph at hand. Does it have negative weights? Does it have cycles? Does it have negative cycles? Beyond these basic characteristics, the characteristics of edge-weighted digraphs can vary widely, so choosing among the algorithms requires some experimentation when more than one can apply.

algorithm	restriction	path length compares (order of growth)		extra space	sweet spot
		typical	worst case		
Dijkstra (eager)	positive edge weights	$E \log V$	$E \log V$	$V$	worst-case guarantee
topological sort	edge-weighted DAGs	$E + V$	$E + V$	$V$	optimal for acyclic
Bellman-Ford (queue-based)	no negative cycles	$E + V$	$VE$	$V$	widely applicable

**Performance characteristics of shortest-paths algorithms**

**Historical notes.** Shortest-paths problems have been intensively studied and widely used since the 1950s. The history of Dijkstra's algorithm for computing shortest paths is similar (and related) to the history of Prim's algorithm for computing the MST. The name *Dijkstra's algorithm* is commonly used to refer both to the abstract method of building an SPT by adding vertices in order of their distance from the source and to its implementation as the optimal algorithm for the adjacency-matrix representation, because E. W. Dijkstra presented both in his 1959 paper (and also showed that the same approach could compute the MST). Performance improvements for sparse graphs are dependent on later improvements in priority-queue implementations that are not specific to the shortest-paths problem. Improved performance of Dijkstra's algorithm is one of the most important applications of that technology (for example, with a data structure known as a *Fibonacci heap*, the worst-case bound can be reduced to  $E + V \log V$ ). The Bellman-Ford algorithm has proven to be useful in practice and has found wide application, particularly for general edge-weighted digraphs. While the running time of the Bellman-Ford algorithm is likely to be linear for typical applications, its worst-case running time is  $VE$ . The development of a worst-case linear-time shortest-paths algorithm for sparse graphs remains an open problem. The basic Bellman-Ford algorithm

was developed in the 1950s by L. Ford and R. Bellman; despite the dramatic strides in performance that we have seen for many other graph problems, we have not yet seen algorithms with better worst-case performance for digraphs with negative edge weights (but no negative cycles).

**Q&A**

**Q.** Why define separate data types for undirected graphs, directed graphs, edge-weighted undirected graphs, and edge-weighted digraphs?

**A.** We do so both for clarity in client code and for simpler and more efficient implementation code in unweighted graphs. In applications or systems where all types of graphs are to be processed, it is a textbook exercise in software engineering to define an ADT from which ADTs can be derived for `Graph`, the unweighted undirected graphs of SECTION 4.1; `Digraph`, the unweighted digraphs of SECTION 4.2; `EdgeWeightedGraph`, the edge-weighted undirected graphs of SECTION 4.3; or `EdgeWeightedDigraph`, the edge-weighted directed graphs of this section.

**Q.** How can we find shortest paths in undirected (edge-weighted) graphs?

**A.** For positive edge weights, Dijkstra's algorithm does the job. We just build an `EdgeWeightedDigraph` corresponding to the given `EdgeWeightedGraph` (by adding two directed edges corresponding to each undirected edge, one in each direction) and then run Dijkstra's algorithm. If edge weights can be negative, efficient algorithms are available, but they are more complicated than the Bellman-Ford algorithm.

## EXERCISES

**4.4.1** True or false: Adding a constant to every edge weight does not change the solution to the single-source shortest-paths problem.

**4.4.2** Provide implementations of the constructor `EdgeWeightedDigraph(In in)` and the method `toString()` for `EdgeWeightedDigraph`.

**4.4.3** Develop an implementation of `EdgeWeightedDigraph` for dense graphs that uses an adjacency-matrix (two-dimensional array of weights) representation (see EXERCISE 4.3.9). Ignore parallel edges.

**4.4.4** Draw the (unique) SPT for source 0 of the edge-weighted digraph obtained by deleting vertex 7 from `tinyEWD.txt` (see page 644), and give the parent-link representation of the SPT. Answer the question for the same graph with all edge reversed.

**4.4.5** Change the direction of edge  $0 \rightarrow 2$  in `tinyEWD.txt` (see page 644). Draw two different SPTs that are rooted at 2 for this modified edge-weighted digraph.

**4.4.6** Give a trace that shows the process of computing the SPT of the digraph defined in EXERCISE 4.4.5 with the eager version of Dijkstra's algorithm.

**4.4.7** Develop a version of `DijkstraSP` that supports a client method that returns a *second* shortest path from  $s$  to  $t$  in an edge-weighted digraph (and returns `null` if there is only one shortest path from  $s$  to  $t$ ).

**4.4.8** The *diameter* of a digraph is the length of the maximum-length shortest path connecting two vertices. Write a `DijkstraSP` client that finds the diameter of a given `EdgeWeightedDigraph` that has nonnegative weights.

**4.4.9** The table below, from an old published road map, purports to give the length of the shortest routes connecting the cities. It contains an error. Correct the table. Also, add a table that shows how to achieve the shortest routes.

	Providence	Westerly	New London	Norwich
Providence	-	53	54	48
Westerly	53	-	18	101
New London	54	18	-	12
Norwich	48	101	12	-

**EXERCISES (continued)**

**4.4.10** Consider the edges in the digraph defined in EXERCISE 4.4.4 to be undirected edges such that each edge corresponds to equal-weight edges in both directions in the edge-weighted digraph. Answer EXERCISE 4.4.6 for this corresponding edge-weighted digraph.

**4.4.11** Use the memory-cost model of SECTION 1.4 to determine the amount of memory used by EdgeWeightedDigraph to represent a graph with  $V$  vertices and  $E$  edges.

**4.4.12** Adapt the DirectedCycle and Topological classes from SECTION 4.2 to use the EdgeWeightedDigraph and DirectedEdge APIs of this section, thus implementing EdgeWeightedDirectedCycle and Topological classes.

**4.4.13** Show, in the style of the trace in the text, the process of computing the SPT with Dijkstra's algorithm for the digraph obtained by removing the edge  $5 \rightarrow 7$  from tinyEWD.txt (see page 644).

**4.4.14** Show the paths that would be discovered by the two strawman approaches described on page 668 for the example tinyEWdn.txt shown on that page.

**4.4.15** What happens to Bellman-Ford if there is a negative cycle on the path from  $s$  to  $v$  and then you call pathTo( $v$ )?

**4.4.16** Suppose that we convert an EdgeWeightedGraph into an EdgeWeightedDigraph by creating two DirectedEdge objects in the EdgeWeightedDigraph (one in each direction) for each Edge in the EdgeWeightedGraph (as described for Dijkstra's algorithm in the Q&A onpage 684) and then use the Bellman-Ford algorithm. Explain why this approach fails spectacularly.

**4.4.17** What happens if you allow a vertex to be enqueued more than once in the same pass in the Bellman-Ford algorithm?

*Answer:* The running time of the algorithm can go exponential. For example, consider what happens for the complete edge-weighted digraph whose edge weights are all  $-1$ .

**4.4.18** Write a CPM client that prints all critical paths.

**4.4.19** Find the lowest-weight cycle (best arbitrage opportunity) in the example shown in the text.

**4.4.20** Find a currency-conversion table online or in a newspaper. Use it to build an arbitrage table. *Note:* Avoid tables that are derived (calculated) from a few values and that therefore do not give sufficiently accurate conversion information to be interesting. *Extra credit:* Make a killing in the money-exchange market!

**4.4.21** Show, in the style of the trace in the text, the process of computing the SPT with the Bellman-Ford algorithm for the edge-weighted digraph of EXERCISE 4.4.5.

## CREATIVE PROBLEMS

**4.4.22** *Vertex weights.* Show that shortest-paths computations in digraphs with non-negative weights on vertices (where the weight of a path is defined to be the sum of the weights of the vertices) can be handled by building an edge-weighted digraph that has weights on only the edges.

**4.4.23** *Source-sink shortest paths.* Develop an API and implementation that use a version of Dijkstra's algorithm to solve the *source-sink* shortest path problem on edge-weighted digraphs.

**4.4.24** *Multisource shortest paths.* Develop an API and implementation that uses Dijkstra's algorithm to solve the *multisource* shortest-paths problem on edge-weighted digraphs with positive edge weights: given a *set* of sources, find a shortest-paths forest that enables implementation of a method that returns to clients the shortest path from any source to each vertex. *Hint:* Add a dummy vertex with a zero-weight edge to each source, or initialize the priority queue with all sources, with their `distTo[]` entries set to 0.

**4.4.25** *Shortest path between two subsets.* Given a digraph with positive edge weights, and two distinguished subsets of vertices  $S$  and  $T$ , find a shortest path from any vertex in  $S$  to any vertex in  $T$ . Your algorithm should run in time proportional to  $E \log V$ , in the worst case.

**4.4.26** *Single-source shortest paths in dense graphs.* Develop a version of Dijkstra's algorithm that can find the SPT from a given vertex in a dense edge-weighted digraph in time proportional to  $V^2$ . Use an adjacency-matrix representation (see EXERCISE 4.4.3 and EXERCISE 4.3.29).

**4.4.27** *Shortest paths in Euclidean graphs.* Adapt our APIs to speed up Dijkstra's algorithm in the case where it is known that vertices are points in the plane.

**4.4.28** *Longest paths in DAGs.* Develop an implementation `AcyclicLP` that can solve the *longest-paths* problem in edge-weighted DAGs, as described in PROPOSITION T.

**4.4.29** *General optimality.* Complete the proof of PROPOSITION W by showing that if there exists a directed path from  $s$  to  $v$  and no vertex on any path from  $s$  to  $v$  is on a negative cycle, then there exists a shortest path from  $s$  to  $v$ . (*Hint:* See PROPOSITION P.)

**4.4.30** *All-pairs shortest paths in digraphs without negative cycles.* Articulate an API like the one implemented on page 656 for the all-pairs shortest-paths problem in graphs

with no negative cycles. Develop an implementation that runs a version of Bellman-Ford to identify real-valued weights  $\pi[v]$  such that for any edge  $v \rightarrow w$ , the edge weight plus the difference between  $\pi[v]$  and  $\pi[w]$  is nonnegative. Then use these weights to reweight the graph, so that Dijkstra's algorithm is effective for finding all shortest paths in the reweighted graph.

**4.4.31 All-pairs shortest paths on a line.** Given a weighted line graph (undirected connected graph, all vertices of degree 2, except two endpoints which have degree 1), devise an algorithm that preprocesses the graph in linear time and can return the distance of the shortest path between any two vertices in constant time.

**4.4.32 Parent-checking heuristic.** Modify Bellman-Ford to visit a vertex  $v$  only if its SPT parent  $\text{edgeTo}[v]$  is not currently on the queue. This heuristic has been reported by Cherkassky, Goldberg, and Radzik to be useful in practice. Prove that it correctly computes shortest paths and that the worst-case running time is proportional to  $EV$ .

**4.4.33 Shortest path in a grid.** Given an  $N$ -by- $N$  matrix of positive integers, find the shortest path from the  $(0, 0)$  entry to the  $(N-1, N-1)$  entry, where the length of the path is the sum of the integers in the path. Repeat the problem but assume you can only move right and down.

**4.4.34 Monotonic shortest path.** Given an edge-weighted digraph, find a *monotonic* shortest path from  $s$  to every other vertex. A path is monotonic if the weight of its edges are either strictly increasing or strictly decreasing. *Hint:* Relax edges in ascending order and find a best path; then relax edges in descending order and find a best path.

**4.4.35 Bitonic shortest path.** Given an edge-weighted digraph, find a *bitonic* shortest path from  $s$  to every other vertex (if one exists). A path is bitonic if there is an intermediate vertex  $v$  such that the weights of the edges on the path from  $s$  to  $v$  are strictly increasing and the weights of the edges on the path from  $v$  to  $t$  are strictly decreasing. The path should be simple (no repeated vertices).

**4.4.36 Neighbors.** Develop an SP client that finds all vertices within a given distance  $d$  of a given vertex in a given edge-weighted digraph. The running time of your method should be proportional to the number of vertices and edges in the subgraph induced by those vertices and the edges incident to them, plus  $V$  (to initialize data structures).

**CREATIVE PROBLEMS (continued)**

**4.4.37 Critical edges.** Develop an algorithm for finding an edge whose removal causes maximal increase in the shortest-paths length from one given vertex to another given vertex in a given edge-weighted digraph.

**4.4.38 Sensitivity.** Develop an SP client that performs a sensitivity analysis on the edge-weighted digraph's edges with respect to a given pair of vertices  $s$  and  $t$ : Compute a  $V$ -by- $V$  boolean matrix such that, for every  $v$  and  $w$ , the entry in row  $v$  and column  $w$  is `true` if  $v \rightarrow w$  is an edge whose weight can be increased without the shortest-path length from  $v$  to  $w$  being increased and is `false` otherwise.

**4.4.39 Lazy implementation of Dijkstra's algorithm.** Develop an implementation of the lazy version of Dijkstra's algorithm that is described in the text.

**4.4.40 Bottleneck SPT.** Show that an MST of an undirected graph is equivalent to a bottleneck SPT of the graph: For every pair of vertices  $v$  and  $w$ , it gives the path connecting them whose longest edge is as short as possible.

**4.4.41 Bidirectional search.** Develop a class for the source-sink shortest-paths problem that is based on code like ALGORITHM 4.9 but that initializes the priority queue with both the source and the sink. Doing so leads to the growth of an SPT from each vertex; your main task is to decide precisely what to do when the two SPTs collide.

**4.4.42 Worst case (Dijkstra).** Describe a family of graphs with  $V$  vertices and  $E$  edges for which the worst-case running time of Dijkstra's algorithm is achieved.

**4.4.43 Negative cycle detection.** Suppose that we add a constructor to ALGORITHM 4.11 that differs from the constructor given only in that it omits the second argument and that it initializes all `distTo[]` entries to 0. Show that, if a client uses that constructor, a client call to `hasNegativeCycle()` returns `true` if and only if the graph has a negative cycle (and `negativeCycle()` returns that cycle).

*Answer:* Consider a digraph formed from the original by adding a new source with an edge of weight 0 to all the other vertices. After one pass, all `distTo[]` entries are 0, and finding a negative cycle reachable from that source is the same as finding a negative cycle anywhere in the original graph.

**4.4.44 Worst case (Bellman-Ford).** Describe a family of graphs for which ALGORITHM 4.11 takes time proportional to  $VE$ .

**4.4.45** *Fast Bellman-Ford.* Develop an algorithm that breaks the linearithmic running time barrier for the single-source shortest-paths problem in general edge-weighted digraphs for the special case where the weights are integers known to be bounded in absolute value by a constant.

**4.4.46** *Bellman-Ford queue never empties.* Show that if there is a negative cycle reachable from the source in the queue-based implementation of the Bellman-Ford algorithm, then the queue never empties.

**4.4.47** *Bellman-Ford negative cycle detection.* Show that if any edge is relaxed during the  $V$ th pass of the generic Bellman-Ford algorithm, then the `edgeTo[]` array has a directed cycle and any such cycle is a negative cycle.

**4.4.48** *Animate.* Write a client program that does dynamic graphical animations of Dijkstra's algorithm.

## EXPERIMENTS

**4.4.49** *Random sparse edge-weighted digraphs.* Modify your solution to EXERCISE 4.3.34 to assign a random direction to each edge.

**4.4.50** *Random Euclidean edge-weighted digraphs.* Modify your solution to EXERCISE 4.3.35 to assign a random direction to each edge.

**4.4.51** *Random grid edge-weighted digraphs.* Modify your solution to EXERCISE 4.3.36 to assign a random direction to each edge.

**4.4.52** *Negative weights I.* Modify your random edge-weighted digraph generators to generate weights between  $x$  and  $y$  (where  $x$  and  $y$  are both between  $-1$  and  $1$ ) by rescaling.

**4.4.53** *Negative weights II.* Modify your random edge-weighted digraph generators to generate negative weights by negating a fixed percentage (whose value is supplied by the client) of the edge weights.

**4.4.54** *Negative weights III.* Develop client programs that use your edge-weighted digraph generator to produce edge-weighted digraphs that have a large percentage of negative weights but have at most a few negative cycles, for as large a range of values of  $V$  and  $E$  as possible.

*Testing all algorithms and studying all parameters against all edge-weighted digraph models is unrealistic. For each problem listed below, write a client that addresses the problem for any given input digraph, then choose among the generators above to run experiments for that graph model. Use your judgment in selecting experiments, perhaps in response to results of previous experiments. Write a narrative explaining your results and any conclusions that might be drawn.*

**4.4.55** *Prediction.* Estimate, to within a factor of 10, the largest graph with  $E = 10V$  that your computer and programming system could handle if you were to use Dijkstra's algorithm to compute all its shortest paths in 10 seconds.

**4.4.56** *Cost of laziness.* Run empirical studies to compare the performance of the lazy version of Dijkstra's algorithm with the eager version, for various edge-weighted digraph models.

**4.4.57** *Johnson's algorithm.* Develop a priority-queue implementation that uses a  $d$ -way heap. Find the best value of  $d$  for various edge-weighted digraph models.

**4.4.58** *Arbitrage model.* Develop a model for generating random arbitrage problems. Your goal is to generate tables that are as similar as possible to the tables that you used in EXERCISE 4.4.20.

**4.4.59** *Parallel job-scheduling-with-deadlines model.* Develop a model for generating random instances of the parallel job-scheduling-with-deadlines problem. Your goal is to generate nontrivial problems that are likely to be feasible.

FIVE



# Strings

<b>5.1</b>	String Sorts . . . . .	702
<b>5.2</b>	Tries . . . . .	730
<b>5.3</b>	Substring Search . . . . .	758
<b>5.4</b>	Regular Expressions . . . . .	788
<b>5.5</b>	Data Compression . . . . .	810

We communicate by exchanging strings of characters. Accordingly, numerous important and familiar applications are based on processing strings. In this chapter, we consider classic algorithms for addressing the underlying computational challenges surrounding applications such as the following:

**Information processing.** When you search for web pages containing a given keyword, you are using a string-processing application. In the modern world, virtually *all* information is encoded as a sequence of strings, and the applications that process it are string-processing applications of crucial importance.

**Genomics.** Computational biologists work with a *genetic code* that reduces DNA to (very long) strings formed from four characters (A, C, T, and G). Vast databases giving codes describing all manner of living organisms have been developed in recent years, so that string processing is a cornerstone of modern research in computational biology.

**Communications systems.** When you send a text message or an email or download an ebook, you are transmitting a string from one place to another. Applications that process strings for this purpose were an original motivation for the development of string-processing algorithms.

**Programming systems.** Programs are strings. Compilers, interpreters, and other applications that convert programs into machine instructions are critical applications that use sophisticated string-processing techniques. Indeed, all written languages are expressed as strings, and another motivation for the development of string-processing algorithms was the theory of formal languages, the study of describing sets of strings.

This list of a few significant examples illustrates the diversity and importance of string-processing algorithms.

The plan of this chapter is as follows: After addressing basic properties of strings, we revisit in **SECTIONS 5.1 AND 5.2** the sorting and searching APIs from **CHAPTERS 2** and **3**. Algorithms that exploit special properties of string keys are faster and more flexible than the algorithms that we considered earlier. In **SECTION 5.3** we consider algorithms for *substring search*, including a famous algorithm due to Knuth, Morris, and Pratt. In **SECTION 5.4** we introduce *regular expressions*, the basis of the *pattern-matching* problem, a generalization of substring search, and a quintessential search tool known as *grep*. These classic algorithms are based on the related conceptual devices known as *formal languages* and *finite automata*. **SECTION 5.5** is devoted to a central application: *data compression*, where we try to reduce the size of a string as much as possible.

**Rules of the game** For clarity and efficiency, our implementations are expressed in terms of the Java `String` class, but we intentionally use as few operations as possible from that class to make it easier to adapt our algorithms for use on other string-like types of data and to other programming languages. We introduced strings in detail in **SECTION 1.2** but briefly review here their most important characteristics.

**Characters.** A `String` is a sequence of characters. Characters are of type `char` and can have one of  $2^{16}$  possible values. For many decades, programmers restricted attention to characters encoded in 7-bit ASCII (see page 815 for a conversion table) or 8-bit extended ASCII, but many modern applications call for 16-bit Unicode.

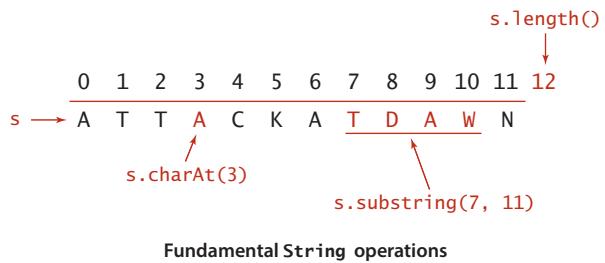
**Immutability.** `String` objects are immutable, so that we can use them in assignment statements and as arguments and return values from methods without having to worry about their values changing.

**Indexing.** The operation that we perform most often is *extract a specified character from a string* that the `charAt()` method in Java's `String` class provides. We expect `charAt()` to complete its work in *constant* time, as if the string were stored in a `char[]` array. As discussed in **CHAPTER 1**, this expectation is quite reasonable.

**Length.** In Java, the *find the length of a string* operation is implemented in the `length()` method in `String`. Again, we expect `length()` to complete its work in *constant* time, and again, this expectation is reasonable, although some care is needed in some programming environments.

**Substring.** Java's `substring()` method implements the *extract a specified substring* operation. *Its running time depends on the underlying representation.* It takes *constant* time and space in typical Java 6 (and earlier) implementations but *linear* time and space in typical Java 7 implementations (see page 202).

**Concatenation.** In Java, the *create a new string formed by appending one string to another* operation is a built-in operation (using the + operator) that takes time proportional to the length of the result. For example, we avoid forming a string by appending one character at a time because that is a *quadratic* process in Java. (Java has a `StringBuilder` class for that use.)



Fundamental String operations

**Character arrays.** The Java `String` is decidedly not a primitive type. The standard implementation provides the operations just described to facilitate client programming. By contrast, many of the algorithms that we consider can work with a low-level representation such as an array of `char` values, and many clients might prefer such a representation, because it consumes less space and takes less time. For several of the algorithms that we consider, the cost of converting from one representation to the other would be higher than the cost of running the algorithm. As indicated in the table below, the differences in code that processes the two representations are minor (`substring()` is more complicated and is omitted), so use of one representation or the other is no barrier to understanding the algorithm.

UNDERSTANDING THE EFFICIENCY OF THESE OPERATIONS is a key ingredient in understanding the efficiency of several string-processing algorithms. Not all programming languages provide `String` implementations with these performance characteristics. For example, determining the length of a string take time proportional to the number of characters in the string in the widely used C programming language. Adapting the algorithms that we describe to such languages is always possible (implement an ADT like Java's `String`), but also might present different challenges and opportunities.

operation	array of characters	Java string
<i>declare</i>	<code>char[] a</code>	<code>String s</code>
<i>indexed character access</i>	<code>a[i]</code>	<code>s.charAt(i)</code>
<i>length</i>	<code>a.length</code>	<code>s.length()</code>
<i>convert</i>	<code>a = s.toCharArray();</code>	<code>s = new String(a);</code>

Two ways to represent strings in Java

We primarily use the `String` data type in the text, with liberal use of indexing and length and occasional use of substring extraction and concatenation. When appropriate, we also provide on the booksite the corresponding code for `char` arrays. In performance-critical applications, the primary consideration in choosing between the two for clients is often the cost of accessing a character (`a[i]` is likely to be much faster than `s.charAt(i)` in typical Java implementations).

**Alphabets** Some applications involve strings taken from a restricted alphabet. In such applications, it often makes sense to use an `Alphabet` class with the following API:

---

<code>public class Alphabet</code>	
<code>Alphabet(String s)</code>	<i>create a new alphabet from chars in s</i>
<code>char toChar(int index)</code>	<i>convert index to corresponding alphabet char</i>
<code>int toIndex(char c)</code>	<i>convert c to an index between 0 and R-1</i>
<code>boolean contains(char c)</code>	<i>is c in the alphabet?</i>
<code>int R()</code>	<i>radix (number of characters in alphabet)</i>
<code>int lgR()</code>	<i>number of bits to represent an index</i>
<code>int[] toIndices(String s)</code>	<i>convert s to base-R integer</i>
<code>String toChars(int[] indices)</code>	<i>convert base-R integer to string over this alphabet</i>
<b>Alphabet API</b>	

---

This API is based on a constructor that takes as argument an  $R$ -character string that specifies the alphabet and the `toChar()` and `toIndex()` methods for converting (in constant time) between string characters and `int` values between 0 and  $R-1$ . It also includes a `contains()` method for checking whether a given character is in the alphabet, the methods `R()` and `lgR()` for finding the number of characters in the alphabet and the number of bits needed to represent them, and the methods `toIndices()` and `toChars()` for converting between strings of characters in the alphabet and `int` arrays. For convenience, we also include the built-in alphabets in the table at the top of the next page, which you can access with code such as `Alphabet.UNICODE16`. Implementing `Alphabet` is a straightforward exercise (see EXERCISE 5.1.12). We will examine a sample client on page 699.

**Character-indexed arrays.** One of the most important reasons to use `Alphabet` is that many algorithms gain efficiency through the use of character-indexed arrays, where we associate information with each character that we can retrieve with a single array

name	R()	TgR()	characters
BINARY	2	1	01
DNA	4	2	ACTG
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
PROTEIN	20	5	ACDEFGHIJKLMNOPQRSTUVWXYZ
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	<i>ASCII characters</i>
EXTENDED_ASCII	256	8	<i>extended ASCII characters</i>
UNICODE16	65536	16	<i>Unicode characters</i>

**Standard alphabets**

```
public class Count
{
    public static void main(String[] args)
    {
        Alphabet alpha = new Alphabet(args[0]);
        int R = alpha.R();
        int[] count = new int[R];

        String s = StdIn.readAll();
        int N = s.length();
        for (int i = 0; i < N; i++)
            if (alpha.contains(s.charAt(i)))
                count[alpha.toIntIndex(s.charAt(i))]++;

        for (int c = 0; c < R; c++)
            StdOut.println(alpha.toChar(c)
                           + " " + count[c]);
    }
}
```

Typical Alphabet client

```
% more abra.txt
ABRACADABRA!
```

```
% java Count ABCDR < abra.txt
A 5
B 2
C 1
D 1
R 2
```

access. With a Java `String`, we have to use an array of size 65,536; with `Alphabet`, we just need an array with one entry for each alphabet character. Some of the algorithms that we consider can produce huge numbers of such arrays, and in such cases, the space for arrays of size 65,536 can be prohibitive. As an example, consider the class `Count` at the bottom of the previous page, which takes a string of characters from the command line and prints a table of the frequency of occurrence of those characters that appear on standard input. The `count[]` array that holds the frequencies in `Count` is an example of a character-indexed array. This calculation may seem to you to be a bit frivolous; actually, it is the basis for a family of fast sorting methods that we will consider in SECTION 5.1.

**Numbers.** As you can see from several of the standard `Alphabet` examples, we often represent numbers as strings. The method `toIndices()` converts any `String` over a given `Alphabet` into a base- $R$  number represented as an `int[]` array with all values between 0 and  $R-1$ . In some situations, doing this conversion at the start leads to compact code, because any digit can be used as an index in a character-indexed array. For example, if we know that the input consists only of characters from the alphabet, we could replace the inner loop in `Count` with the more compact code

```
int[] a = alpha.toIndices(s);
for (int i = 0; i < N; i++)
    count[a[i]]++;
```

In this context, we refer to  $R$  as the *radix*, the base of the number system. Several of the algorithms that we consider are often referred to as “radix” methods because they work with one digit at a time.

```
% more pi.txt
3141592653
5897932384
6264338327
9502884197
... [100,000 digits of pi]

% java Count 0123456789 < pi.txt
0 9999
1 10137
2 9908
3 10026
4 9971
5 10026
6 10028
7 10025
8 9978
9 9902
```

DESPITE THE ADVANTAGES of using a data type such as `Alphabet` in string-processing algorithms (particularly for small alphabets), we do not develop our implementations in the book for strings taken from a general `Alphabet` because

- The preponderance of clients just use `String`
- Conversion to and from indices tends to fall in the inner loop and slow down implementations considerably
- The code is more complicated, and therefore more difficult to understand

Accordingly we use `String`, use the constant `R = 256` in the code and `R` as a parameter in the analysis, and discuss performance for general alphabets when appropriate. You can find full `Alphabet`-based implementations on the booksite.



## 5.1 STRING SORTS

FOR MANY SORTING APPLICATIONS, the keys that define the order are strings. In this section, we look at methods that take advantage of special properties of strings to develop sorts for string keys that are more efficient than the general-purpose sorts that we considered in CHAPTER 2.

We consider two fundamentally different approaches to string sorting. Both of them are venerable methods that have served programmers well for many decades.

The first approach examines the characters in the keys in a right-to-left order. Such methods are generally referred to as least-significant-digit (LSD) string sorts. Use of the term *digit* instead of *character* traces back to the application of the same basic method to numbers of various types. Thinking of a string as a base-256 number, considering characters from right to left amounts to considering first the least significant digits. This approach is the method of choice for string-sorting applications where all the keys are the same length.

The second approach examines the characters in the keys in a left-to-right order, working with the most significant character first. These methods are generally referred to as most-significant-digit (MSD) string sorts—we will consider two such methods in this section. MSD string sorts are attractive because they can get a sorting job done without necessarily examining all of the input characters. MSD string sorts are similar to quicksort, because they partition the array to be sorted into independent pieces such that the sort is completed by recursively applying the same method to the subarrays. The difference is that MSD string sorts use just the first character of the sort key to do the partitioning, while quicksort uses comparisons that could involve examining the whole key. The first method that we consider creates a partition for each character value; the second always creates three partitions, for sort keys whose first character is less than, equal to, or greater than the partitioning key's first character.

The number of characters in the alphabet is an important parameter when analyzing string sorts. Though we focus on extended ASCII strings ( $R = 256$ ), we will also consider strings taken from much smaller alphabets (such as genomic sequences) and from much larger alphabets (such as the 65,536-character Unicode alphabet that is an international standard for encoding natural languages).

**Key-indexed counting** As a warmup, we consider a simple method for sorting that is effective whenever the keys are small integers. This method, known as *key-indexed counting*, is useful in its own right and is also the basis for two of the three string sorts that we consider in this section.

Consider the following data-processing problem, which might be faced by a teacher maintaining grades for a class with students assigned to sections, which are numbered 1, 2, 3, and so forth. On some occasions, it is necessary to have the class listed by section. Since the section numbers are small integers, sorting by key-indexed counting is appropriate. To describe the method, we assume that the information is kept in an array  $a[]$  of items that each contain a name and a section number, that section numbers are integers between 0 and  $R-1$ , and that

```
for (i = 0; i < N; i++)
    count[a[i].key() + 1]++;
    count[]
```

*always 0*

Anderson	2	0 0 0 1 0 0
Brown	3	0 0 0 1 1 0
Davis	3	0 0 0 1 2 0
Garcia	4	0 0 0 1 2 1
Harris	1	0 0 1 1 2 1
Jackson	3	0 0 1 1 3 1
Johnson	4	0 0 1 1 3 2
Jones	3	0 0 1 1 4 2
Martin	1	0 0 2 1 4 2
Martinez	2	0 0 2 2 4 2
Miller	2	0 0 2 3 4 2
Moore	1	0 0 3 3 4 2
Robinson	2	0 0 3 4 4 2
Smith	4	0 0 3 4 4 3
Taylor	3	0 0 3 4 5 3
Thomas	4	0 0 3 4 5 4
Thompson	4	0 0 3 4 5 5
White	2	0 0 3 5 5 5
Williams	3	0 0 3 5 6 5
Wilson	4	0 0 3 5 6 6

*number of 3s*

Computing frequency counts

the code  $a[i].key()$  returns the section number for the indicated student. The method breaks down into four steps, which we describe in turn.

input name	section	sorted result (by section)
Anderson	2	Harris
Brown	3	Martin
Davis	3	Moore
Garcia	4	Anderson
Harris	1	Martinez
Jackson	3	Miller
Johnson	4	Robinson
Jones	3	White
Martin	1	Brown
Martinez	2	Davis
Miller	2	Jackson
Moore	1	Jones
Robinson	2	Taylor
Smith	4	Williams
Taylor	3	Garcia
Thomas	4	Johnson
Thompson	4	Smith
White	2	Thomas
Williams	3	Thompson
Wilson	4	Wilson

↑  
*keys are  
small integers*

Typical candidate for key-indexed counting

**Compute frequency counts.** The first step is to count the frequency of occurrence of each key value, using an int array  $count[]$ . For each item, we use the key to access an entry in  $count[]$  and increment that entry. If the key value is  $r$ , we increment  $count[r+1]$ . (Why +1? The reason for that will become clear in the next step.) In the example at left, we first increment  $count[3]$  because Anderson is in section 2, then we increment  $count[4]$  twice because Brown and Davis are in section 3, and so forth. Note that  $count[0]$  is always 0, and that  $count[1]$  is 0 in this example (no students are in section 0).

**Transform counts to indices.** Next, we use `count[]` to compute, for each key value, the starting index positions in the sorted order of items with that key. In our example, since there are three items with key 1 and five items with key 2, then the items with key 3 start at position 8 in the sorted array. In general, to get the starting index for items with any given key value we sum the frequency counts of smaller values. For each key value  $r$ , the sum of the counts for key values less than  $r+1$  is equal to the sum of the counts for key values less than  $r$  plus `count[r]`, so it is easy to proceed from left to right to transform `count[]` into an index table that we can use to sort the data.

```
for (int i = 0; i < N; i++)
    aux[count[a[i].key()]++] = a[i];
```

count[]				
i	1	2	3	4
0	0	3	8	14
1	0	4	8	14
2	0	4	9	14
3	0	4	10	14
4	0	4	10	15
5	1	4	10	15
6	1	4	11	15
7	1	4	11	16
8	1	4	12	16
9	2	4	12	16
10	2	5	12	16
11	2	6	12	16
12	3	6	12	16
13	3	7	12	16
14	3	7	12	17
15	3	7	13	17
16	3	7	13	18
17	3	7	13	19
18	3	8	13	19
19	3	8	14	19
	3	8	14	20
3	8	14	20	

i	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]	a[16]	a[17]	a[18]	a[19]
0	Anderson	Brown	Davis	Garcia	Harris	Jackson	Johnson	Jones	Martin	Martinez	Miller	Robinson	White	Brown	Davis	Jackson	Jones	Taylor	Williams	
1	Harris	Martin	Moore	Anderson	Martinez	Miller	Robinson	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
2	Martin	Moore	Anderson	Martinez	Miller	Robinson	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
3	Moore	Anderson	Martinez	Miller	Robinson	White	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
4	Anderson	Martinez	Miller	Robinson	White	White	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
5	Martinez	Miller	Robinson	White	White	White	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
6	Miller	Robinson	White	White	White	White	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
7	Robinson	White	White	White	White	White	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White	
8	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
9	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
10	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
11	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
12	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
13	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
14	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
15	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
16	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
17	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
18	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							
19	White	White	White	Robinson	Miller	Anderson	Moore	Anderson	White	White	White	White	White							

Distributing the data (records with key 3 highlighted)

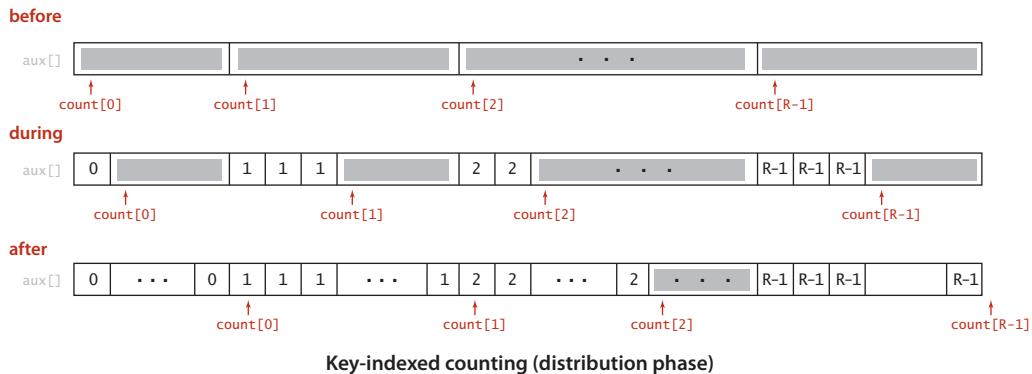
```
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
```

always 0	count[]					
r	0	1	2	3	4	5
0	0	0	3	5	6	6
1	0	0	3	5	6	6
2	0	0	3	5	6	6
3	0	0	3	8	6	6
4	0	0	3	8	14	6
5	0	0	3	8	14	20
0	0	3	8	14	20	

number of keys less than 3  
(start index of 3s in output)

Transforming counts to start indices

**Distribute the data.** With the `count[]` array transformed into an index table, we accomplish the actual sort by moving the items to an auxiliary array `aux[]`. We move each item to the position in `aux[]` indicated by the `count[]` entry corresponding to its key, and then increment that entry to maintain the following invariant for `count[]`: for each key value  $r$ , `count[r]` is the index of the position in `aux[]` where the next item with key value  $r$  (if any) should be placed. This process produces a sorted result with one pass through the data, as illustrated at left. Note: In one of our applications, the fact that this implementation is *stable* is critical: items with equal keys are brought together but kept in the same relative order.



**Copy back.** Since we accomplished the sort by moving the items to an auxiliary array, the last step is to copy the sorted result back to the original array.

**Proposition A.** Key-indexed counting uses  $11N + 4R + 1$  array accesses to stably sort  $N$  items whose keys are integers between 0 and  $R - 1$ .

**Proof:** Immediate from the code. Initializing the arrays uses  $N + R + 1$  array accesses. The first loop increments a counter for each of the  $N$  items ( $3N$  array accesses); the second loop does  $R$  additions ( $3R$  array accesses); the third loop does  $N$  counter increments and  $N$  data moves ( $5N$  array accesses); and the fourth loop does  $N$  data moves ( $2N$  array accesses). Both moves preserve the relative order of equal keys.

KEY-INDEXED COUNTING is an extremely effective and often overlooked sorting method for applications where keys are small integers. Understanding how it works is a first step toward understanding string sorting. PROPOSITION A implies that key-indexed counting breaks through the  $N \log N$  lower bound that we proved for sorting. How does it manage to do so? PROPOSITION I in SECTION 2.2 is a lower bound on the number of *compares* needed (when data is accessed only through `compareTo()`)—key-indexed counting does *no* compares (it accesses data only through `key()`). When  $R$  is within a constant factor of  $N$ , we have a linear-time sort.

```

int N = a.length;
[] aux = new String[N];
int[] count = new int[R+1];
// Compute frequency counts.
for (int i = 0; i < N; i++)
    count[a[i].key() + 1]++;
// Transform counts to indices.
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
// Distribute the records.
for (int i = 0; i < N; i++)
    aux[count[a[i].key()]++] = a[i];
// Copy back.
for (int i = 0; i < N; i++)
    a[i] = aux[i];

```

Key-indexed counting (`a[i].key()` is an int in  $[0, R]$ ).

**LSD string sort** The first string-sorting method that we consider is known as *least-significant-digit first* (LSD) string sort. Consider the following motivating application: Suppose that a highway engineer sets up a device that records the license plate numbers of all vehicles using a busy highway for a given period of time and wants to know the number of *different* vehicles that used the highway. As you know from SECTION 2.1, one easy way to solve this problem is to sort the numbers, then make a pass through to count the different values, as in Dedup (page 490). License plates are a mixture of numbers and letters, so it is natural to represent them as strings. In the simplest situation (such as the California license plate examples at right) the strings all have the same number of characters. This situation is often found in sort applications—for example, telephone numbers, bank account numbers, and IP addresses are typically fixed-length strings.

Sorting such strings can be done with key-indexed counting, as shown in ALGORITHM 5.1 (LSD) and the example below it on the facing page. If the strings are each of length  $W$ , we sort the strings  $W$  times with key-indexed counting, using each of the positions as the key, proceeding from right to left. It is not easy, at first, to be convinced that the method produces a sorted array—in fact, it does not work at all unless the key-indexed count implementation is stable. Keep this fact in mind and refer to the example when studying this proof of correctness:

input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CI0720	10HV845
1ICK750	10HV845
10HV845	10HV845
4JZY524	2IYE230
1ICK750	2RLA629
3CI0720	2RLA629
10HV845	3ATW723
10HV845	3CI0720
2RLA629	3CI0720
2RLA629	4JZY524
3ATW723	4PGC938

↑  
keys are all  
the same length

Typical candidate for  
LSD string sort

**Proposition B.** LSD string sort stably sorts fixed-length strings.

**Proof:** This fact depends crucially on the key-indexed counting implementation being *stable*, as indicated in PROPOSITION A. After sorting keys on their  $i$  trailing characters (in a stable manner), we know that any two keys appear in proper order in the array (considering just those characters) either because the first of their  $i$  trailing characters is different, in which case the sort on that character puts them in order, or because the first of their  $i$  trailing characters is the same, in which case they are in order because of stability (and by induction, for  $i-1$ ).

Another way to state the proof is to think about the future: if the characters that have not been examined for a pair of keys are identical, any difference between the keys is restricted to the characters already examined, so the keys have been properly ordered and will remain so because of stability. If, on the other hand, the characters that have not

---

**ALGORITHM 5.1 LSD string sort**


---

```

public class LSD
{
    public static void sort(String[] a, int W)
    { // Sort a[] on leading W characters.
        int N = a.length;
        int R = 256;
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--)
        { // Sort by key-indexed counting on dth char.

            int[] count = new int[R+1]; // Compute frequency counts.
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++) // Transform counts to indices.
                count[r+1] += count[r];
            for (int i = 0; i < N; i++) // Distribute.
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++) // Copy back.
                a[i] = aux[i];
        }
    }
}

```

---

To sort an array  $a[]$  of strings that each have exactly  $W$  characters, we do  $W$  key-indexed counting sorts: one for each character position, proceeding from right to left.

<b>input (<math>W=7</math>)</b>	<b><math>d=6</math></b>	<b><math>d=5</math></b>	<b><math>d=4</math></b>	<b><math>d=3</math></b>	<b><math>d=2</math></b>	<b><math>d=1</math></b>	<b><math>d=0</math></b>	<b>output</b>
4PGC938	2IYE230	3CI0720	2IYE230	2RLA629	1ICK750	3ATW723	1ICK750	1ICK750
2IYE230	3CI0720	3CI0720	4JZY524	2RLA629	1ICK750	3CI0720	1ICK750	1ICK750
3CI0720	1ICK750	3ATW723	2RLA629	4PGC938	4PGC938	3CI0720	10HV845	10HV845
1ICK750	1ICK750	4JZY524	2RLA629	2IYE230	10HV845	1ICK750	10HV845	10HV845
10HV845	3CI0720	2RLA629	3CI0720	1ICK750	10HV845	1ICK750	10HV845	10HV845
4JZY524	3ATW723	2RLA629	3CI0720	1ICK750	10HV845	2IYE230	2IYE230	2IYE230
1ICK750	4JZY524	2IYE230	3ATW723	3CI0720	3CI0720	4JZY524	2RLA629	2RLA629
3CI0720	10HV845	4PGC938	1ICK750	3CI0720	3CI0720	10HV845	2RLA629	2RLA629
10HV845	10HV845	10HV845	1ICK750	10HV845	2RLA629	10HV845	3ATW723	3ATW723
10HV845	10HV845	10HV845	10HV845	10HV845	2RLA629	10HV845	3CI0720	3CI0720
2RLA629	4PGC938	10HV845	10HV845	10HV845	3ATW723	4PGC938	3CI0720	3CI0720
2RLA629	2RLA629	1ICK750	10HV845	3ATW723	2IYE230	2RLA629	4JZY524	4JZY524
3ATW723	2RLA629	1ICK750	4PGC938	4JZY524	4JZY524	2RLA629	4PGC938	4PGC938

♣ J	♦ A	♠ A
♥ 6	♥ A	♠ 2
♦ A	♣ A	♠ 3
♥ A	♠ A	♠ 4
♠ K	♠ 2	♠ 5
♥ J	♣ 2	♠ 6
♦ Q	♥ 2	♠ 7
♣ 6	♦ 2	♠ 8
♠ J	♥ 3	♠ 9
♣ A	♠ 3	♠ 10
♦ 9	♣ 3	♠ J
♥ 9	♦ 3	♠ Q
♦ 8	♦ 4	♠ K
♠ 9	♣ 4	♥ A
♣ K	♥ 4	♥ 2
♦ 4	♠ 4	♥ 3
♠ 5	♠ 5	♥ 4
♣ Q	♦ 5	♥ 5
♥ 3	♣ 5	♥ 6
♠ 2	♥ 5	♥ 7
♣ 10	♥ 6	♥ 8
♣ 9	♣ 6	♥ 9
♥ 7	♠ 6	♥ 10
♣ 4	♦ 6	♥ J
♥ 4	♥ 7	♥ Q
♦ 10	♣ 7	♥ K
♠ A	♠ 7	♦ A
♦ 5	♦ 7	♦ 2
♠ 3	♦ 8	♦ 3
♥ 8	♥ 8	♦ 4
♣ 2	♠ 8	♦ 5
♦ K	♣ 8	♦ 6
♠ 4	♦ 9	♦ 7
♣ 7	♥ 9	♦ 8
♥ Q	♦ 9	♦ 9
♦ J	♣ 9	♦ 10
♠ 6	♣ 10	♦ J
♣ 3	♦ 10	♦ Q
♠ 7	♠ 10	♦ K
♠ 8	♥ 10	♣ A
♣ 10	♣ J	♣ 2
♦ 3	♥ J	♣ 3
♥ 10	♣ J	♣ 4
♦ 7	♦ J	♣ 5
♠ Q	♦ Q	♣ 6
♥ 2	♣ Q	♣ 7
♦ 2	♥ Q	♣ 8
♣ 5	♣ Q	♣ 9
♥ K	♣ K	♣ 10
♥ 5	♣ K	♣ J
♦ 6	♦ K	♣ Q
♣ 8	♥ K	♣ K

been examined are different, the characters already examined do not matter, and a later pass will correctly order the pair based on the more significant differences.

LSD radix sorting is the method used by the old punched-card-sorting machines that were developed at the beginning of the 20th century and thus predated the use of computers in commercial data processing by several decades. Such machines had the capability of distributing a deck of punched cards among 10 bins, according to the pattern of holes punched in the selected columns. If a deck of cards had numbers punched in a particular set of columns, an operator could sort the cards by running them through the machine on the rightmost digit, then picking up and stacking the output decks in order, then running them through the machine on the next-to-rightmost digit, and so forth, until getting to the first digit. The physical stacking of the cards is a stable process, which is mimicked by key-indexed counting sort. Not only was this version of LSD radix sorting important in commercial applications up through the 1970s, but it was also used by many cautious programmers (and students!), who would have to keep their programs on punched cards (one line per card) and would punch sequence numbers in the final few columns of a program deck so as to be able to put the deck back in order mechanically if it were accidentally dropped. This method is also a neat way to sort a deck of playing cards: deal them into thirteen piles (one for each value), pick up the piles in order, then deal into four piles (one for each suit). The (stable) dealing process keeps the cards in order within each suit, so picking up the piles in suit order yields a sorted deck.

In many string-sorting applications (even license plates, for some states), the keys are not all be the same length. It is possible to adapt LSD string sort to work for such applications, but we leave this task for exercises because we will next consider two other methods that are specifically designed for variable-length keys.

From a theoretical standpoint, LSD string sort is significant because it is a *linear-time* sort for typical applications. No matter how large the value of  $N$ , it makes  $W$  passes through the data. Specifically:

**Proposition B (continued).** LSD string sort uses  $\sim 7WN + 3WR$  array accesses and extra space proportional to  $N + R$  to sort  $N$  items whose keys are  $W$ -character strings taken from an  $R$ -character alphabet.

**Proof:** The method is  $W$  passes of key-indexed counting, except that the `aux[]` array is initialized just once. The total is immediate from the code and PROPOSITION A.

For typical applications,  $R$  is far smaller than  $N$ , so PROPOSITION B implies that the total running time is proportional to  $WN$ . An input array of  $N$  strings that each have  $W$  characters has a total of  $WN$  characters, so the running time of LSD string sort is *linear* in the size of the input.

♣ J	♠ K	♦ A
♥ 6	♠ J	♦ 2
♦ A	♠ 9	♦ 3
♥ A	♠ 5	♦ 4
♠ K	♠ 2	♦ 5
♥ J	♠ A	♦ 6
♦ Q	♠ 3	♦ 7
♣ 6	♠ 4	♦ K
♠ J	♠ 6	♦ 8
♣ A	♠ 7	♦ 10
♦ 9	♠ 8	♦ J
♥ 9	♠ 10	♦ Q
♦ 8	♠ Q	♦ K
♠ 9	♥ 6	♥ A
♣ K	♥ A	♥ 2
♦ 4	♥ J	♥ 3
♠ 5	♥ 9	♥ 4
♣ Q	♥ 3	♥ 5
♥ 3	♥ 7	♥ 6
♠ 2	♥ 4	♥ 7
♣ 10	♥ 8	♥ 8
♣ 9	♥ Q	♥ 9
♥ 7	♥ 10	♥ 10
♣ 4	♥ 2	♥ J
♥ 4	♥ K	♥ Q
♦ 10	♥ 5	♥ K
♠ A	♦ A	♦ A
♦ 5	♦ Q	♦ 2
♠ 3	♦ 9	♦ 3
♥ 8	♦ 8	♦ 4
♣ 2	♦ 4	♦ 5
♦ K	♦ 10	♦ 6
♠ 4	♦ 5	♦ 7
♣ 7	♦ K	♦ 8
♥ Q	♦ J	♦ 9
♦ J	♦ 3	♦ 10
♠ 6	♦ 7	♦ J
♣ 3	♦ 2	♦ Q
♠ 7	♦ 6	♦ K
♣ 8	♣ J	♣ A
♣ 10	♣ 6	♣ 2
♦ 3	♣ A	♣ 3
♥ 10	♣ K	♣ 4
♦ 7	♣ Q	♣ 5
♠ Q	♣ 10	♣ 6
♥ 2	♣ 9	♣ 7
♦ 2	♣ 4	♣ 8
♣ 5	♣ 2	♣ 9
♥ K	♣ 7	♣ 10
♥ 5	♣ 3	♣ J
♦ 6	♣ 5	♣ Q
♣ 8	♣ 8	♣ K

Sorting a card deck with  
MSD string sort

**MSD string sort** To implement a general-purpose string sort, where strings are not necessarily all the same length, we consider the characters in left-to-right order. We know that strings that start with a should appear before strings that start with b, and so forth. The natural way to implement this idea is a recursive method known as *most-significant-digit-first* (MSD) string sort. We use key-indexed counting to sort the strings according to their first character, then (recursively) sort the subarrays corresponding to each character (excluding the first character, which we know to be the same for each string in each subarray). Like quicksort, MSD string sort partitions the array into subarrays that can be sorted independently to complete the job, but it partitions the array into one subarray for each possible value of the first character, instead of the two or three partitions in quicksort.

**End-of-string convention.** We need to pay particular attention to reaching the ends of strings in MSD string sort. For a proper sort, we need the subarray for strings whose characters have all been examined to appear as the first subarray, and we do not want to recursively sort this subarray. To facilitate these two parts of the computation we use a private two-argument `charAt()` method to convert from an indexed string character to an array index that returns -1 if the specified character position is past the end of the string. This convention means that we have  $R+1$  different possible character values at each string position: -1 to signify *end of string*, 0 for the first alphabet character, 1 for the second alphabet character, and so forth. Then, we just add 1 to each returned value, to get a nonnegative `int` that we can use to index `count[]`. Since

*sort on first character value  
to partition into subarrays*      *recursively sort subarrays  
(excluding first character)*



Overview of MSD string sort

key-indexed counting already needs one extra position, we use the code `int count[] = new int[R+2];` to create the array of frequency counts (and set all of its values to 0). Note: Some languages, notably C and C++, have a built-in end-of-string convention, so our code needs to be adjusted accordingly for such languages.

WITH THESE PREPARATIONS, the implementation of MSD string sort, in ALGORITHM 5.2, requires very little new code. We add a test to cutoff to insertion sort for small subarrays (using a specialized insertion sort that we will consider later), and we add a loop to key-indexed counting to do the recursive calls. As summarized in the table at the bottom of this page, the values in the `count[]` array (after serving to count the frequencies, transform counts to indices, and distribute the data) give us precisely the information that we need to (recursively) sort the subarrays corresponding to each character value.

input	sorted result
she	are
sells	by
seashells	seashells
by	seashells
the	seashore
seashore	sells
the	sells
shells	she
she	she
sells	shells
are	surely
surely	the
seashells	the

Typical candidate for MSD string sort

**Specified alphabet.** The cost of MSD string sort depends strongly on the number of possible characters in the alphabet. It is easy to modify our sort method to take an `Alphabet` as argument, to allow for improved efficiency in clients involving strings taken from relatively small alphabets. The following changes will do the job:

- Save the alphabet in an instance variable `alpha` in the constructor.
- Set `R` to `alpha.R()` in the constructor.
- Replace `s.charAt(d)` with `alpha.toInt(s.charAt(d))` in `charAt()`.

at completion of phase for dth character	value of <code>count[r]</code> is				
	<code>r = 0</code>	<code>r = 1</code>	<code>r between 2 and R-1</code>	<code>r = R</code>	<code>r = R+1</code>
<code>count frequencies</code>	<code>0 (not used)</code>	<code>number of strings of length d</code>	<code>number of strings whose dth character value is r-2</code>		
<code>transform counts to indices</code>	<code>start index of subarray for strings of length d</code>	<code>start index of subarray for strings whose dth character value is r-1</code>		<code>not used</code>	
<code>distribute</code>	<code>start index of subarray for strings whose dth character value is r</code>		<code>not used</code>		
	<code>1 + end index of subarray for strings of length d</code>	<code>1 + end index of subarray for strings whose dth character value is r-1</code>		<code>not used</code>	

Interpretation of `count[]` values during MSD string sort

**ALGORITHM 5.2 MSD string sort**

```
public class MSD
{
    private static int R = 256;           // radix
    private static final int M = 15;       // cutoff for small subarrays
    private static String[] aux;          // auxiliary array for distribution

    private static int charAt(String s, int d)
    { if (d < s.length()) return s.charAt(d); else return -1; }

    public static void sort(String[] a)
    {
        int N = a.length;
        aux = new String[N];
        sort(a, 0, N-1, 0);
    }

    private static void sort(String[] a, int lo, int hi, int d)
    { // Sort from a[lo] to a[hi], starting at the dth character.

        if (hi <= lo + M)
        { Insertion.sort(a, lo, hi, d); return; }

        int[] count = new int[R+2];           // Compute frequency counts.
        for (int i = lo; i <= hi; i++)
            count[charAt(a[i], d) + 2]++;
        for (int r = 0; r < R+1; r++)         // Transform counts to indices.
            count[r+1] += count[r];
        for (int i = lo; i <= hi; i++)        // Distribute.
            aux[count[charAt(a[i], d) + 1]++] = a[i];
        for (int i = lo; i <= hi; i++)        // Copy back.
            a[i] = aux[i - lo];

        // Recursively sort for each character value.
        for (int r = 0; r < R; r++)
            sort(a, lo + count[r], lo + count[r+1] - 1, d+1);
    }
}
```

---

To sort an array `a[]` of strings, we sort them on their first character using key-indexed counting, then (recursively) sort the subarrays corresponding to each first-character value.

In our running examples, we use strings made up of lowercase letters. It is also easy to extend LSD string sort to provide this feature, but typically with much less impact on performance than for MSD string sort.

THE CODE IN ALGORITHM 5.2 is deceptively simple, masking a rather sophisticated computation. It is definitely worth your while to study the trace of the top level at the bottom of this page and the trace of recursive calls on the next page, to be sure that you understand the intricacies of the algorithm. This trace uses a cutoff-for-small-subarrays threshold value ( $M$ ) of 0, so that you can see the sort to completion for this small example. The strings in this example are taken from `Alphabet.LOWERCASE`, with  $R = 26$ ; bear in mind that typical applications might use `Alphabet.EXTENDED_ASCII`, with  $R = 256$ , or `Alphabet.UNICODE16`, with  $R = 65536$ . For large alphabets, MSD string sort is so simple as to be dangerous—improperly used, it can consume outrageous amounts of time and space. Before considering performance characteristics in detail, we shall discuss three important issues (all of which we have considered before, in CHAPTER 2) that must be addressed in any application.

**Small subarrays.** The basic idea behind MSD string sort is quite effective: in typical applications, the strings will be in order after examining only a few characters in the key. Put another way, the method quickly divides the array to be sorted into small

use key-indexed counting on first character			recursively sort subarrays	
count frequencies	transform counts to indices	distribute and copy back	indices at completion of distribute phase	
0 s 0	0 0	0 are	0 0 0	sort(a, 0, 0, 1);
1 e 1	1 a 0	1 by	1 1 1	sort(a, 1, 1, 1);
2 s 2	2 b 1	2 she	2 2 2	sort(a, 2, 1, 1);
3 h 1	3 c 2	3 sells	3 3 2	sort(a, 3, 1, 1);
4 e 0	4 d 2	4 seashells	4 4 2	sort(a, 4, 1, 1);
5 e 0	5 e 2	5 sea	5 5 2	sort(a, 5, 1, 1);
6 f 0	6 f 2	6 shore	6 6 2	sort(a, 6, 1, 1);
7 g 0	7 g 2	7 shells	7 7 2	sort(a, 7, 1, 1);
8 h 0	8 h 2	8 she	8 8 2	sort(a, 8, 1, 1);
9 i 0	9 i 2	9 sells	9 9 2	sort(a, 9, 1, 1);
10 j 0	10 j 2	10 surely	10 10 2	sort(a, 10, 1, 1);
11 k 0	11 k 2	11 seashells	11 11 2	sort(a, 11, 1, 1);
12 l 0	12 l 2	12 the	12 12 2	sort(a, 12, 1, 1);
13 m 0	13 m 2	13 the	13 13 2	sort(a, 13, 1, 1);
14 n 0	14 n 2		14 14 2	sort(a, 14, 1, 1);
15 o 0	15 o 2		15 15 2	sort(a, 15, 1, 1);
16 p 0	16 p 2		16 16 2	sort(a, 16, 1, 1);
17 q 0	17 q 2		17 17 2	sort(a, 17, 1, 1);
18 r 0	18 r 2		18 18 2	sort(a, 18, 1, 1);
19 s 0	19 s 2		19 19 2	sort(a, 19, 1, 1);
20 t 10	20 t 12		20 20 2	sort(a, 20, 1, 1);
21 u 2	21 u 14		21 21 2	sort(a, 21, 1, 1);
22 v 0	22 v 14		22 22 2	sort(a, 22, 1, 1);
23 w 0	23 w 14		23 23 2	sort(a, 23, 1, 1);
24 x 0	24 x 14		24 24 2	sort(a, 24, 1, 1);
25 y 0	25 y 14		25 25 2	sort(a, 25, 1, 1);
26 z 0	26 z 14		26 26 2	sort(a, 26, 1, 1);
27 0	27 14		27 27 2	sort(a, 27, 1, 1);

start of s subarray  
1 + end of s subarray

Trace of MSD string sort: top level of `sort(a, 0, 13, 0)`

input	are							
she	by	to	by	by	by	by	by	by
sells	she	sells	seashells	sea	sea	sea	seas	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
sea	sea	sells	se1ls	sells	sells	sells	sells	sells
shore	shore	seashells	sell1s	sells	sells	sells	sells	sells
the	shells	she						
shells	she	shore	shore	shore	shore	shore	shells	shells
she	sells	shells	shells	shells	shells	shells	shore	shore
sells	surely	she						
are	seashells	surely						
surely	the	hi	the	the	the	the	the	the
seashells	the							

need to examine every character in equal keys									end of string goes before any char value	output
are	are	are	are	are	are	are	are	are	are	are
by	by	by	by	by	by	by	by	by	by	by
sea	sea	sea	sea	sea	sea	sea	sea	sea	sea	sea
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells	seashells
sells	sells	sells	sells	sells	sells	sells	sells	sells	sells	sells
sells	sells	sells	sells	sells	sells	sells	sells	sells	sells	sells
she	she	she	she	she	she	she	she	she	she	she
shells	shells	shells	shells	shells	shells	shells	shells	shells	shells	shells
she	she	she	she	she	she	she	she	she	she	she
shore	shore	shore	shore	shore	shore	shore	shore	shore	shore	shore
surely	surely	surely	surely	surely	surely	surely	surely	surely	surely	surely
the	the	the	the	the	the	the	the	the	the	the
the	the	the	the	the	the	the	the	the	the	the

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

subarrays. But this is a double-edged sword: we are certain to have to handle huge numbers of tiny subarrays, so we had better be sure that we handle them efficiently. *Small subarrays are of critical importance in the performance of MSD string sort.* We have seen this situation for other recursive sorts (quicksort and mergesort), but it is much more dramatic for MSD string sort. For example, suppose that you are sorting millions of ASCII strings ( $R = 256$ ) that are all different, with no cutoff for small subarrays. Each string eventually finds its way to its own subarray, so you will sort millions of subarrays of size 1. But each such sort involves initializing the 258 entries of the `count[]` array to 0 and transforming them all to indices. This cost is likely to dominate the rest of the sort. With Unicode ( $R = 65536$ ) the sort might be *thousands* of times slower. Indeed, many unsuspecting sort clients have seen their running times explode from minutes to hours on switching from ASCII to Unicode, for precisely this reason. Accordingly, the

```

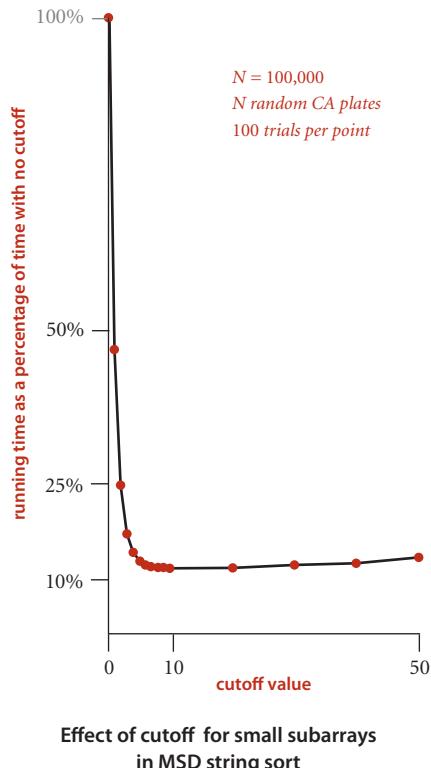
public static void sort(String[] a, int lo, int hi, int d)
{ // Sort from a[lo] to a[hi], starting at the dth character.
    for (int i = lo; i <= hi; i++)
        for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
            exch(a, j, j-1);
}
private static boolean less(String v, String w, int d)
{
    for (int i = d; i < Math.min(v.length(), w.length()); i++)
        if (v.charAt(i) < w.charAt(i)) return true;
        else if (v.charAt(i) > w.charAt(i)) return false;
    return v.length() < w.length();
}

```

Insertion sort for strings whose first  $d$  characters are equal

switch to insertion sort for small subarrays is a *must* for MSD string sort. To avoid the cost of reexamining characters that we know to be equal, we use the version of insertion sort given at the top of the page, which takes an extra argument  $d$  and assumes that the first  $d$  characters of all the strings to be sorted are known to be equal. As with quicksort and mergesort, most of the benefit of this improvement is achieved with a small value of the cutoff, but the savings here are much more dramatic. The diagram at right shows the results of experiments where using a cutoff to insertion sort for subarrays of size 10 or less decreases the running time by a factor of 10 for a typical application.

**Equal keys.** A second pitfall for MSD string sort is that it can be relatively slow for subarrays containing large numbers of equal keys. If a substring occurs sufficiently often that the cutoff for small subarrays does not apply, then a recursive call is needed for every character in all of the equal keys. Moreover, key-indexed counting is an inefficient way to determine that the characters are all equal: not only does each character need to be examined and each string moved, but all the counts have to be initialized, converted to indices, and so forth. Thus, the worst case for MSD string sorting is when all keys are equal. The same problem arises when large numbers of keys have long common prefixes, a situation often found in applications.



**Extra space.** To do the partitioning, MSD uses two auxiliary arrays: the temporary array for distributing keys (`aux[]`) and the array that holds the counts that are transformed into partition indices (`count[]`). The `aux[]` array is of size  $N$  and can be created outside the recursive `sort()` method. This extra space can be eliminated by sacrificing stability (see EXERCISE 5.1.17), but it is often not a major concern in practical applications of MSD string sort. Space for the `count[]` array, on the other hand, can be an important issue (because it *cannot* be created outside the recursive `sort()` method) as addressed in PROPOSITION D below.

**Random string model.** To study the performance of MSD string sort, we use a *random string model*, where each string consists of (independently) random characters, with no bound on their length. Long equal keys are essentially ignored, because they are extremely unlikely. The behavior of MSD string sort in this model is similar to its behavior in a model where we consider random fixed-length keys and also to its performance for typical real data; in all three, MSD string sort tends to examine just a few characters at the beginning of each key, as we will see.

**Performance.** The running time of MSD string sort depends on the data. For compare-based methods, we were primarily concerned with the *order* of the keys; for MSD string sort, the order of the keys is immaterial, but we are concerned with the *values* of the keys.

random (sublinear)	nonrandom with duplicates (nearly linear)	worst case (linear)
1E10402	are	1DNB377
1HYL490	by	1DNB377
1R0Z572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Characters examined by MSD string sort

- For *random* inputs, MSD string sort examines just enough characters to distinguish among the keys, and the running time is *sublinear* in the number of characters in the data (it examines a small fraction of the input characters).
- For *nonrandom* inputs, MSD string sort still could be sublinear but might need to examine more characters than in the random case, depending on the data. In particular, it has to examine all the characters in equal keys, so the running time is nearly linear in the number of characters in the data when significant numbers of equal keys are present.
- In the *worst case*, MSD string sort examines all the characters in the keys, so the running time is *linear* in the number of characters in the data (like LSD string sort). A worst-case input is one with all strings equal.

Some applications involve distinct keys that are well-modeled by the random string model; others have significant numbers of equal keys or long common prefixes, so the sort time is closer to the worst case. Our license-plate-processing application, for example, can fall anywhere between these extremes: if our engineer takes an hour of data from a busy interstate, there will not be many duplicates and the random model will apply; for a week's worth of data on a local road, there will be numerous duplicates and performance will be closer to the worst case.

**Proposition C.** To sort  $N$  random strings from an  $R$ -character alphabet, MSD string sort examines about  $N \log_R N$  characters, on average.

**Proof sketch:** We expect the subarrays to be all about the same size, so the recurrence  $C_N = RC_{N/R} + N$  approximately describes the performance, which leads to the stated result, generalizing our argument for quicksort in CHAPTER 2. Again, this description of the situation is not entirely accurate, because  $N/R$  is not necessarily an integer, and the subarrays are the same size only on the average (and because the number of characters in real keys is finite). These effects turn out to be less significant for MSD string sort than for standard quicksort, so the leading term of the running time is the solution to this recurrence. The detailed analysis that proves this fact is a classical example in the analysis of algorithms, first done by Knuth in the early 1970s.

As food for thought and to indicate why the proof is beyond the scope of this book, note that key length does not play a role. Indeed, the random-string model allows key length to approach infinity. There is a nonzero probability that two keys will match for any specified number of characters, but this probability is so small as to not play a role in our performance estimates.

As we have discussed, the number of characters examined is not the full story for MSD string sort. We also have to take into account the time and space required to count frequencies and turn the counts into indices.

**Proposition D.** MSD string sort uses between  $8N + 3R$  and  $\sim 7wN + 3WR$  array accesses to sort  $N$  strings taken from an  $R$ -character alphabet, where  $w$  is the average string length.

**Proof:** Immediate from the code, PROPOSITION A, and PROPOSITION B. In the best case MSD sort uses just one pass; in the worst case, it performs like LSD string sort.

When  $N$  is small, the factor of  $R$  dominates. Though precise analysis of the total cost becomes difficult and complicated, you can estimate the effect of this cost just by considering small subarrays when keys are distinct. With no cutoff for small subarrays, each key appears in its own subarray, so  $NR$  array accesses are needed for just these subarrays. If we cut off to small subarrays of size  $M$ , we have about  $N/M$  subarrays of size  $M$ , so we are trading off  $NR/M$  array accesses with  $NM/4$  compares, which tells us that we should choose  $M$  to be proportional to the square root of  $R$ .

**Proposition D (continued).** To sort  $N$  strings taken from an  $R$ -character alphabet, the amount of space needed by MSD string sort is proportional to  $R$  times the length of the longest string (plus  $N$ ), in the worst case.

**Proof:** The `count[]` array must be created within `sort()`, so the total amount of space needed is proportional to  $R$  times the depth of recursion (plus  $N$  for the auxiliary array). Precisely, the depth of the recursion is the length of the longest string that is a prefix of two or more of the strings to be sorted.

As just discussed, equal keys cause the depth of the recursion to be proportional to the length of the keys. The immediate practical lesson to be drawn from PROPOSITION D is that it is quite possible for MSD string sort to run out of time or space when sorting long strings taken from large alphabets, particularly if long equal keys are to be expected. For example, with `Alphabet.UNICODE16` and more than  $M$  equal 1,000-character strings, `MSD.sort()` would require space for over 65 million counters!

THE MAIN CHALLENGE in getting maximum efficiency from MSD string sort on keys that are long strings is to deal with lack of randomness in the data. Typically, keys may have long stretches of equal data, or parts of them might fall in only a narrow range. For example, an information-processing application for student data might have keys that include graduation year (4 bytes, but one of four different values), state names (perhaps 10 bytes, but one of 50 different values), and gender (1 byte with one of two given values), as well as a person's name (more similar to random strings, but probably not short, with nonuniform letter distributions, and with trailing blanks in a fixed-length field). Restrictions like these lead to large numbers of empty subarrays during the MSD string sort. Next, we consider a graceful way to adapt to such situations.

**Three-way string quicksort** We can also adapt quicksort to MSD string sorting by using 3-way partitioning on the leading character of the keys, moving to the next character on only the middle subarray (keys with leading character equal to the partitioning character). This method is not difficult to implement, as you can see in ALGORITHM 5.3: we just add an argument to the recursive method in ALGORITHM 2.5 that keeps track of the current character, adapt the 3-way partitioning code to use that character, and appropriately modify the recursive calls.

Although it does the computation in a different order, 3-way string quicksort amounts to sorting the array on the leading characters of the keys (using quicksort), then applying the method recursively on the remainder of the keys. For sorting strings, the method compares favorably with normal quicksort and with MSD string sort. Indeed, it is a hybrid of these two algorithms.

Three-way string quicksort divides the array into only three parts, so it involves more data movement than MSD string sort when the number of nonempty partitions is large because it has to

*use first character value  
to partition into “less,” “equal,”  
and “greater” subarrays*

*recursively sort subarrays  
(excluding first character  
for “equal” subarray)*



Overview of 3-way string quicksort

do a series of 3-way partitions to get the effect of the multiway partition. On the other hand, MSD string sort can create large numbers of (empty) subarrays, whereas 3-way string quicksort always has just three. Thus, 3-way string quicksort adapts well to handling equal keys, keys with long common prefixes, keys that fall into a small range, and small arrays—all situations where MSD string sort runs slowly. Of particular importance is that the

input	sorted result
edu.princeton.cs	com.adobe
com.apple	com.apple
edu.princeton.cs	com.cnn
com.cnn	com.google
com.google	edu.princeton.cs
edu.uva.cs	edu.princeton.cs
edu.princeton.cs	edu.princeton.cs
edu.princeton.cs.www	edu.princeton.cs.www
edu.uva.cs	edu.princeton.ee
edu.uva.cs	edu.uva.cs
edu.uva.cs	edu.uva.cs
edu.uva.cs	edu.uva.cs
com.adobe	edu.uva.cs
edu.princeton.ee	edu.uva.cs

Typical 3-way string quicksort candidate

**ALGORITHM 5.3 Three-way string quicksort**

```
public class Quick3string
{
    private static int charAt(String s, int d)
    { if (d < s.length()) return s.charAt(d); else return -1; }

    public static void sort(String[] a)
    { sort(a, 0, a.length - 1, 0); }

    private static void sort(String[] a, int lo, int hi, int d)
    {
        if (hi <= lo) return;

        int lt = lo, gt = hi;
        int v = charAt(a[lo], d);
        int i = lo + 1;
        while (i <= gt)
        {
            int t = charAt(a[i], d);
            if      (t < v) exch(a, lt++, i++);
            else if (t > v) exch(a, i, gt--);
            else             i++;
        }

        // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi]

        sort(a, lo, lt-1, d);
        if (v >= 0) sort(a, lt, gt, d+1);
        sort(a, gt+1, hi, d);
    }
}
```

To sort an array  $a[]$  of strings, we 3-way partition them on their first character, then (recursively) sort the three resulting subarrays: the strings whose first character is less than the partitioning character, the strings whose first character is equal to the partitioning character (excluding their first character in the sort), and the strings whose first character is greater than the partitioning character.

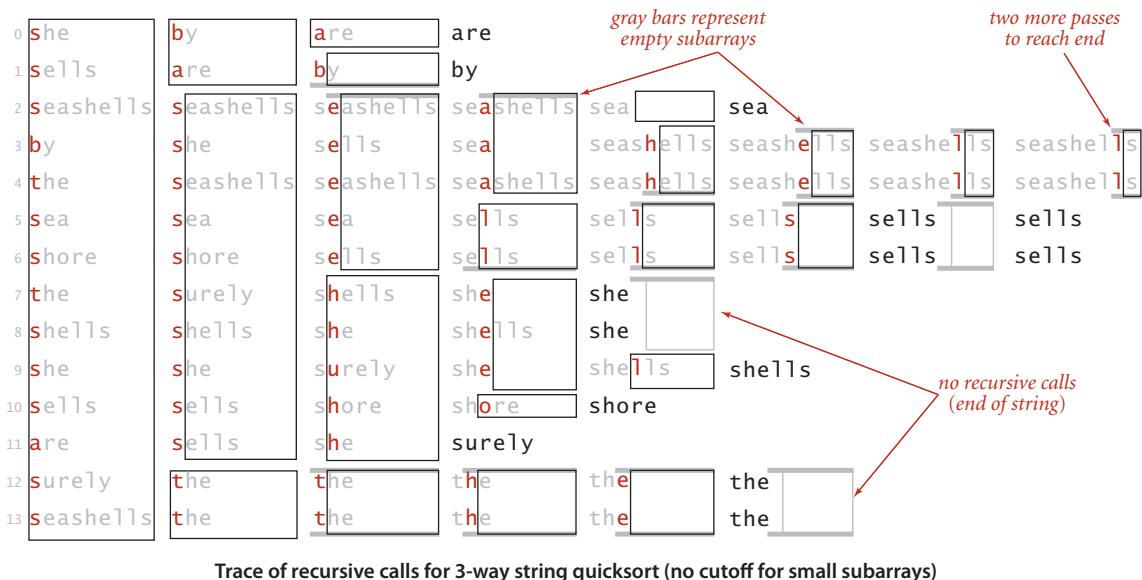
partitioning adapts to different kinds of structure in different parts of the key. Also, like quicksort, 3-way string quicksort does not use extra space (other than the implicit stack to support recursion), which is an important advantage over MSD string sort, which requires space for both frequency counts and an auxiliary array.

The figure at the bottom of this page shows all of the recursive calls that Quick3string makes for our example. Each subarray is sorted using precisely three recursive calls, except when we skip the recursive call on reaching the ends of the (equal) string(s) in the middle subarray.

As usual, in practice, it is worthwhile to consider various standard improvements to the implementation in ALGORITHM 5.3:

**Small subarrays.** In any recursive algorithm, we can gain efficiency by treating small subarrays differently. In this case, we use the insertion sort from page 715, which skips the characters that are known to be equal. The improvement due to this change is likely to be significant, though not nearly as important as for MSD string sort.

**Restricted alphabet.** To handle specialized alphabets, we could add an `Alphabet` argument `alpha` to each of the methods and replace `s.charAt(d)` with `alpha.toIndex(s.charAt(d))` in `charAt()`. In this case, there is no benefit to doing so, and adding this code is likely to substantially slow the algorithm down because this code is in the inner loop.



**Randomization.** As with any quicksort, it is generally worthwhile to shuffle the array beforehand or to use a random partitioning item by swapping the first item with a random one. The primary reason to do so is to protect against worst-case performance in the case that the array is already sorted or nearly sorted.

For string keys, standard quicksort and all the other sorts in CHAPTER 2 are actually MSD string sorts, because the `compareTo()` method in `String` accesses the characters in left-to-right order. That is, `compareTo()` accesses only the leading characters if they are different, the leading two characters if the first characters are the same and the second different, and so forth. For example, if the first characters of the strings are all different, the standard sorts will examine just those characters, thus automatically realizing some of the same performance gain that we seek in MSD string sorting. The essential idea behind 3-way quicksort is to take special action when the leading characters are equal. Indeed, one way to think of ALGORITHM 5.3 is as a way for standard quicksort to keep track of leading characters that are known to be equal. In the small subarrays, where most of the compares in the sort are done, the strings are likely to have numerous equal leading characters. The standard algorithm has to scan over all those characters for each compare; the 3-way algorithm avoids doing so.

**Performance.** Consider a case where the string keys are long (and are all the same length, for simplicity), but most of the leading characters are equal. In such a situation, the running time of standard quicksort is proportional to the string length *times*  $2N \ln N$ , whereas the running time of 3-way string quicksort is proportional to  $N$  times the string length (to discover all the leading equal characters) *plus*  $2N \ln N$  character comparisons (to do the sort on the remaining short keys). That is, 3-way string quicksort requires up to a factor of  $2 \ln N$  fewer character compares than normal quicksort. It is not unusual for keys in practical sorting applications to have characteristics similar to this artificial example.

**Proposition E.** To sort an array of  $N$  random strings, 3-way string quicksort uses  $\sim 2N \ln N$  character compares, on the average.

**Proof:** There are two instructive ways to understand this result. First, considering the method to be equivalent to quicksort partitioning on the leading character, then (recursively) using the same method on the subarrays, we should not be surprised that the total number of operations is about the same as for normal quicksort—but they are single-character compares, not full-key compares. Second, considering the method as replacing key-indexed counting by quicksort, we expect that the  $N \log_R N$  running time from PROPOSITION C should be multiplied by a factor of  $2 \ln R$  because it takes quicksort  $2R \ln R$  steps to sort  $R$  characters, as opposed to  $R$  steps for the same characters in the MSD string sort. We omit the full proof.

As emphasized on page 716, considering random strings is instructive, but more detailed analysis is needed to predict performance for practical situations. Researchers have studied this algorithm in depth and have proved that no algorithm can beat 3-way string quicksort (measured by number of character compares) by more than a constant factor, under very general assumptions. To appreciate its versatility, note that 3-way string quicksort has no direct dependencies on the size of the alphabet.

**Example: web logs.** As an example where 3-way string quicksort shines, we can consider a typical modern data-processing task. Suppose that you have built a website and want to analyze the traffic that it generates. You can have your system administrator supply you with a web log of all transactions on your site. Among the information associated with a transaction is the domain name of the originating machine. For example, the file `week.log.txt` on the booksite is a log of one week's transactions on our booksite. Why does 3-way string quicksort do well on such a file? Because the sorted result is replete with long common prefixes that this method does not have to reexamine.

**Which string-sorting algorithm should I use?** Naturally, we are interested in how the string-sorting methods that we have considered compare to the general-purpose methods that we considered in CHAPTER 2. The following table summarizes the important characteristics of the string-sort algorithms that we have discussed in this section (the rows for quicksort, mergesort, and 3-way quicksort are included from CHAPTER 2, for comparison).

algorithm	stable?	inplace?	order of growth of typical number calls to <code>charAt()</code> to sort $N$ strings from an $R$ -character alphabet (average length $w$ , max length $W$ )		sweet spot
			running time	extra space	
<i>insertion sort for strings</i>	yes	yes	between $N$ and $N^2$	1	small arrays, arrays in order
<i>quicksort</i>	no	yes	$N \log^2 N$	$\log N$	general-purpose when space is tight
<i>mergesort</i>	yes	no	$N \log^2 N$	$N$	general-purpose stable sort
<i>3-way quicksort</i>	no	yes	between $N$ and $N \log^2 N$	$\log N$	large numbers of equal keys
<i>LSD string sort</i>	yes	no	$NW$	$N$	short fixed-length strings
<i>MSD string sort</i>	yes	no	between $N$ and $Nw$	$N + WR$	random strings
<i>3-way string quicksort</i>	no	yes	between $N$ and $Nw \log R$	$W + \log N$	general-purpose, strings with long prefix matches

Performance characteristics of string-sorting algorithms

As in CHAPTER 2, multiplying these growth rates by appropriate algorithm- and data-dependent constants gives an effective way to predict running time.

As explored in the examples that we have already considered and in many other examples in the exercises, different specific situations call for different methods, with appropriate parameter settings. In the hands of an expert (maybe that's you, by now), dramatic savings can be realized for certain situations.

**Q&A**

- Q.** Does the Java system sort use one of these methods for `String` sorts?
- A.** No, but the standard implementation includes a fast string compare that makes standard sorts competitive with the methods considered here.
- Q.** So, I should just use the system sort for `String` keys?
- A.** Probably yes in Java, though if you have huge numbers of strings or need an exceptionally fast sort, you may wish to switch to `char` arrays instead of `String` values and use a radix sort.
- Q.** What is explanation of the  $\log^2 N$  factors on the table in the previous page?
- A.** They reflect the idea that most of the comparisons for these algorithms wind up being between keys with a common prefix of length  $\log N$ . Recent research has established this fact for random strings with careful mathematical analysis (see booksite for reference).

## EXERCISES

**5.1.1** Develop a sort implementation that counts the number of different key values, then uses a symbol table to apply key-indexed counting to sort the array. (This method is *not* for use when the number of different key values is large.)

**5.1.2** Give a trace for LSD string sort for the keys

no is th ti fo al go pe to co to th ai of th pa

**5.1.3** Give a trace for MSD string sort for the keys

no is th ti fo al go pe to co to th ai of th pa

**5.1.4** Give a trace for 3-way string quicksort for the keys

no is th ti fo al go pe to co to th ai of th pa

**5.1.5** Give a trace for MSD string sort for the keys

now is the time for all good people to come to the aid of

**5.1.6** Give a trace for 3-way string quicksort for the keys

now is the time for all good people to come to the aid of

**5.1.7** Develop an implementation of key-indexed counting that makes use of an array of Queue objects.

**5.1.8** Give the number of characters examined by MSD string sort and 3-way string quicksort for a file of  $N$  keys a, aa, aaa, aaaa, aaaaa, ...

**5.1.9** Develop an implementation of LSD string sort that works for variable-length strings.

**5.1.10** What is the total number of characters examined by 3-way string quicksort when sorting  $N$  fixed-length strings (all of length  $W$ ), in the worst case?

## CREATIVE PROBLEMS

**5.1.11 Queue sort.** Implement MSD string sorting using queues, as follows: Keep one queue for each bin. On a first pass through the items to be sorted, insert each item into the appropriate queue, according to its leading character value. Then, sort the sublists and stitch together all the queues to make a sorted whole. Note that this method does not involve keeping the `count[]` arrays within the recursive method.

**5.1.12 Alphabet.** Develop an implementation of the `Alphabet` API that is given on page 698 and use it to develop LSD and MSD sorts for general alphabets.

**5.1.13 Hybrid sort.** Investigate the idea of using standard MSD string sort for large arrays, in order to get the advantage of multiway partitioning, and 3-way string quicksort for smaller arrays, in order to avoid the negative effects of large numbers of empty bins.

**5.1.14 Array sort.** Develop a method that uses 3-way string quicksort for keys that are `arrays of int` values.

**5.1.15 Sublinear sort.** Develop a sort implementation for `int` values that makes two passes through the array to do an LSD sort on the leading 16 bits of the keys, then does an insertion sort.

**5.1.16 Linked-list sort.** Develop a sort implementation that takes a linked list of nodes with `String` key values as argument and rearranges the nodes so that they appear in sorted order (returning a link to the node with the smallest key). Use 3-way string quicksort.

**5.1.17 In-place key-indexed counting.** Develop a version of key-indexed counting that uses only a constant amount of extra space. Prove that your version is stable or provide a counterexample.

## EXPERIMENTS

**5.1.18 Random decimal keys.** Write a static method `randomDecimalKeys` that takes `int` values `N` and `W` as arguments and returns an array of `N` string values that are each `W`-digit decimal numbers.

**5.1.19 Random CA license plates.** Write a static method `randomPlatesCA` that takes an `int` value `N` as argument and returns an array of `N` `String` values that represent CA license plates as in the examples in this section.

**5.1.20 Random fixed-length words.** Write a static method `randomFixedLengthWords` that takes `int` values `N` and `W` as arguments and returns an array of `N` string values that are each strings of `W` characters from the alphabet.

**5.1.21 Random items.** Write a static method `randomItems` that takes an `int` value `N` as argument and returns an array of `N` string values that are each strings of length between 15 and 30 made up of three fields: a 4-character field with one of a set of 10 fixed strings; a 10-char field with one of a set of 50 fixed strings; a 1-character field with one of two given values; and a 15-byte field with random left-justified strings of letters equally likely to be 4 through 15 characters long.

**5.1.22 Timings.** Compare the running times of MSD string sort and 3-way string quicksort, using various key generators. For fixed-length keys, include LSD string sort.

**5.1.23 Array accesses.** Compare the number of array accesses used by MSD string sort and 3-way string sort, using various key generators. For fixed-length keys, include LSD string sort.

**5.1.24 Rightmost character accessed.** Compare the position of the rightmost character accessed for MSD string sort and 3-way string quicksort, using various key generators.

*This page intentionally left blank*

## 5.2 TRIES

As with sorting, we can take advantage of properties of strings to develop search methods (symbol-table implementations) that can be more efficient than the general-purpose methods of CHAPTER 3 for typical applications where search keys are strings.

Specifically, the methods that we consider in this section achieve the following performance characteristics in typical applications, even for huge tables:

- Search hits take time proportional to the length of the search key.
- Search misses involve examining only a few characters.

On reflection, these performance characteristics are quite remarkable, one of the crowning achievements of algorithmic technology and a primary factor in enabling the development of the computational infrastructure we now enjoy that has made so much information instantly accessible. Moreover, we can extend the symbol-table API to include character-based operations defined for string keys (but not necessarily for all Comparable types of keys) that are powerful and quite useful in practice, as in the following API:

---

<code>public class StringST&lt;Value&gt;</code>	
<code>StringST()</code>	<i>create a symbol table</i>
<code>void put(String key, Value val)</code>	<i>put key-value pair into the table (remove key if value is null)</i>
<code>Value get(String key)</code>	<i>value paired with key (null if key is absent)</i>
<code>void delete(String key)</code>	<i>remove key (and its value)</i>
<code>boolean contains(String key)</code>	<i>is there a value paired with key?</i>
<code>boolean isEmpty()</code>	<i>is the table empty?</i>
<code>String longestPrefixOf(String s)</code>	<i>the longest key that is a prefix of s</i>
<code>Iterable&lt;String&gt; keysWithPrefix(String s)</code>	<i>all the keys having s as a prefix</i>
<code>Iterable&lt;String&gt; keysThatMatch(String s)</code>	<i>all the keys that match s (where . matches any character)</i>
<code>int size()</code>	<i>number of key-value pairs</i>
<code>Iterable&lt;String&gt; keys()</code>	<i>all the keys in the table</i>

API for a symbol table with string keys

This API differs from the symbol-table API introduced in [CHAPTER 3](#) in the following aspects:

- We replace the generic type `Key` with the concrete type `String`.
- We add three new methods, `longestPrefixOf()`, `keysWithPrefix()` and `keysThatMatch()`.

We retain the basic conventions of our symbol-table implementations in [CHAPTER 3](#) (no duplicate or null keys and no null values).

As we saw for sorting with string keys, it is often quite important to be able to work with strings from a specified alphabet. Simple and efficient implementations that are the method of choice for small alphabets turn out to be useless for large alphabets because they consume too much space. In such cases, it is certainly worthwhile to add a constructor that allows clients to specify the alphabet. We will consider the implementation of such a constructor later in this section but omit it from the API for now, in order to concentrate on string keys.

The following descriptions of the three new methods use the keys `{ she, sells, sea, shells, by, the, sea, shore }` to give examples:

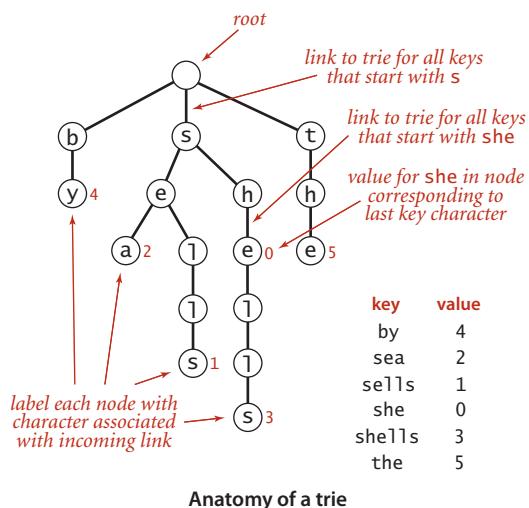
- `longestPrefixOf()` takes a string as argument and returns the longest key in the symbol table that is a prefix of that string. For the keys above, `longestPrefixOf("shell")` is `she` and `longestPrefixOf("shellsort")` is `shells`.
- `keysWithPrefix()` takes a string as argument and returns all the keys in the symbol table having that string as prefix. For the keys above, `keysWithPrefix("she")` is `she` and `shells`, and `keysWithPrefix("se")` is `sells` and `sea`.
- `keysThatMatch()` takes a string as argument and returns all the keys in the symbol table that match that string, in the sense that a period (.) in the argument string matches any character. For the keys above, `keysThatMatch(".he")` returns `she` and `the`, and `keysThatMatch("s..")` returns `she` and `sea`.

We will consider in detail implementations and applications of these operations after we have seen the basic symbol-table methods. These particular operations are representative of what is possible with string keys; we discuss several other possibilities in the exercises.

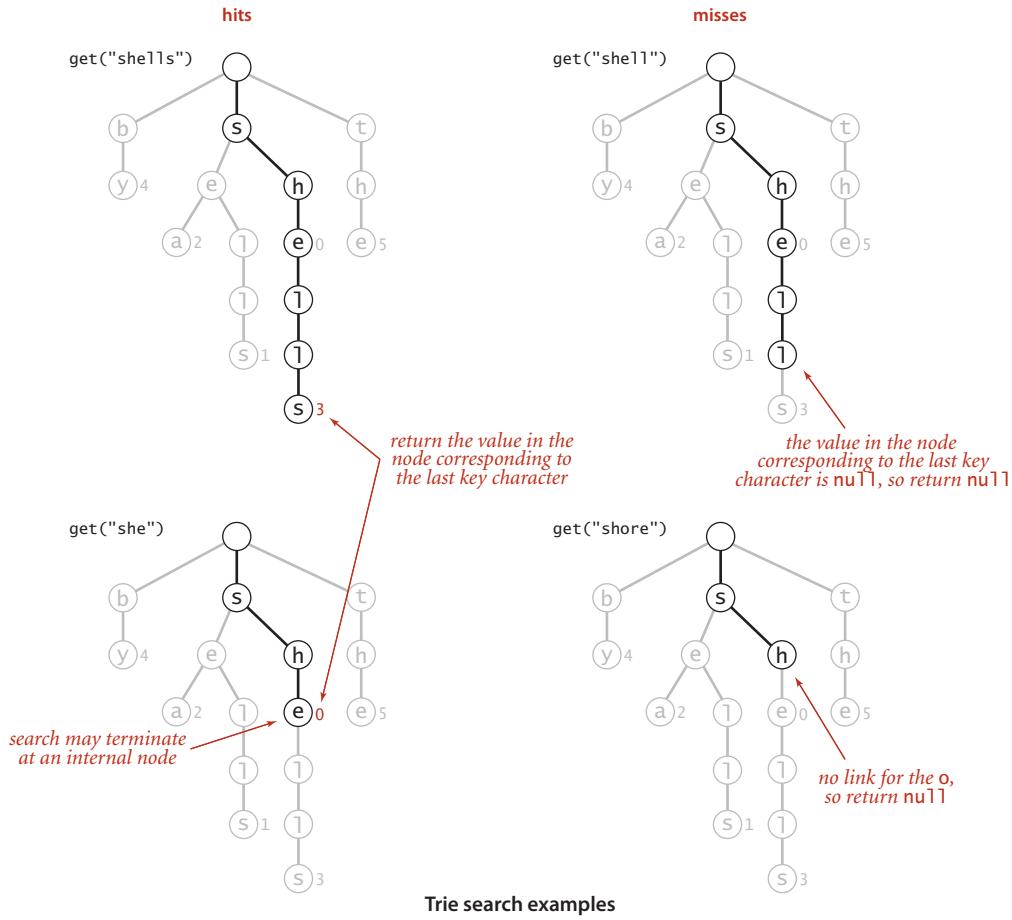
To focus on the main ideas, we concentrate on `put()`, `get()`, and the new methods; we assume (as in [CHAPTER 3](#)) default implementations of `contains()` and `isEmpty()`; and we leave implementations of `size()` and `delete()` for exercises. Since strings are `Comparable`, extending the API to also include the ordered operations defined in the ordered symbol-table API in [CHAPTER 3](#) is possible (and worthwhile); we leave those implementations (which are generally straightforward) to exercises and booksite code.

**Tries** In this section, we consider a search tree known as a *trie*, a data structure built from the characters of the string keys that allows us to use the characters of the search key to guide the search. The name “trie” is a bit of wordplay introduced by E. Fredkin in 1960 because the data structure is used for retrieval, but we pronounce it “try” to avoid confusion with “tree.” We begin with a high-level description of the basic properties of tries, including search and insert algorithms, and then proceed to the details of the representation and Java implementation.

**Basic properties.** As with search trees, tries are data structures composed of *nodes* that contain *links* that are either *null* or references to other nodes. Each node is pointed to by just one other node, which is called its *parent* (except for one node, the *root*, which has no nodes pointing to it), and each node has  $R$  links, where  $R$  is the alphabet size. Often, tries have a substantial number of null links, so when we draw a trie, we typically omit null links. Although links point to nodes, we can view each link as pointing to a trie, the trie whose root is the referenced node. Each link corresponds to a character value—since each link points to exactly one node, we label each node with the character value corresponding to the link that points to it (except for the root, which has no link pointing to it). Each node also has a corresponding *value*, which may be *null* or the value associated with one of the string keys in the symbol table. Specifically, we store the value associated with each key in the node corresponding to its last character. It is very important to bear in mind the following fact: *nodes with null values exist to facilitate search in the trie and do not correspond to keys*. An example of a trie is shown at right.



**Search in a trie.** Finding the value associated with a given string key in a trie is a simple process, guided by the characters in the search key. Each node in the trie has a link corresponding to each possible string character. We start at the root, then follow the link associated with the first character in the key; from that node we follow the link associated with the second character in the key; from that node we follow the link associated with the third character in the key and



so forth, until reaching the last character of the key or a null link. At this point, one of the following three conditions holds (refer to the figure above for examples):

- The value at the node corresponding to the last character in the key is not `null` (as in the searches for `shells` and `she` depicted at left above). This result is a *search hit*—the value associated with the key is the value in the node corresponding to its last character.
- The value in the node corresponding to the last character in the key is `null` (as in the search for `shell` depicted at top right above). This result is a *search miss*: the key is not in the table.
- The search terminated with a null link (as in the search for `shore` depicted at bottom right above). This result is also a search miss.

In all cases, the search is accomplished just by examining nodes along a path from the root to another node in the trie.

**Insertion into a trie.** As with binary search trees, we insert by first doing a search: in a trie that means using the characters of the key to guide us down the trie until reaching the last character of the key or a null link. At this point, one of the following two conditions holds:

- We encountered a null link before reaching the last character of the key. In this case, there is no trie node corresponding to the last character in the key, so we need to create nodes for each of the characters in the key not yet encountered and set the value in the last one to the value to be associated with the key.
- We encountered the last character of the key before reaching a null link. In this case, we set that node's value to the value to be associated with the key (whether or not that value is null), as usual with our associative array convention.

In all cases, we examine or create a node in the trie for each key character. The construction of the trie for our standard indexing client from CHAPTER 3 with the input

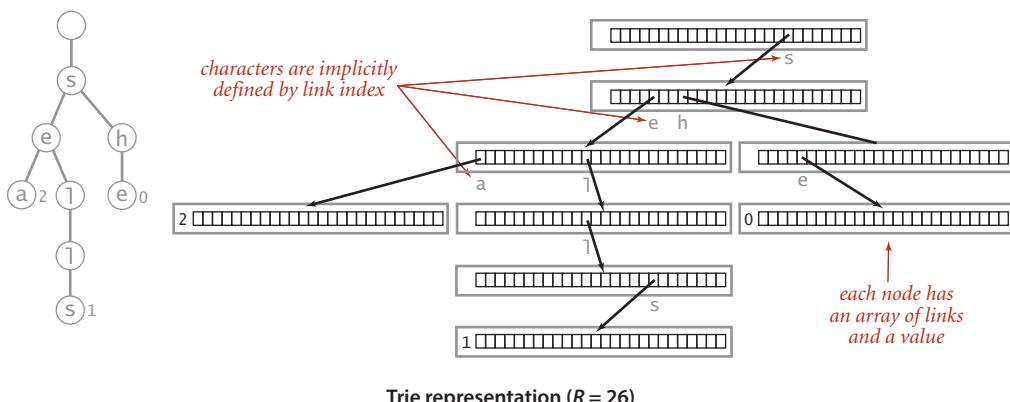
she sells sea shells by the sea shore

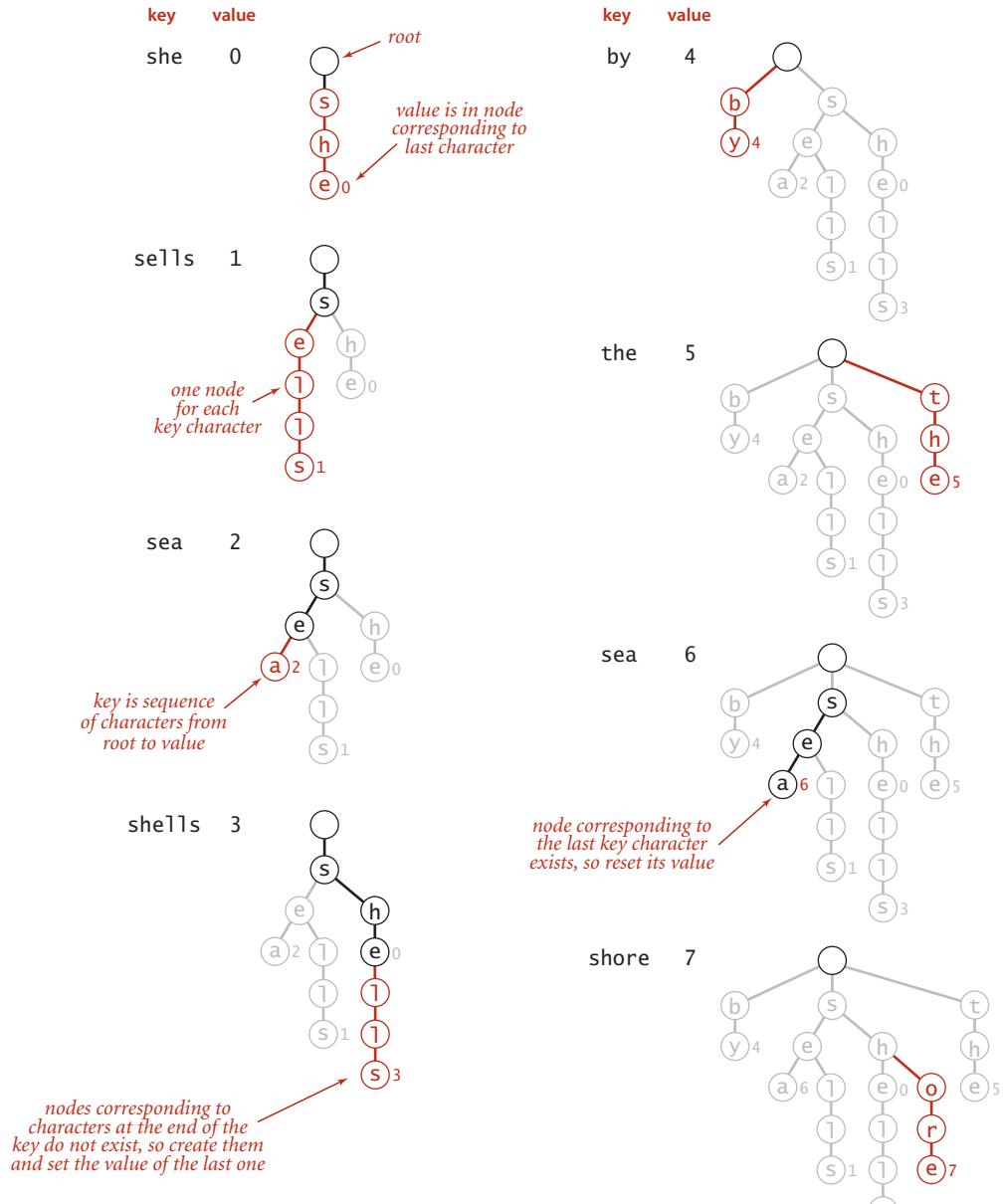
is shown on the facing page.

**Node representation.** As mentioned at the outset, our trie diagrams do not quite correspond to the data structures our programs will build, because we do not draw null links. Taking null links into account emphasizes the following important characteristics of tries:

- Every node has  $R$  links, one for each possible character.
- Characters and keys are *implicitly* stored in the data structure.

For example, the figure below depicts a trie for keys made up of lowercase letters, with each node having a value and 26 links. The first link points to a subtrie for keys beginning with a, the second points to a subtrie for substrings beginning with b, and so forth.





Trie construction trace for standard indexing client

Keys in the trie are implicitly represented by paths from the root that end at nodes with non-null values. For example, the string `sea` is associated with the value 2 in the trie because the 19th link in the root (which points to the trie for all keys that start with `s`) is not null and the 5th link in the node that link refers to (which points to the trie for all keys that start with `se`) is not null, and the first link in the node that link refers to (which points to the trie for all keys that starts with `sea`) has the value 2. Neither the string `sea` nor the characters `s`, `e`, and `a` are stored in the data structure. Indeed, the data structure contains no characters or strings, just links and values. Since the parameter  $R$  plays such a critical role, we refer to a trie for an  $R$ -character alphabet as an  *$R$ -way trie*.

WITH THESE PREPARATIONS, the symbol-table implementation `TrieST` on the facing page is straightforward. It uses recursive methods like those that we used for search trees in CHAPTER 3, based on a private `Node` class with instance variable `val` for client values and an array `next[]` of `Node` references. The methods are compact recursive implementations that are worthy of careful study. Next, we discuss implementations of the constructor that takes an `Alphabet` as argument and the methods `size()`, `keys()`, `longestPrefixOf()`, `keysWithPrefix()`, `keysThatMatch()`, and `delete()`. These are also easily understood recursive methods, each slightly more complicated than the last.

**Size.** As for the binary search trees of CHAPTER 3, three straightforward options are available for implementing `size()`:

- An eager implementation where we maintain the number of keys in an instance variable  $N$ .
- A very eager implementation where we maintain the number of keys in a subtree as a node instance variable that we update after the recursive calls in `put()` and `delete()`.
- A lazy recursive implementation like the one at right. It traverses all of the nodes in the trie, counting the number having a non-null value.

As with binary search trees, the lazy implementation is instructive but should be avoided because it can lead to performance problems for clients. The eager implementations are explored in the exercises.

```
public int size()
{ return size(root); }

private int size(Node x)
{
    if (x == null) return 0;
    int cnt = 0;
    if (x.val != null) cnt++;
    for (char c = 0; c < R; c++)
        cnt += size(next[c]);
    return cnt;
}
```

Lazy recursive `size()` for tries

**ALGORITHM 5.4** Trie symbol table

```
public class TrieST<Value>
{
    private static int R = 256;          // radix
    private Node root = new Node();     // root of trie

    private static class Node
    {
        private Object val;
        private Node[] next = new Node[R];
    }

    public Value get(String key)
    {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val;
    }

    private Node get(Node x, String key, int d)
    { // Return node associated with key in the subtrie rooted at x.
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d); // Use dth key char to identify subtrie.
        return get(x.next[c], key, d+1);
    }

    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    { // Change value associated with key if in subtrie rooted at x.
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d); // Use dth key char to identify subtrie.
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }
}
```

This code uses an  $R$ -way trie to implement a symbol table. Additional methods in the string symbol-table API of page 730 are presented in the next several pages. Modifying this code to handle keys from specialized alphabets is straightforward (see page 741). The value in `Node` has to be an `Object` because Java does not support arrays of generics; we cast values back to `Value` in `get()`.

**Collecting keys.** Because characters and keys are represented implicitly in tries, providing clients with the ability to iterate through the keys presents a challenge. As with binary search trees, we accumulate the string keys in a Queue, but for tries we need to create explicit representations of all of the string keys, not just find them in the data structure. We do so with a recursive private method `collect()` that is similar to `size()` but also maintains a string with the sequence of characters on the path from the root. Each time that we visit a node via a call to `collect()` with that node as first argument, the second

```
public Iterable<String> keys()
{   return keysWithPrefix(""); }

public Iterable<String> keysWithPrefix(String pre)
{
    Queue<String> q = new Queue<String>();
    collect(get(root, pre, 0), pre, q);
    return q;
}

private void collect(Node x, String pre,
                     Queue<String> q)
{
    if (x == null) return;
    if (x.val != null) q.enqueue(pre);
    for (char c = 0; c < R; c++)
        collect(x.next[c], pre + c, q);
}
```

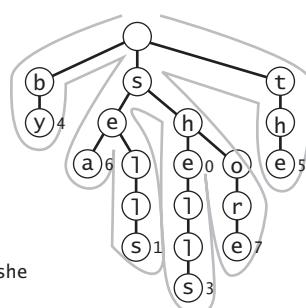
### Collecting the keys in a trie

```

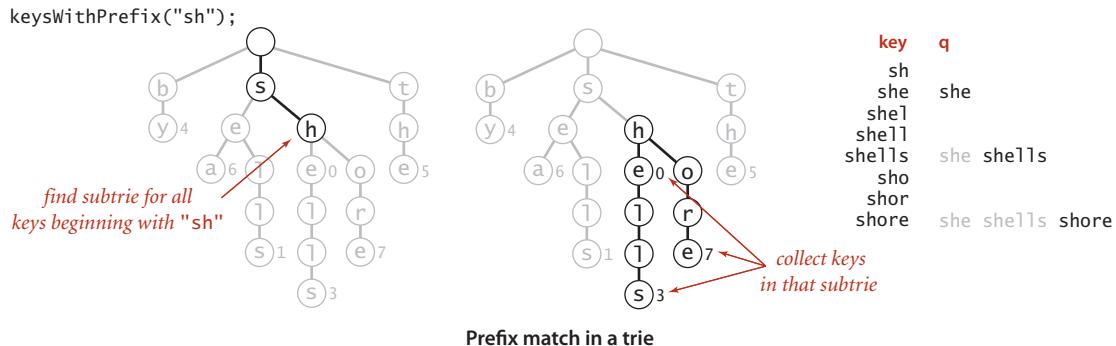
keysWithPrefix("");
key  q
  b by
  by
  s
  se
  sea by sea
  sel
  sell
  sells by sea sells
  sh
  she
  shel
  shell
  shells by sea sells she shells
  sho
  shor
  shore
  t
  th
  the by sea sells she shells shore the

```

### Collecting the keys in a trie (trace)



argument is the string associated with that node (the sequence of characters on the path from the root to the node). To visit a node, we add its associated string to the queue if its value is not null, then visit (recursively) all the nodes in its array of links, one for each possible character. To create the key for each call, we append the character corresponding to the link to the current key. We use this `collect()` method to collect keys for both the `keys()` and the `keysWithPrefix()` methods in the API. To implement `keys()` we call `keysWithPrefix()` with the empty string as argument; to implement `keysWithPrefix()`, we call `get()` to find the trie node corresponding to the given prefix (`null` if there is no such node), then use the `collect()` method to complete the job. The diagram at left shows a trace of `collect()` (or `keysWithPrefix("")`) for an example trie, giving the value of the second argument key and the contents of the queue for each call to `collect()`. The diagram at the top of the facing page illustrates the process for `keysWithPrefix("sh")`.



**Wildcard match.** To implement `keysThatMatch()`, we use a similar process, but add an argument specifying the pattern to `collect()` and add a test to make a recursive call for all links when the pattern character is a wildcard or only for the link corresponding to the pattern character otherwise, as in the code below. Note also that we do not need to consider keys longer than the pattern.

**Longest prefix.** To find the longest key that is a prefix of a given string, we use a recursive method like `get()` that keeps track of the length of the longest key found on the search path (by passing it as a parameter to the recursive method, updating the value

```
public Iterable<String> keysThatMatch(String pat)
{
    Queue<String> q = new Queue<String>();
    collect(root, "", pat, q);
    return q;
}

private void collect(Node x, String pre, String pat, Queue<String> q)
{
    int d = pre.length();
    if (x == null) return;
    if (d == pat.length() && x.val != null) q.enqueue(pre);
    if (d == pat.length()) return;

    char next = pat.charAt(d);
    for (char c = 0; c < R; c++)
        if (next == '.' || next == c)
            collect(x.next[c], pre + c, pat, q);
}
```

Wildcard match in a trie

```

public String longestPrefixOf(String s)
{
    int length = search(root, s, 0, 0);
    return s.substring(0, length);
}

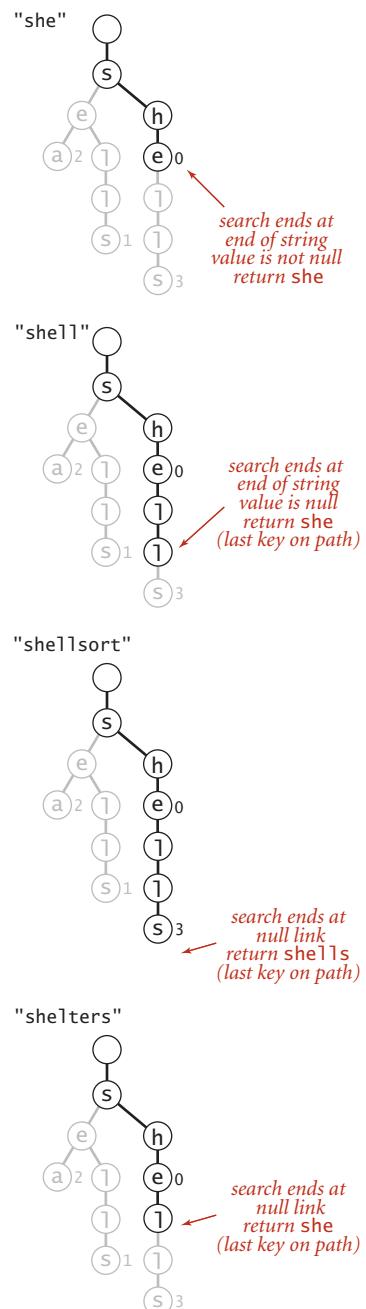
private int search(Node x, String s, int d, int length)
{
    if (x == null) return length;
    if (x.val != null) length = d;
    if (d == s.length()) return length;
    char c = s.charAt(d);
    return search(x.next[c], s, d+1, length);
}

```

#### Matching the longest prefix of a given string

whenever a node with a non-null value is encountered). The search ends when the end of the string or a null link is encountered, whichever comes first.

**Deletion.** The first step needed to delete a key-value pair from a trie is to use a normal search to find the node corresponding to the key and set the corresponding value to `null`. If that node has a non-null link to a child, then no more work is required; if all the links are null, we need to remove the node from the data structure. If doing so leaves all the links null in its parent, we need to remove that node, and so forth. The implementation on the facing page demonstrates that this action can be accomplished with remarkably little code, using our standard recursive setup: after the recursive calls for a node `x`, we return `null` if the client value and all of the links in a node are `null`; otherwise we return `x`.



Possibilities for `longestPrefixOf()`

**Alphabet.** As usual, ALGORITHM 5.4 is coded for Java String keys, but it is a simple matter to modify the implementation to handle keys taken from any alphabet, as follows:

- Implement a constructor that takes an Alphabet as argument, which sets an Alphabet instance variable to that argument value and the instance variable R to the number of characters in the alphabet.
- Use the toIndex() method from Alphabet in get() and put() to convert string characters to indices between 0 and  $R-1$ .
- Use the toChar() method from Alphabet to convert indices between 0 and  $R-1$  to char values. This operation is not needed in get() and put() but is important in the implementations of keys(), keysWithPrefix(), and keysThatMatch().

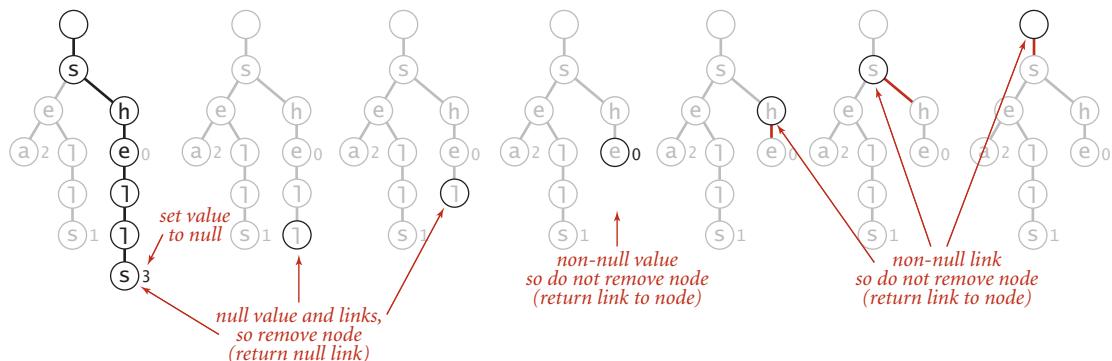
With these changes, you can save a considerable amount of space (use only  $R$  links per node) when you know that your keys are taken from a small alphabet, at the cost of the time required to do the conversions between characters and indices.

```
public void delete(String key)
{ root = delete(root, key, 0); }

private Node delete(Node x, String key, int d)
{
    if (x == null) return null;
    if (d == key.length())
        x.val = null;
    else
    {
        char c = key.charAt(d);
        x.next[c] = delete(x.next[c], key, d+1);
    }
    if (x.val != null) return x;
    for (char c = 0; c < R; c++)
        if (x.next[c] != null) return x;
    return null;
}
```

Deleting a key (and its associated value) from a trie

delete("shells");



Deleting a key (and its associated value) from a trie

THE CODE THAT WE HAVE CONSIDERED is a compact and complete implementation of the string symbol-table API that has broadly useful practical applications. Several variations and extensions are discussed in the exercises. Next, we consider basic properties of tries, and some limitations on their utility.

**Properties of tries** As usual, we are interested in knowing the amount of time and space required to use tries in typical applications. Tries have been extensively studied and analyzed, and their basic properties are relatively easy to understand and to apply.

**Proposition F.** The linked structure (shape) of a trie is independent of the key insertion/deletion order: there is a unique trie for any given set of keys.

**Proof:** Immediate, by induction on the subtrees.

This fundamental fact is a distinctive feature of tries: for all of the other search tree structures that we have considered so far, the tree that we construct depends both on the set of keys and on the order in which we insert those keys.

**Worst-case time bound for search and insert.** How long does it take to find the value associated with a key? For BSTs, hashing, and other methods in CHAPTER 3, we needed mathematical analysis to study this question, but for tries it is very easy to answer:

**Proposition G.** The number of array accesses when searching in a trie or inserting a key into a trie is at most 1 plus the length of the key.

**Proof:** Immediate from the code. The recursive `get()` and `put()` implementations carry an argument `d` that starts at 0, increments for each call, and is used to stop the recursion when it reaches the key length.

From a theoretical standpoint, the implication of PROPOSITION G is that tries are *optimal* for search hit—we could not expect to do better than search time proportional to the length of the search key. Whatever algorithm or data structure we are using, we cannot know that we have found a key that we seek without examining all of its characters. From a practical standpoint this guarantee is important because *it does not depend on the number of keys*: when we are working with 7-character keys like license plate numbers, we know that we need to examine at most 8 nodes to search or insert; when we are working with 20-digit account numbers, we only need to examine at most 21 nodes to search or insert.

**Expected time bound for search miss.** Suppose that we are searching for a key in a trie and find that the link in the root node that corresponds to its first character is null. In this case, we know that the key is not in the table on the basis of examining just *one* node. This case is typical: one of the most important properties of tries is that search misses typically require examining just a few nodes. If we assume that the keys are drawn from the random string model (each character is equally likely to have any one of the  $R$  different character values) we can prove this fact:

**Proposition H.** The average number of nodes examined for search miss in a trie built from  $N$  random keys over an alphabet of size  $R$  is  $\sim \log_R N$ .

**Proof sketch** (for readers who are familiar with probabilistic analysis): The probability that each of the  $N$  keys in a random trie differs from a random search key in at least one of the leading  $t$  characters is  $(1 - R^{-t})^N$ . Subtracting this quantity from 1 gives the probability that one of the keys in the trie matches the search key in all of the leading  $t$  characters. In other words,  $1 - (1 - R^{-t})^N$  is the probability that the search requires more than  $t$  character compares. From probabilistic analysis, the sum for  $t = 0, 1, 2, \dots$  of the probabilities that an integer random variable is  $> t$  is the average value of that random variable, so the average search cost is

$$1 - (1 - R^{-1})^N + 1 - (1 - R^{-2})^N + \dots + 1 - (1 - R^{-t})^N + \dots$$

Using the elementary approximation  $(1 - 1/x)^x \sim e^{-1}$ , we find the search cost to be approximately

$$(1 - e^{-N/R^1}) + (1 - e^{-N/R^2}) + \dots + (1 - e^{-N/R^t}) + \dots$$

The summand is extremely close to 1 for approximately  $\log_R N$  terms with  $R^t$  substantially smaller than  $N$ ; it is extremely close to 0 for all the terms with  $R^t$  substantially greater than  $N$ ; and it is somewhere between 0 and 1 for the few terms with  $R^t \approx N$ . So the grand total is about  $\log_R N$ .

From a practical standpoint, the most important implication of this proposition is that *search miss does not depend on the key length*. For example, it says that unsuccessful search in a trie built with 1 million random keys will require examining only three or four nodes, whether the keys are 7-digit license plates or 20-digit account numbers. While it is unreasonable to expect truly random keys in practical applications, it is reasonable to hypothesize that the behavior of trie algorithms for keys in typical applica-

tions is described by this model. Indeed, this sort of behavior is widely seen in practice and is an important reason for the widespread use of tries.

**Space.** How much space is needed for a trie? Addressing this question (and understanding how much space is *available*) is critical to using tries effectively.

**Proposition I.** The number of links in a trie is between  $RN$  and  $RNw$ , where  $w$  is the average key length.

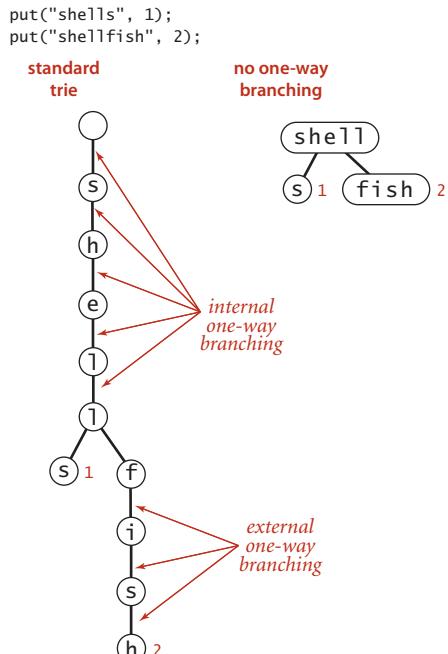
**Proof:** Every key in the trie has a node containing its associated value that also has  $R$  links, so the number of links is at least  $RN$ . If the first characters of all the keys are different, then there is a node with  $R$  links for every key character, so the number of links is  $R$  times the total number of key characters, or  $RNw$ .

The table on the facing page shows the costs for some typical applications that we have considered. It illustrates the following rules of thumb for tries:

- When keys are short, the number of links is close to  $RN$ .
- When keys are long, the number of links is close to  $RNw$ .
- Therefore, decreasing  $R$  can save a huge amount of space.

A more subtle message of this table is that it is important to understand the properties of the keys to be inserted before deploying tries in an application.

**One-way branching.** The primary reason that trie space is excessive for long keys is that long keys tend to have long tails in the trie, with each node having a single link to the next node (and, therefore,  $R-1$  null links). This situation known as *external one-way branching* is not difficult to correct (see EXERCISE 5.2.11). A trie might also have *internal one-way branching*. For example, two long keys may be equal except for their last character. This situation is a bit more difficult to address (see EXERCISE 5.2.12). These changes can make trie space



Removing one-way branching in a trie

usage a less important factor than for the straightforward implementation that we have considered, but they are not necessarily effective in practical applications. Next, we consider an alternative approach to reducing space usage for tries.

THE BOTTOM LINE is this: *do not try to use ALGORITHM 5.4 for large numbers of long keys taken from large alphabets*, because it will require space proportional to  $R$  times the total number of key characters. Otherwise, if you can afford the space, trie performance is difficult to beat.

application	typical key	average length $w$	alphabet size $R$	links in trie built from 1 million keys
CA license plates	4PGC938	7	256	256 million
account numbers	02400019992993299111	20	256	4 billion
			10	256 million
URLs	www.cs.princeton.edu	28	256	4 billion
text processing	seashells	11	256	256 million
proteins in genomic data	ACTGACTG	8	256	256 million
			4	4 million

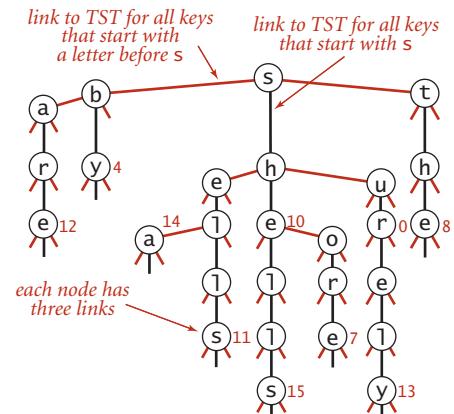
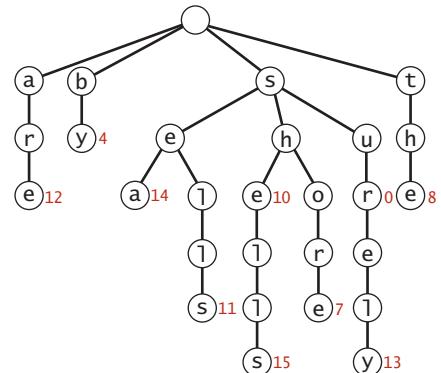
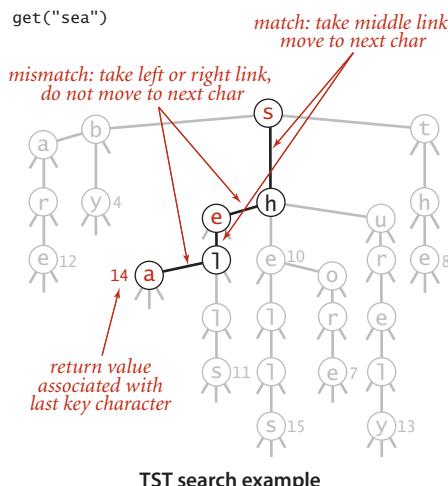
Space requirements for typical tries

**Ternary search tries (TSTs)** To help us avoid the excessive space cost associated with  $R$ -way tries, we now consider an alternative representation: the *ternary search trie* (TST). In a TST, each node has a character, *three* links, and a value. The three links correspond to keys whose current characters are less than, equal to, or greater than the node's character. In the  $R$ -way tries of ALGORITHM 5.4, trie nodes are represented by  $R$  links, with the character corresponding to each non-null link implicitly represented by its index. In the corresponding TST, characters appear *explicitly* in nodes—we find characters corresponding to keys only when we are traversing the middle links.

**Search and insert.** The search and insert code for implementing our symbol-table API with TSTs writes itself. To search, we compare the first character in the key with the character at the root. If it is less, we take the left link; if it is greater, we take the right link; and if it is equal, we take the middle link and move to the next search key character. In each case, we apply

the algorithm recursively. We terminate with a *search miss* if we encounter a null link or if the node where the search ends has a null value, and we terminate with a *search hit* if the node where the search ends has a non-null value. To insert a new key, we search, then add new nodes for the characters in the tail of the key, just as we did for tries. ALGORITHM 5.5 gives the details of the implementation of these methods.

Using this arrangement is equivalent to implementing each  $R$ -way trie node as a binary search tree that uses as keys the characters corresponding to non-null links. By contrast, ALGORITHM 5.4 uses a key-indexed array. A



TST representation of a trie

**ALGORITHM 5.5 TST symbol table**

```
public class TST<Value>
{
    private Node root;           // root of trie

    private class Node
    {
        char c;                 // character
        Node left, mid, right;   // left, middle, and right subtrees
        Value val;               // value associated with string
    }

    public Value get(String key) // same as for tries (See page 737).
    private Node get(Node x, String key, int d)
    {
        if (x == null) return null;
        char c = key.charAt(d);
        if (c < x.c) return get(x.left, key, d);
        else if (c > x.c) return get(x.right, key, d);
        else if (d < key.length() - 1)
            return get(x.mid, key, d+1);
        else return x;
    }

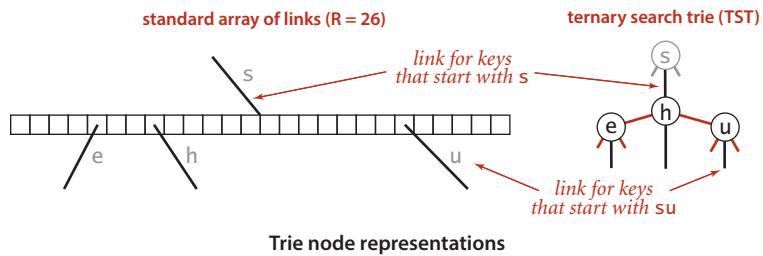
    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    {
        char c = key.charAt(d);
        if (x == null) { x = new Node(); x.c = c; }
        if (c < x.c) x.left = put(x.left, key, val, d);
        else if (c > x.c) x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1)
            x.mid = put(x.mid, key, val, d+1);
        else x.val = val;
        return x;
    }
}
```

---

This implementation uses a `char` value `c` and three links per node to build string search tries where subtrees have keys whose first character is less than `c` (left), equal to `c` (middle), and greater than `c` (right).

TST and its corresponding trie are illustrated above. Continuing the correspondence described in CHAPTER 3 between binary search trees and sorting algorithms, we see that TSTs correspond to 3-way string quicksort in the same way that BSTs correspond to quicksort and tries correspond to MSD sorting. The figures on page 714 and 721, which show the recursive call structure for MSD and 3-way string quicksort (respectively), correspond precisely to the trie and TST drawn on page 746 for that set of keys. Space for links in tries corresponds to the space for counters in string sorting; 3-way branching provides an effective solution to both problems.



**Properties of TSTs** A TST is a compact representation of an  $R$ -way trie, but the two data structures have remarkably different properties. Perhaps the most important difference is that PROPOSITION F does not hold for TSTs: the BST representations of each trie node depend on the order of key insertion, as with any other BST.

**Space.** The most important property of TSTs is that they have just three links in each node, so a TST requires far less space than the corresponding trie.

**Proposition J.** The number of links in a TST built from  $N$  string keys of average length  $w$  is between  $3N$  and  $3Nw$ .

**Proof.** Immediate, by the same argument as for PROPOSITION I.

Actual space usage is generally less than the upper bound of three links per character, because keys with common prefixes share nodes at high levels in the tree.

**Search cost.** To determine the cost of search (and insert) in a TST, we multiply the cost for the corresponding trie by the cost of traversing the BST representation of each trie node.

**Proposition K.** A search miss in a TST built from  $N$  random string keys requires  $\sim \ln N$  character compares, on the average. A search hit or an insertion in a TST uses  $\sim \ln N + L$  character compares, where  $L$  is the length of the search key.

**Proof:** The search hit/insertion cost is immediate from the code. The search miss cost is a consequence of the same arguments discussed in the proof sketch of PROPOSITION H. We assume that all but a constant number of the nodes on the search path (a few at the top) act as random BSTs on  $R$  character values with average path length  $\ln R$ , so we multiply the time cost  $\log_R N = \ln N / \ln R$  by  $\ln R$ .

In the worst case, a node might be a full  $R$ -way node that is unbalanced, stretched out like a singly linked list, so we would need to multiply by a factor of  $R$ . More typically, we might expect to do  $\ln R$  or fewer character compares at the first level (since the root node behaves like a random BST on the  $R$  different character values) and perhaps at a few other levels (if there are keys with a common prefix and up to  $R$  different values on the character following the prefix), and to do only a few compares for most characters (since most trie nodes are sparsely populated with non-null links). Search misses are likely to involve only a few character compares, ending at a null link high in the trie,

and search hits involve only about one compare per search key character, since most of them are in nodes with one-way branching at the bottom of the trie.

**Alphabet.** The prime virtue of using TSTs is that they adapt gracefully to irregularities in search keys that are likely to appear in practical applications. In particular, note that there is no reason to allow for strings to be built from a client-supplied alphabet, as was crucial for tries. There are two main effects. First, keys in practical applications come from large alphabets, and usage of particular characters in the character sets is far from uniform. With TSTs, we can use a 256-character ASCII encoding or a 65,536-character Unicode encoding without having to worry about the excessive costs of nodes with 256- or 65,536-way branching, and without having to determine which sets of characters are relevant. Unicode strings in non-Roman alphabets can have thousands of characters—TSTs are especially appropriate for standard Java `String` keys that consist of such characters. Second, keys in practical applications often have a structured format, differing from application to application, perhaps using only letters in one part of the key, only digits in another part of the key. In our CA license plate example, the second, third, and fourth characters are uppercase letter ( $R = 26$ ) and the other characters are decimal digits ( $R = 10$ ). In a TST for such keys, some of the trie nodes will be represented as 10-node BSTs (for places where all keys have digits) and others will be represented as 26-node BSTs (for places where all keys have letters). This structure develops automatically, without any need for special analysis of the keys.

**Prefix match, collecting keys, and wildcard match.** Since a TST represents a trie, implementations of `longestPrefixOf()`, `keys()`, `keysWithPrefix()`, and `keysThatMatch()` are easily adapted from the corresponding code for tries in the previous section, and a worthwhile exercise for you to cement your understanding of both tries and TSTs (see EXERCISE 5.2.9). The same tradeoff as for search (linear memory usage but an extra  $\ln R$  multiplicative factor per character compare) holds.

**Deletion.** The `delete()` method for TSTs requires more work. Essentially, each character in the key to be deleted belongs to a BST. In a trie, we could remove the link corresponding to a character by setting the corresponding entry in the array of links to `null`; in a TST, we have to use BST node deletion to remove the node corresponding to the character.

**Hybrid TSTs.** An easy improvement to TST-based search is to use a large explicit multiway node at the root. The simplest way to proceed is to keep a table of  $R$  TSTs: one for each possible value of the first character in the keys. If  $R$  is not large, we might use the first two letters of the keys (and a table of size  $R^2$ ). For this method to be effective, the leading digits of the keys must be well-distributed. The resulting hybrid search

algorithm corresponds to the way that a human might search for names in a telephone book. The first step is a multiway decision (“Let’s see, it starts with ‘A’”), followed perhaps by some two-way decisions (“It’s before ‘Andrews,’ but after ‘Aitken,’”), followed by sequential character matching (“‘Algonquin,’ ... No, ‘Algorithms’ isn’t listed, because nothing starts with ‘Algor!’”). These programs are likely to be among the fastest available for searching with string keys.

**One-way branching.** Just as with tries, we can make TSTs more efficient in their use of space by putting keys in leaves at the point where they are distinguished and by eliminating one-way branching between internal nodes.

**Proposition L.** A search or an insertion in a TST built from  $N$  random string keys with no external one-way branching and  $R^t$ -way branching at the root requires roughly  $\ln N - t \ln R$  character compares, on the average.

**Proof:** These rough estimates follow from the same argument we used to prove PROPOSITION K. We assume that all but a constant number of the nodes on the search path (a few at the top) act as random BSTs on  $R$  character values, so we multiply the time cost by  $\ln R$ .

DESPITE THE TEMPTATION to tune the algorithm to peak performance, we should not lose sight of the fact that one of the most attractive features of TSTs is that they free us from having to worry about application-specific dependencies, often providing good performance without any tuning.

**Which string symbol-table implementation should I use?** As with string sorting, we are naturally interested in how the string-searching methods that we have considered compare to the general-purpose methods that we considered in CHAPTER 3. The following table summarizes the important characteristics of the algorithms that we have discussed in this section (the rows for BSTs, red-black BSTs and hashing are included from CHAPTER 3, for comparison). For a particular application, these entries must be taken as indicative, not definitive, since so many factors (such as characteristics of keys and mix of operations) come into play when studying symbol-table implementations.

algorithm (data structure)	typical growth rate for $N$ strings from an $R$ -character alphabet (average length $w$ )		sweet spot
	characters examined for search miss	memory usage	
<i>binary tree search (BST)</i>	$c_1 (\lg N)^2$	$64N$	randomly ordered keys
<i>2-3 tree search (red-black BST)</i>	$c_2 (\lg N)^2$	$64N$	guaranteed performance
<i>linear probing<sup>†</sup> (parallel arrays)</i>	$w$	$32N$ to $128N$	built-in types cached hash values
<i>trie search (<math>R</math>-way trie)</i>	$\log_R N$	$(8R+56)N$ to $(8R+56)Nw$	short keys small alphabets
<i>trie search (TST)</i>	$1.39 \lg N$	$64N$ to $64Nw$	nonrandom keys

<sup>†</sup> under uniform hashing assumption

#### Performance characteristics of string-searching algorithms

If space is available,  $R$ -way tries provide the fastest search, essentially completing the job with a *constant* number of character compares. For large alphabets, where space may not be available for  $R$ -way tries, TSTs are preferable, since they use a logarithmic number of *character* compares, while BSTs use a logarithmic number of *key* compares. Hashing can be competitive, but, as usual, cannot support ordered symbol-table operations or extended character-based API operations such as prefix or wildcard match.

**Q&A**

- Q.** Does the Java system sort use one of these methods for searching with `String` keys?
- A.** No.

## EXERCISES

**5.2.1** Draw the  $R$ -way trie that results when the keys

no is th ti fo al go pe to co to th ai of th pa

are inserted in that order into an initially empty trie (do not draw null links).

**5.2.2** Draw the TST that results when the keys

no is th ti fo al go pe to co to th ai of th pa

are inserted in that order into an initially empty TST.

**5.2.3** Draw the  $R$ -way trie that results when the keys

now is the time for all good people to come to the aid of

are inserted in that order into an initially empty trie (do not draw null links).

**5.2.4** Draw the TST that results when the keys

now is the time for all good people to come to the aid of

are inserted in that order into an initially empty TST.

**5.2.5** Develop nonrecursive versions of `TrieST` and `TST`.

**5.2.6** Implement the following API, for a `StringSET` data type:

---

```
public class StringSET
```

StringSET()	<i>create a string set</i>
-------------	----------------------------

void add(String key)	*put key into the set*
void delete(String key)	*remove key from the set*
boolean contains(String key)	*is key in the set?*
boolean isEmpty()	*is the set empty?*
int size()	*number of keys in the set*
String toString()	*string representation of the set*

API for a string set data type

## CREATIVE PROBLEMS

**5.2.7** *Empty string in TSTs.* The code in TST does not handle the empty string properly. Explain the problem and suggest a correction.

**5.2.8** *Ordered operations for tries.* Implement the `floor()`, `ceiling()`, `rank()`, and `select()` (from our standard ordered ST API from CHAPTER 3) for `TrieST`.

**5.2.9** *Extended operations for TSTs.* Implement `keys()` and the extended operations introduced in this section—`longestPrefixOf()`, `keysWithPrefix()`, and `keysThatMatch()`—for TST.

**5.2.10** *Size.* Implement very eager `size()` (that keeps in each node the number of keys in its subtree) for `TrieST` and `TST`.

**5.2.11** *External one-way branching.* Add code to `TrieST` and `TST` to eliminate external one-way branching.

**5.2.12** *Internal one-way branching.* Add code to `TrieST` and `TST` to eliminate internal one-way branching.

**5.2.13** *Hybrid TST with  $R^2$ -way branching at the root.* Add code to `TST` to do multiway branching at the first two levels, as described in the text.

**5.2.14** *Unique substrings of length  $L$ .* Write a `TST` client that reads in text from standard input and calculates the number of unique substrings of length  $L$  that it contains. For example, if the input is `cgcgggcgcg`, then there are five unique substrings of length 3: `cgc`, `cgg`, `gca`, `gac`, and `ggc`. Hint: Use the string method `substring(i, i + L)` to extract the  $i$ th substring, then insert it into a symbol table.

**5.2.15** *Unique substrings.* Write a `TST` client that reads in text from standard input and calculates the number of distinct substrings of any length. This can be done very efficiently with a suffix tree—see CHAPTER 6.

**5.2.16** *Document similarity.* Write a `TST` client with a static method that takes an `int` value  $k$  and two file names as command-line arguments and computes the  $k$ -similarity of the two documents: the Euclidean distance between the frequency vectors defined by the number of occurrences of each  $k$ -gram divided by the number of  $k$ -gram. Include a static method `main()` that takes an `int` value  $k$  as command-line argument and a list of file names from standard input and prints a matrix showing the  $k$ -similarity of all pairs of documents.

**CREATIVE PROBLEMS (continued)**

**5.2.17 Spell checking.** Write a TST client `SpellChecker` that takes as command-line argument the name of a file containing a dictionary of words in the English language, and then reads a string from standard input and prints out any word that is not in the dictionary. Use a string set.

**5.2.18 Whitelist.** Write a TST client that solves the whitelisting problem presented in SECTION 1.1 and revisited in SECTION 3.5 (see page 491).

**5.2.19 Random phone numbers.** Write a `TrieST` client (with  $R = 10$ ) that takes as command line argument an `int` value  $N$  and prints  $N$  random phone numbers of the form (xxx) xxx-xxxx. Use a symbol table to avoid choosing the same number more than once. Use the file `AreaCodes.txt` from the booksite to avoid printing out bogus area codes.

**5.2.20 Contains prefix.** Add a method `containsPrefix()` to `StringSET` (see EXERCISE 5.2.6) that takes a string  $s$  as input and returns `true` if there is a string in the set that contains  $s$  as a prefix.

**5.2.21 Substring matches.** Given a list of (short) strings, your goal is to support queries where the user looks up a string  $s$  and your job is to report back all strings in the list that contain  $s$ . Design an API for this task and develop a TST client that implements your API. Hint: Insert the suffixes of each word (e.g., `string`, `tring`, `ring`, `ing`, `ng`, `g`) into the TST.

**5.2.22 Typing monkeys.** Suppose that a typing monkey creates random words by appending each of 26 possible letter with probability  $p$  to the current word and finishes the word with probability  $1 - 26p$ . Write a program to estimate the frequency distribution of the lengths of words produced. If "abc" is produced more than once, count it only once.

## EXPERIMENTS

**5.2.23 Duplicates (revisited again).** Redo EXERCISE 3.5.30 using StringSET (see EXERCISE 5.2.6) instead of HashSET. Compare the running times of the two approaches. Then use Dedup to run the experiments for  $N = 10^7, 10^8$ , and  $10^9$ , repeat the experiments for random long values and discuss the results.

**5.2.24 Spell checker.** Redo EXERCISE 3.5.31, which uses the file `dictionary.txt` from the booksite and the `BlackFilter` client on page 491 to print all misspelled words in a text file. Compare the performance of `TrieST` and `TST` for the file `WarAndPeace.txt` with this client and discuss the results.

**5.2.25 Dictionary.** Redo EXERCISE 3.5.32: Study the performance of a client like `LookupCSV` (using `TrieST` and `TST`) in a scenario where performance matters. Specifically, design a query-generation scenario instead of taking commands from standard input, and run performance tests for large inputs and large numbers of queries.

**5.2.26 Indexing.** Redo EXERCISE 3.5.33: Study a client like `LookupIndex` (using `TrieST` and `TST`) in a scenario where performance matters. Specifically, design a query-generation scenario instead of taking commands from standard input, and run performance tests for large inputs and large numbers of queries.

## 5.3 SUBSTRING SEARCH

A FUNDAMENTAL OPERATION on strings is *substring search*: given a *text* string of length  $N$  and a *pattern* string of length  $M$ , find an occurrence of the pattern within the text. Most algorithms for this problem can easily be extended to find all occurrences of the pattern in the text, to count the number of occurrences of the pattern in the text, or to provide context (substrings of the text surrounding each occurrence of the pattern).

When you search for a word while using a text editor or a web browser, you are doing substring search. Indeed, the original motivation for this problem was to support such searches. Another classic application is searching for some important pattern in an intercepted communication. A military leader might be interested in finding the pattern `ATTACK AT DAWN` somewhere in an intercepted text message; a hacker might be interested in finding the pattern `Password`: somewhere in your computer's memory. In today's world, we are often searching through the vast amount of information available on the web.

To best appreciate the algorithms, think of the pattern as being relatively short (with  $M$  equal to, say, 100 or 1,000) and the text as being relatively long (with  $N$  equal to, say, 1 million or 1 billion). In substring search, we typically preprocess the pattern in order to be able to support fast searches for that pattern in the text.

Substring search is an interesting and classic problem: several very different (and surprising) algorithms have been discovered that not only provide a spectrum of useful practical methods but also illustrate a spectrum of fundamental algorithm design techniques.



**A short history** The algorithms that we examine have an interesting history; we summarize it here to help place the various methods in perspective.

There is a simple brute-force algorithm for substring search that is in widespread use. While it has a worst-case running time proportional to  $MN$ , the strings that arise in many applications lead to a running time that is (except in pathological cases) proportional to  $M + N$ . Furthermore, it is well-suited to standard architectural features on most computer systems, so an optimized version provides a standard benchmark that is difficult to beat, even with a clever algorithm.

In 1970, S. Cook proved a theoretical result about a particular type of abstract machine that implied the existence of an algorithm that solves the substring search problem in time proportional to  $M + N$  in the worst case. D. E. Knuth and V. R. Pratt laboriously followed through the construction Cook used to prove his theorem (which was not intended to be practical) and refined it into a relatively simple and practical algorithm. This seemed a rare and satisfying example of a theoretical result with immediate (and unexpected) practical applicability. But it turned out that J. H. Morris had discovered virtually the same algorithm as a solution to an annoying problem confronting him when implementing a text editor (he wanted to avoid having to “back up” in the text string). The fact that the same algorithm arose from two such different approaches lends it credibility as a fundamental solution to the problem.

Knuth, Morris, and Pratt didn’t get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered an algorithm that is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm to achieve a noticeable decrease in response time for substring search.

Both the Knuth-Morris-Pratt (KMP) and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used. (In fact, the story goes that an unknown systems programmer found Morris’s algorithm too difficult to understand and replaced it with a brute-force implementation.)

In 1980, M. O. Rabin and R. M. Karp used hashing to develop an algorithm almost as simple as the brute-force algorithm that runs in time proportional to  $M + N$  with very high probability. Furthermore, their algorithm extends to two-dimensional patterns and text, which makes it more useful than the others for image processing.

This story illustrates that the search for a better algorithm is still very often justified; indeed, one suspects that there are still more developments on the horizon even for this classic problem.

### Brute-force substring search

An obvious method for substring search is to check, for each possible position in the text at which the pattern could match, whether it does in fact match. The `search()` method below operates in this way to find the first occurrence of a pattern string `pat` in a text string `txt`. The program keeps one pointer (`i`) into the text and another pointer (`j`) into the pattern. For each `i`, it resets `j` to 0 and increments it until finding a mismatch or the end of the pattern (`j == M`). If we reach the end of the text (`i == N-M+1`) before the end of the pattern, then there is no match: the pattern does not occur in the text. Our convention is to return the value `N` to indicate a mismatch.

```
public static int search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i; // found
    }
    return N; // not found
}
```

#### Brute-force substring search

compares find a mismatch with the first character of the pattern. For example, suppose that you search for the pattern `pattern` in the text of this paragraph. There are 196 characters up to the end of the first occurrence of the pattern, only 7 of which are the character `p` (not including the `p` in pattern). Also, there is only 1 other occurrence of `pa` and no other occurrences of `pat`, so the total number of character compares is  $196 + 7 + 1$ , for an average of 1.041 compares per character in the text. On the other hand, there is no guarantee that the algorithm will always be so efficient. For example, a pattern might begin with a long string of `As`. If it does, and the text also has long strings of `As`, then brute-force substring search will be slow.

<code>i</code>	<code>j</code>	<code>i+j</code>	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
txt	→												
0	2	2	A	B	R	A	← pat						
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

↙  
 entries in black  
 match the text  
 ↙  
 entries in red are  
 mismatches  
 ↙  
 entries in gray are  
 for reference only  
 ↙  
 return i when j is M  
 ↗  
 match

Brute-force substring search

**Proposition M.** Brute-force substring search requires  $\sim NM$  character compares to search for a pattern of length  $M$  in a text of length  $N$ , in the worst case.

**Proof:** A worst-case input is when both pattern and text are all As followed by a B. Then for each of the  $N - M + 1$  possible match positions, all the characters in the pattern are checked against the text, for a total cost of  $M(N - M + 1)$ . Normally  $M$  is very small compared to  $N$ , so the total is  $\sim NM$ .

Such degenerate strings are not likely to appear in English text, but they may well occur in other applications (for example, in binary texts), so we seek better algorithms.

i	j	i+j	0	1	2	3	4	5	6	7	8	9
			txt →	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

Brute-force substring search (worst case)

The alternate implementation at the bottom of this page is instructive. As before, the program keeps one pointer ( $i$ ) into the text and another pointer ( $j$ ) into the pattern. As long as they point to matching characters, both pointers are incremented. This code performs precisely the same character compares as the previous implementation. To understand it,

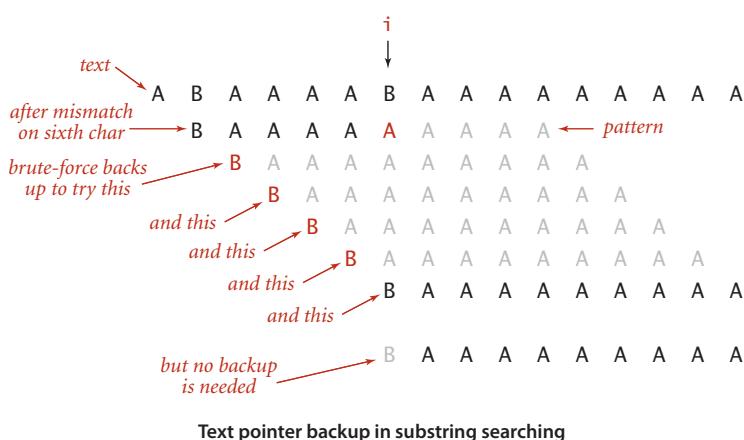
note that  $i$  in this code maintains the value of  $i+j$  in the previous code: it points to the *end* of the sequence of already-matched characters in the text (where  $i$  pointed to the *beginning* of the sequence before). If  $i$  and  $j$  point to mismatching characters, then we *back up* both pointers:  $j$  to point to the beginning of the pattern and  $i$  to correspond to moving the pattern to the right one position for matching against the text.

```
public static int search(String pat, String txt)
{
    int j, M = pat.length();
    int i, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0; }
    }
    if (j == M) return i - M; // found
    else return N; // not found
}
```

Alternate implementation of brute-force substring search (explicit backup)

**Knuth-Morris-Pratt substring search** The basic idea behind the algorithm discovered by Knuth, Morris, and Pratt is this: whenever we detect a mismatch, we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We can take advantage of this information to avoid backing up the text pointer over all those known characters.

As a specific example, suppose that we have a two-character alphabet and are searching for the pattern B A A A A A A A A A. Now, suppose that we match five characters in the pattern, with a mismatch on the sixth. When the mismatch is detected,



we know that the six previous characters in the text must be B A A A A B (the first five match and the sixth does not), with the text pointer now pointing at the B at the end. The key observation is that we need not back up the text pointer  $i$ , since the previous four characters in the text are all As and do not match the first character in the pattern. Furthermore, the character currently pointed to by

$i$  is a B and does match the first character in the pattern, so we can increment  $i$  and compare the next character in the text with the second character in the pattern. This argument leads to the observation that, for this pattern, we can change the `else` clause in the alternate brute-force implementation to just set  $j = 1$  (and not decrement  $i$ ). Since the value of  $i$  does not change within the loop, this method does at most  $N$  character compares. The practical effect of this particular change is limited to this particular pattern, but the idea is worth thinking about—the Knuth-Morris-Pratt algorithm is a generalization of it. Surprisingly, it is *always* possible to find a value to set the  $j$  pointer to on a mismatch, so that the  $i$  pointer is never decremented.

Fully skipping past all the matched characters when detecting a mismatch will not work when the pattern could match itself at any position overlapping the point of the mismatch. For example, when searching for the pattern A A B A A in the text A A B A A B A A A, we first detect the mismatch at position 5, but we had better restart at position 3 to continue the search, since otherwise we would miss the match. The insight of the KMP algorithm is that we can decide ahead of time exactly how to restart the search, because that decision depends only on the pattern.

**Backing up the pattern pointer.** In KMP substring search, we never back up the text pointer  $i$ , and we use an array  $dfa[][]$  to record how far to back up the pattern pointer  $j$  when a mismatch is detected. For every character  $c$ ,  $dfa[c][j]$  is the pattern position to compare against the next text position after comparing  $c$  with  $\text{pat.charAt}(j)$ . During the search,  $dfa[\text{txt.charAt}(i)][j]$  is the pattern position to compare with  $\text{txt.charAt}(i+1)$  after comparing  $\text{txt.charAt}(i)$  with  $\text{pat.charAt}(j)$ . For a match, we want to just move on to the next character, so  $dfa[\text{pat.charAt}(j)][j]$  is always  $j+1$ . For a mismatch, we know not just  $\text{txt.charAt}(i)$ , but also the  $j-1$  previous characters in the text: *they are the first  $j-1$  characters in the pattern*. For each character  $c$ , imagine that we slide a copy of the pattern over these  $j$  characters (the first  $j-1$  characters in the pattern followed by  $c$ —we are deciding what to do when these characters are  $\text{txt.charAt}(i-j+1..i)$ ), from left to right, stopping when all overlapping characters match (or there are none). This gives the next possible place the pattern could match. The index of the pattern character to compare with  $\text{txt.charAt}(i+1)$  ( $dfa[\text{txt.charAt}(i)][j]$ ) is precisely the number of overlapping characters.

**KMP search method.** Once we have computed the  $dfa[][]$  array, we have the substring search method at the top of the next page: when  $i$  and  $j$  point to mismatching characters (testing for a pattern match beginning at position  $i-j+1$  in the text string), then the next possible position for a pattern match is beginning at position  $i-dfa[\text{txt.charAt}(i)][j]$ . But by construction, the first  $dfa[\text{txt.charAt}(i)][j]$  characters at that position match the first  $dfa[\text{txt.charAt}(i)][j]$  characters of the pattern, so there is no need to back up the  $i$  pointer: we can simply set  $j$  to  $dfa[\text{txt.charAt}(i)][j]$  and increment  $i$ , which is precisely what we do when  $i$  and  $j$  point to matching characters.

	$j$	$\text{pat.charAt}(j)$	$dfa[\text{txt.charAt}(i)][j]$	$\text{text (pattern itself)}$
			A    B    C	
0	A	1		A B A B A B C
1	B	2	0	A B A A A B A B C
2	A	3	1	A B A A B B A B A B C
3	B	4	0	A B A B A B A A A B A B C
4	A	5	1	A B A B A A B A B B A B A B C
5	C	6	0	A B A B A C A B A B A B A B A B A B C
				known text chars on mismatch
				mismatch (back up in pattern) → 4
				backup is length of max overlap of beginning of pattern with known text chars

Pattern backup for A B A B A C in KMP substring search

**DFA simulation.** A useful way to describe this process is in terms of a *deterministic finite-state automaton* (DFA). Indeed, as indicated by its name, our `dfa[][]` array precisely defines a DFA. The graphical DFA representation shown at the bottom of this page consists of states (indicated by circled numbers) and transitions (indicated by labeled lines). There is one state for each character in the pattern, each such state having one transition leaving it for each character in the alphabet. For the substring-matching DFAs that we are considering, one of the transitions is a *match* transition (going from  $j$  to  $j+1$ ) and all the others are *mismatch* transitions (going left).

```
public int search(String txt)
{ // Simulate operation of DFA on txt.
    int i, j;
    int N = txt.length(), M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
    if (j == M) return i - M; // found
    else return N; // not found
}
```

KMP substring search (DFA simulation)

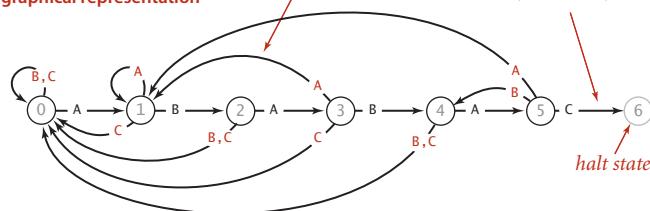
The states correspond to character compares, one for each value of the pattern index. The transitions correspond to changing the value of the pattern index. When examining the text character  $i$  when in the state labeled  $j$ , the machine does the following: “Take the transition to `dfa[txt.charAt(i)][j]` and move to the next character (by incrementing  $i$ ).” For a match transition, we move to the right one position because `dfa[pat.charAt(j)][j]` is always  $j+1$ ; for a mismatch transition we move to the left. The automaton reads the text characters one at a time, from left to right, moving to a new state each time it reads a character. We also include a *halt* state  $M$  that has no transitions. We start the machine at state 0: if the machine reaches state  $M$ , then a substring of the text matching the pattern has been found (and we say that the DFA *recognizes* the pattern); if the machine reaches the end of the text before reaching state  $M$ , then we know the pattern does not appear as a substring of the text. Each pattern corresponds to an automaton (which is represented by the `dfa[][]` array that gives the transitions). The KMP substring search() method is a Java program that simulates the operation of such an automaton.

#### internal representation

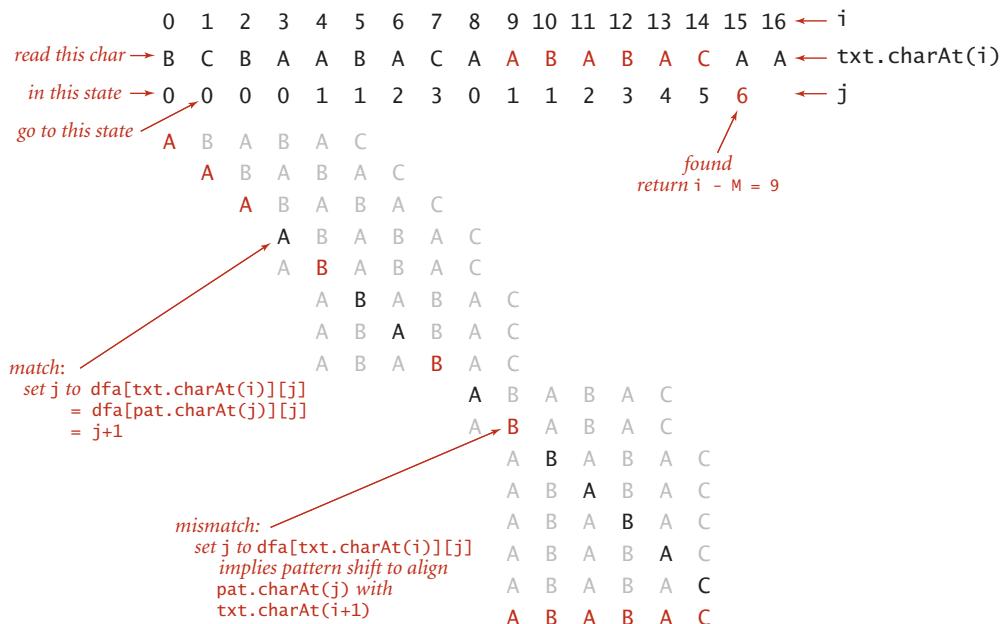
	j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C	
dfa[][][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

*mismatch transition (back up)*      *match transition (increment)*

#### graphical representation



DFA corresponding to the string A B A B A C

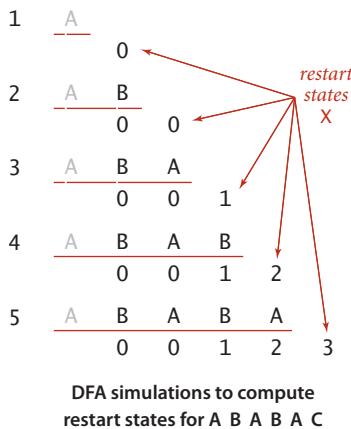


Trace of KMP substring search (DFA simulation) for A B A B A C

To get a feeling for the operation of a substring search DFA, consider two of the simplest things that it does. At the beginning of the process, when started in state 0 at the beginning of the text, it stays in state 0, scanning text characters, until it finds a text character that is equal to the first pattern character, when it moves to the next state and is off and running. At the end of the process, when it finds a match, it matches pattern characters with the right end of the text, incrementing the state until reaching state  $M$ . The trace at the top of this page gives a typical example of the operation of our example DFA. Each match moves the DFA to the next state (which is equivalent to incrementing the pattern index  $j$ ); each mismatch moves the DFA to an earlier state (which is equivalent to setting the pattern index  $j$  to a smaller value). The text index  $i$  marches from left to right, one position at a time, while the pattern index  $j$  bounces around in the pattern as directed by the DFA.

**Constructing the DFA.** Now that you understand the mechanism, we are ready to address the key question for the KMP algorithm: How do we compute the  $\text{dfa}[][]$  array corresponding to a given pattern? Remarkably, the answer to this question lies in the DFA *itself* (!) using the ingenious (and rather tricky) construction that was developed by Knuth, Morris, and Pratt. When we have a mismatch at  $\text{pat.charAt}(j)$ , our interest is in knowing in what state the DFA *would be* if we were to back up the text index and

rescan the text characters that we just saw after shifting to the right one position. We do not want to actually do the backup, just restart the DFA *as if* we had done the backup. The key observation is that the characters in the text that would need to be rescanned



are precisely `pat.charAt(1)` through `pat.charAt(j-1)`: we drop the first character to shift right one position and the last character because of the mismatch. These are pattern characters that we know, so we can figure out ahead of time, for each possible mismatch position, the state where we need to restart the DFA. The figure at left shows the possibilities for our example. *Be sure that you understand this concept.*

What should the DFA do with the next character? Exactly what it would have done if we had backed up, *except* if it finds a match with `pat.charAt(j)`, when it should go to state  $j+1$ . For example, to decide what the DFA should do when we have a mismatch at  $j = 5$  for `A B A B A C`, we use the DFA to learn that a full backup would leave us in state 3 for `B A B A`, so we can copy `dfa[3]` to `dfa[5]`, then set the entry for `C` to 6 because `pat.charAt(5)` is `C` (a match). Since we only need to know how the DFA runs for  $j-1$  characters when we are building the  $j$ th state, we can always get the information that we need from the partially built DFA.

The final crucial detail to the computation is to observe that maintaining the restart position  $X$  when working on column  $j$  of `dfa[][]` is easy because  $X < j$  so that we can use the partially built DFA to do the job—the next value of  $X$  is `dfa[pat.charAt(j)][X]`. Continuing our example from the previous paragraph, we would update the value of  $X$  to `dfa['C'][3] = 0` (but we do not use that value because the DFA construction is complete).

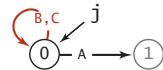
The discussion above leads to the remarkably compact code below for constructing the DFA corresponding to a given pattern. For each  $j$ , it

- Copies `dfa[X]` to `dfa[j]` (for mismatch cases)
- Sets `dfa[pat.charAt(j)][j]` to  $j+1$  (for the match case)
- Updates  $X$

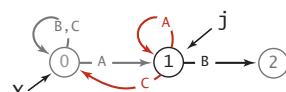
The diagram on the facing page traces this code for our example. To make sure that you understand it, work EXERCISE 5.3.2 and EXERCISE 5.3.3.

```
dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{ // Compute dfa[][]
    for (int c = 0; c < R; c++)
        dfa[c][j] = dfa[c][X];
    dfa[pat.charAt(j)][j] = j+1;
    X = dfa[pat.charAt(j)][X];
}
```

	$j$	0
$\text{pat.charAt}(j)$	A	0
$\text{dfa}[] [j]$	B	0
C	C	0

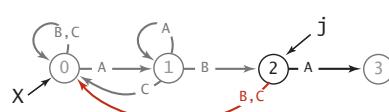


	$X$	
$j$	0	1
$\text{pat.charAt}(j)$	A	B
$\text{dfa}[] [j]$	1	1

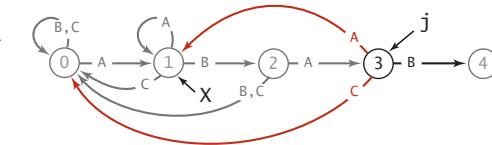


copy  $\text{dfa}[] [X]$  to  $\text{dfa}[] [j]$   
 $\text{dfa}[\text{pat.charAt}(j)] [j] = j+1;$   
 $X = \text{dfa}[\text{pat.charAt}(j)] [X];$

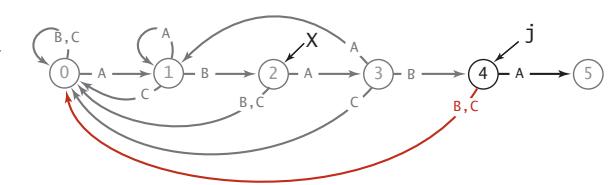
	$X$	
$j$	0	1
$\text{pat.charAt}(j)$	A	B
$\text{dfa}[] [j]$	1	3



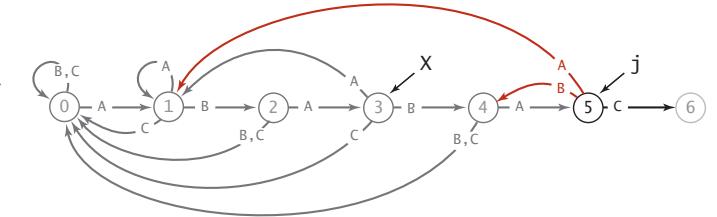
	$X$	
$j$	0	1
$\text{pat.charAt}(j)$	A	B
$\text{dfa}[] [j]$	1	1



	$X$	
$j$	0	1
$\text{pat.charAt}(j)$	A	B
$\text{dfa}[] [j]$	1	1



	$X$	
$j$	0	1
$\text{pat.charAt}(j)$	A	B
$\text{dfa}[] [j]$	1	1



Constructing the DFA for KMP substring search for A B A B A C

**ALGORITHM 5.6 Knuth-Morris-Pratt substring search**

```
public class KMP
{
    private String pat;
    private int[][] dfa;

    public KMP(String pat)
    { // Build DFA from pattern.
        this.pat = pat;
        int M = pat.length();
        int R = 256;
        dfa = new int[R][M];
        dfa[pat.charAt(0)][0] = 1;
        for (int X = 0, j = 1; j < M; j++)
        { // Compute dfa[][]
            for (int c = 0; c < R; c++)
                dfa[c][j] = dfa[c][X]; // Copy mismatch cases.
            dfa[pat.charAt(j)][j] = j+1; // Set match case.
            X = dfa[pat.charAt(j)][X]; // Update restart state.
        }
    }

    public int search(String txt)
    { // Simulate operation of DFA on txt.
        int i, j, N = txt.length(), M = pat.length();
        for (i = 0, j = 0; i < N && j < M; i++)
            j = dfa[txt.charAt(i)][j];
        if (j == M) return i - M; // found (hit end of pattern)
        else return N; // not found (hit end of text)
    }

    public static void main(String[] args)
    // See page 769.
}
```

The constructor in this implementation of the Knuth-Morris-Pratt algorithm for substring search builds a DFA from a pattern string, to support a `search()` method that can find the pattern in a given text string. This program does the same job as the brute-force method, but it runs faster for patterns that are self-repetitive.

```
% java KMP AACAA AABRAACADABRAACAADABRA
text:   AABRAACADABRAACAADABRA
pattern: AACAA
```

ALGORITHM 5.6 on the facing page implements the following API:

---

```
public class KMP
    KMP(String pat)           create a DFA that can search for pat
    int search(String txt)     find index of pat in txt

```

**Substring search API**

---

You can see a typical test client at the bottom of this page. The constructor builds a DFA from a pattern that the `search()` method uses to search for the pattern in a given text.

**Proposition N.** Knuth-Morris-Pratt substring search accesses no more than  $M + N$  characters to search for a pattern of length  $M$  in a text of length  $N$ .

**Proof.** Immediate from the code: we access each pattern character once when computing `dfa[][]` and each text character once (in the worst case) in `search()`.

Another parameter comes into play: for an  $R$ -character alphabet, the total running time (and space) required to build the DFA is proportional to  $MR$ . It is possible to remove the factor of  $R$  by building a DFA where each state has a match transition and a mismatch transition (not transitions for each possible character), though the construction is somewhat more intricate.

The linear-time worst-case guarantee provided by the KMP algorithm is a significant theoretical result. In practice, the speedup over the brute-force method is not often important because few applications involve searching for highly self-repetitive patterns in highly self-repetitive text. Still, the method has the practical advantage that it never backs up in the input. This property makes KMP substring search more convenient for use on an input stream of undetermined length (such as standard input) than algorithms requiring backup, which need some complicated buffering in this situation. Ironically, when backup is easy, we can do significantly better than KMP. Next, we consider a method that generally leads to substantial performance gains precisely because it *can* back up in the text.

```
public static void main(String[] args)
{
    String pat = args[0];
    String txt = args[1];
    KMP kmp = new KMP(pat);
    StdOut.println("text: " + txt);
    int offset = kmp.search(txt);
    StdOut.print("pattern: ");
    for (int i = 0; i < offset; i++)
        StdOut.print(" ");
    StdOut.println(pat);
}
```

**KMP substring search test client**

**Boyer-Moore substring search** When backup in the text string is not a problem, we can develop a significantly faster substring-searching method by scanning the pattern from *right to left* when trying to match it against the text. For example, when searching for the substring BAABBAA, if we find matches on the seventh and sixth characters but not on the fifth, then we can immediately slide the pattern seven positions to the right, and check the 14th character in the text next, because our partial match found XAA where X is not B, which does not appear elsewhere in the pattern. In general, the pattern at the end might appear elsewhere, so we need an array of restart positions as for Knuth-Morris-Pratt. We will not explore this approach in further detail because it is quite similar to our implementation of the Knuth-Morris-Pratt method. Instead, we will consider another suggestion by Boyer and Moore that is typically even more effective in right-to-left pattern scanning.

As with our implementation of KMP substring search, we decide what to do next on the basis of the character that caused the mismatch in the *text* as well as the pattern. The preprocessing step is to decide, for each possible character that could occur in the text, what we would do if that character were to cause the mismatch. The simplest realization of this idea leads immediately to an efficient and useful substring search method.

**Mismatched character heuristic.** Consider the figure at the bottom of this page, which shows a search for the pattern NEEDLE in the text FINDINAHAYSTACKNEEDLE. Proceeding from right to left to match the pattern, we first compare the rightmost E in the pattern with the N (the character at position 5) in the text. Since N appears in the pattern, we slide the pattern five positions to the right to line up the N in the text with the (rightmost) N in the pattern. Then we compare the rightmost E in the pattern with the S (the character at position 10) in the text. This is also a mismatch, but S *does not* appear in the pattern, so we can slide the pattern six positions to the right. We match the rightmost E in the pattern against the E at position 16 in the text, then find a mismatch and discover the N at position 15 and slide to the right four positions, as at the beginning. Finally, we verify, moving from right to left starting at position 20, that the pattern is in the text. This method brings us to the match position at a cost of only four character compares (and six more to verify the match)!

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
text	→																								
0	5	N	E	E	D	L	E																		
5	5							N	E	E	D	L	E												
11	4								N	E	E	D	L	E											
15	0									N	E	E	D	L	E										
<i>return i = 15</i>																									

Mismatched character heuristic for right-to-left (Boyer-Moore) substring search

**Starting point.** To implement the mismatched character heuristic, we use an array `right[]` that gives, for each character in the alphabet, the index of its *rightmost occurrence* in the pattern (or  $-1$  if the character is not in the pattern). This value tells us precisely how far to skip if that character appears in the text and causes a mismatch during the string search. To initialize the `right[]` array, we set all entries to  $-1$  and then, for  $j$  from  $0$  to  $M-1$ , set `right[pat.charAt(j)]` to  $j$ , as shown in the example at right for our example pattern `NEEDLE`.

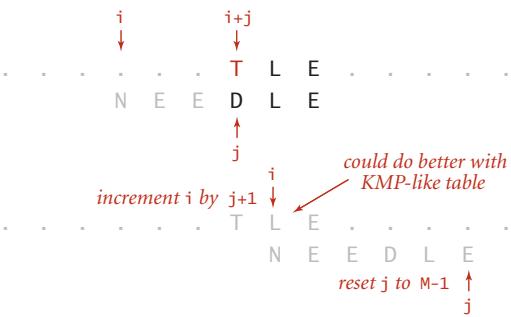
	N	E	E	D	L	E	<u>right[c]</u>
<u>c</u>	0	1	2	3	4	5	
A	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3
E	-1	-1	1	2	2	2	5
...							-1
L	-1	-1	-1	-1	-1	4	4
M	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0
O							-1

## Boyer-Moore skip table computation

**Substring search.** With the `right[]` array pre-computed, the implementation in ALGORITHM 5.7 is straightforward. We have an index  $i$  moving from left to right through the text and an index  $j$  moving from right to left through the pattern. The inner loop tests whether the pattern aligns with the text at position  $i$ . If `txt.charAt(i+j)` is equal to `pat.charAt(j)` for all  $j$  from  $M-1$  down to  $0$ , then there is a match. Otherwise, there is a character mismatch, and we have one of the following three cases:

- If the character causing the mismatch is not found in the pattern, we can slide the pattern  $j+1$  positions to the right (increment  $i$  by  $j+1$ ). Anything less would align that character with some pattern character. Actually, this move aligns some known characters at the beginning of the pattern with known characters at the end of the pattern so that we could further increase  $i$  by precomputing a KMP-like table (see example at right).
  - If the character  $c$  causing the mismatch is found in the pattern, we use the `right[]` array to line up the pattern with the text so that character will match its rightmost occurrence in the pattern. To do so, we increment  $i$  by  $j$  minus `right[c]`. Again, anything less would align that text character with a pattern character it could not match (one to the right of its rightmost occurrence). Again, there is a possibility that we could do better with a KMP-like table, as indicated in the top example in the figure on page 773.

Mismatched character heuristic (mismatch not shown)



Mismatched character heuristic (mismatch not in pattern)

**ALGORITHM 5.7 Boyer-Moore substring search (mismatched character heuristic)**

```
public class BoyerMoore
{
    private int[] right;
    private String pat;

    BoyerMoore(String pat)
    { // Compute skip table.
        this.pat = pat;
        int M = pat.length();
        int R = 256;
        right = new int[R];
        for (int c = 0; c < R; c++)
            right[c] = -1;           // -1 for chars not in pattern
        for (int j = 0; j < M; j++) // rightmost position for
            right[pat.charAt(j)] = j; //   chars in pattern
    }

    public int search(String txt)
    { // Search for pattern in txt.
        int N = txt.length();
        int M = pat.length();
        int skip;
        for (int i = 0; i <= N-M; i += skip)
        { // Does the pattern match the text at position i ?
            skip = 0;
            for (int j = M-1; j >= 0; j--)
                if (pat.charAt(j) != txt.charAt(i+j))
                {
                    skip = j - right[txt.charAt(i+j)];
                    if (skip < 1) skip = 1;
                    break;
                }
            if (skip == 0) return i;           // found.
        }
        return N;                         // not found.
    }

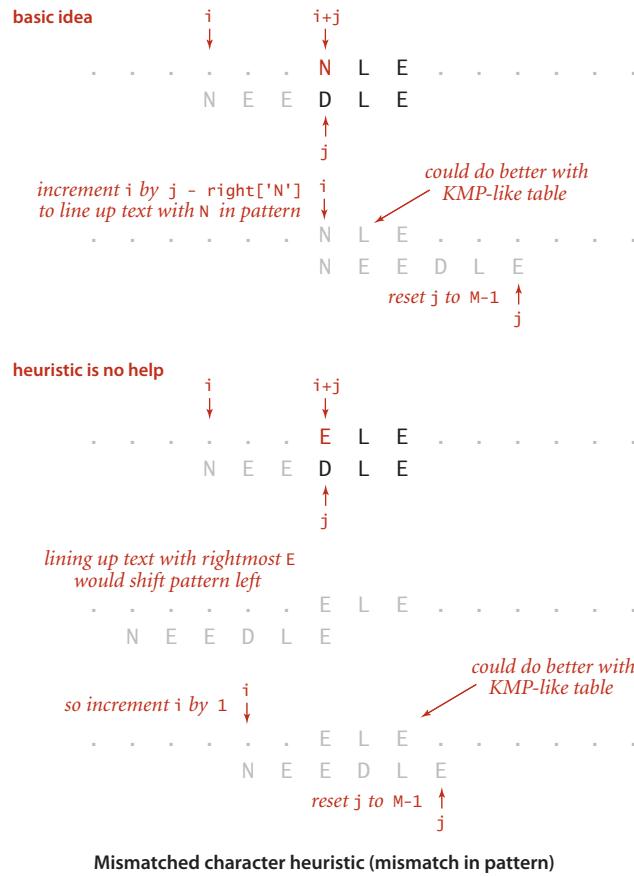
    public static void main(String[] args) // See page 769.
}
```

The constructor in this substring search algorithm builds a table giving the rightmost occurrence in the pattern of each possible character. The search method scans from right to left in the pattern, skipping to align any character causing a mismatch with its rightmost occurrence in the pattern.

- If this computation would not increase  $i$ , we just increment  $i$  instead, to make sure that the pattern always slides at least one position to the right. The bottom example in the figure at right illustrates this situation.

**ALGORITHM 5.7** is a straightforward implementation of this process. Note that the convention of using  $-1$  in the `right[]` array entries corresponding to characters that do not appear in the pattern unifies the first two cases (increment  $i$  by  $j - \text{right}[\text{txt.charAt}(i+j)]$ ).

The full Boyer-Moore algorithm takes into account precomputed mismatches of the pattern with itself (in a manner similar to the KMP algorithm) and provides a linear-time worst-case guarantee (whereas **ALGORITHM 5.7** can take time proportional to  $NM$  in the worst case—see **EXERCISE 5.3.19**). We omit this computation because the mismatched character heuristic controls the performance in typical practical applications.



**Property O.** On typical inputs, substring search with the Boyer-Moore mismatched character heuristic uses  $\sim N/M$  character compares to search for a pattern of length  $M$  in a text of length  $N$ .

**Discussion:** This result can be proved for various random string models, but such models tend to be unrealistic, so we shall skip the details. In many practical situations it is true that all but a few of the alphabet characters appear nowhere in the pattern, so nearly all compares lead to  $M$  characters being skipped, which gives the stated result.

**Rabin-Karp fingerprint search** The method developed by M. O. Rabin and R. M. Karp is a completely different approach to substring search that is based on hashing. We compute a hash function for the pattern and then look for a match by using the same hash function for each possible  $M$ -character substring of the text. If we find a text substring with the same hash value as the pattern, we can check for a match. This process is equivalent to storing the pattern in a hash table, then doing a search for each substring of the text, but we do not need to reserve the memory for the hash table because it would have just one entry. A straightforward implementation based on this description would be much slower than a brute-force search (since computing a hash function that involves every character is likely to be much more expensive than just comparing characters), but Rabin and Karp showed that it is easy to compute hash functions for  $M$ -character substrings in *constant* time (after some preprocessing), which leads to a *linear*-time substring search in practical situations.

**Basic plan.** A string of length  $M$  corresponds to an  $M$ -digit base- $R$  number. To use a hash table of size  $Q$  for keys of this type, we need a hash function to convert an  $M$ -digit base- $R$  number to an `int` value between 0 and  $Q-1$ . Modular hashing (see SECTION 3.4) provides an answer: take the remainder when dividing the number by  $Q$ . In practice, we use a random prime  $Q$ , taking as large a value as possible while avoiding overflow (because we do not actually need to store a hash table). The method is simplest to understand for small  $Q$  and  $R = 10$ , shown in the example below. To find the pattern 2 6 5 3 5 in the text 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3, we choose a table size  $Q$  (997 in the example), compute the hash value  $26535 \% 997 = 613$ , and then look for a match by computing hash values for each five-digit substring in the text. In the example, we get the hash values 508, 201, 715, 971, 442, and 929 before finding the match 613.

Basis for Rabin-Karp substring search														
	<code>pat.charAt(j)</code>													
j	0	1	2	3	4									
	2	6	5	3	5									
	<code>txt.charAt(i)</code>													
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	3	1	4	1	5	9	2	6	5	3	5	8	9	7
0	3	1	4	1	5									
1		1	4	1	5	9								
2			4	1	5	9	2							
3				1	5	9	2	6						
4					5	9	2	6	5					
5						9	2	6	5	3				
6 ← return i = 6							2	6	5	3	5			

**Computing the hash function.** With five-digit values, we could just do all the necessary calculations with `int` values, but what do we do when  $M$  is 100 or 1,000? A simple application of Horner's method, precisely like the method that we examined in SECTION 3.4 for strings and other types of keys with multiple values,

leads to the code shown at right, which computes the hash function for an  $M$ -digit base- $R$  number represented as a `char` array in time proportional to  $M$ . (We pass  $M$  as an argument so that we can use the method for both the pattern and the text, as you will see.) For each digit in the number, we multiply by  $R$ , add the digit, and take the remainder when divided by  $Q$ . For example, computing the hash function for our pattern using this process is shown at the bottom of the page. The same method can work for computing the hash functions in the text, but the cost for the substring search would be a multiplication, addition, and remainder calculation for each text character, for a total of  $NM$  operations in the worst case, no improvement over the brute-force method.

```
private long hash(String key, int M)
{ // Compute hash for key[0..M-1].
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```

Horner's method, applied to modular hashing

**Key idea.** The Rabin-Karp method is based on efficiently computing the hash function for position  $i+1$  in the text, given its value for position  $i$ . It follows directly from a simple mathematical formulation. Using the notation  $t_i$  for `txt.charAt(i)`, the number corresponding to the  $M$ -character substring of `txt` that starts at position  $i$  is

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

and we can assume that we know the value of  $h(x_i) = x_i \bmod Q$ . Shifting one position right in the text corresponds to replacing  $x_i$  by

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}.$$

We subtract off the leading digit, multiply by  $R$ , then add the trailing digit. Now, the crucial point is that we do not have to maintain the values of the numbers, just the values of their remainders when divided by  $Q$ . A fundamental property of the modulus operation is that if we take the remainder when divided by  $Q$  after each arithmetic operation, then we get the same answer as if we were to perform all of the arithmetic operations, then take the remainder

when divided by  $Q$ . We took advantage of this property once before, when implementing modular hashing with Horner's method (see page 460). The result is that we can effectively move right one position in the text in *constant* time, whether  $M$  is 5 or 100 or 1,000.

	pat.charAt(j)				
i	0	1	2	3	4
	2	6	5	3	5
0	2	% 997	= 2		
1	2	6	% 997 = (2*10 + 6) % 997 = 26		
2	2	6	5 % 997 = (26*10 + 5) % 997 = 265		
3	2	6	5 3 % 997 = (265*10 + 3) % 997 = 659		
4	2	6	5 3 5 % 997 = (659*10 + 5) % 997 = 613		

Computing the hash value for the pattern with Horner's method

**Implementation.** This discussion leads directly to the substring search implementation in ALGORITHM 5.8. The constructor computes a hash value `patHash` for the pattern; it also computes the value of  $R^{M-1} \bmod Q$  in the variable `RM`. The `hashSearch()` method begins by computing the hash function for the first  $M$  characters of the text and comparing that value against the hash value for the pattern. If that is not a match, it proceeds through the text string, using the technique above to maintain the hash function for the  $M$  characters starting at position  $i$  for each  $i$  in a variable `txtHash` and comparing each new hash value to `patHash`. (An extra  $Q$  is added during the `txtHash` calculation to make sure that everything stays positive so that the remainder operation works as it should.)

**A trick: Monte Carlo correctness.** After finding a hash value for an  $M$ -character substring of `txt` that matches the pattern hash value, you might expect to see code to compare those characters with the pattern to ensure that we have a true match, not just a hash collision. We do not do that test because using it requires backup in the text string. Instead, we make the hash table “size”  $Q$  as large as we wish, since we are not actually building a hash table, just testing for a collision with one key, our pattern. We will use a `long` value greater than  $10^{20}$ , making the probability that a random key hashes to the

i	...	2	3	4	5	6	7	...
	<i>current value</i>	1	4	1	5	9	2	6
	<i>new value</i>	4	1	5	9	2	6	5
								→ text
		4	1	5	9	2		<i>current value</i>
-		4	0	0	0	0		
			1	5	9	2		<i>subtract leading digit</i>
			*	1	0			<i>multiply by radix</i>
		1	5	9	2	0		
						+	6	<i>add new trailing digit</i>
		1	5	9	2	6		<i>new value</i>

Key computation in Rabin-Karp substring search  
(move right one position in the text)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3	%	997	=	3												
1	3	1	%	997	=	$(3 * 10 + 1) \% 997 = 31$											
2	3	1	4	%	997	=	$(31 * 10 + 4) \% 997 = 314$										
3	3	1	4	1	%	997	=	$(314 * 10 + 1) \% 997 = 150$									
4	3	1	4	1	5	%	997	=	$(150 * 10 + 5) \% 997 = 508$								
5	1	4	1	5	9	%	997	=	$((508 + 3 * (997 - 30)) * 10 + 9) \% 997 = 201$								
6	4	1	5	9	2	%	997	=	$((201 + 1 * (997 - 30)) * 10 + 2) \% 997 = 715$								
7	1	5	9	2	6	%	997	=	$((715 + 4 * (997 - 30)) * 10 + 6) \% 997 = 971$								
8	5	9	2	6	5	%	997	=	$((971 + 1 * (997 - 30)) * 10 + 5) \% 997 = 442$								
9	9	2	6	5	3	%	997	=	$((442 + 5 * (997 - 30)) * 10 + 3) \% 997 = 929$								
10	←	<i>return i-M+1 = 6</i>	2	6	5	3	5	%	997	=	$((929 + 9 * (997 - 30)) * 10 + 5) \% 997 = 613$						

Rabin-Karp substring search example

**ALGORITHM 5.8** Rabin-Karp fingerprint substring search

```

public class RabinKarp
{
    private String pat;           // pattern (only needed for Las Vegas)
    private long patHash;         // pattern hash value
    private int M;                // pattern length
    private long Q;               // a large prime
    private int R = 256;           // alphabet size
    private long RM;              //  $R^{M-1} \% Q$ 

    public RabinKarp(String pat)
    {
        this.pat = pat;          // save pattern (needed only for Las Vegas)
        M = pat.length();
        Q = longRandomPrime();      // See Exercise 5.3.33.
        RM = 1;
        for (int i = 1; i <= M-1; i++) // Compute  $R^{M-1} \% Q$  for use
            RM = (R * RM) % Q;       // in removing leading digit.
        pathash = hash(pat, M);
    }

    public boolean check(int i) // Monte Carlo (See text.)
    { return true; } // For Las Vegas, check pat vs txt(i..i-M+1).

    private long hash(String key, int M)
    // See text (page 775).
    private int search(String txt)
    { // Search for hash match in text.
        int N = txt.length();
        long txtHash = hash(txt, M);
        if (patHash == txtHash && check(0)) return 0; // match
        for (int i = M; i < N; i++)
        { // Remove leading digit, add trailing digit, check for match.
            txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
            txtHash = (txtHash*R + txt.charAt(i)) % Q;
            if (patHash == txtHash)
                if (check(i - M + 1)) return i - M + 1; // match
        }
        return N; // no match
    }
}

```

This substring search algorithm is based on hashing. It computes a hash value for the pattern in the constructor, then searches through the text looking for a hash match.

same value as our pattern less than  $10^{-20}$ , an exceedingly small value. If that value is not small enough for you, you could run the algorithms again to get a probability of failure of less than  $10^{-40}$ . This algorithm is an early and famous example of a *Monte Carlo* algorithm that has a guaranteed completion time but fails to output a correct answer with a small probability. The alternative method of checking for a match could be slow (it might amount to the brute-force algorithm, with a very small probability) but is guaranteed correct. Such an algorithm is known as a *Las Vegas* algorithm.

**Property P.** The Monte Carlo version of Rabin-Karp substring search is linear-time and extremely likely to be correct, and the Las Vegas version of Rabin-Karp substring search is correct and extremely likely to be linear-time.

**Discussion:** The use of the very large value of  $Q$ , made possible by the fact that we need not maintain an actual hash table, makes it extremely unlikely that a collision will occur. Rabin and Karp showed that when  $Q$  is properly chosen, we get a hash collision for random strings with probability  $1/Q$ , which implies that, for practical values of the variables, there are no hash matches when there are no substring matches and only one hash match if there is a substring match. Theoretically, a text position could lead to a hash collision and not a substring match, but in practice it can be relied upon to find a match.

If your belief in probability theory (or in the random string model and the code we use to generate random numbers) is more half-hearted than resolute, you can add to `check()` the code to check that the text matches the pattern, which turns ALGORITHM 5.8 into the Las Vegas version of the algorithm (see EXERCISE 5.3.12). If you also add a check to see whether that code ever returns a negative value, you might develop more faith in probability theory as time wears on.

RABIN-KARP SUBSTRING SEARCH is known as a *fingerprint* search because it uses a small amount of information to represent a (potentially very large) pattern. Then it looks for this fingerprint (the hash value) in the text. The algorithm is efficient because the fingerprints can be efficiently computed and compared.

**Summary** The table at the bottom of the page summarizes the algorithms that we have discussed for substring search. As is often the case when we have several algorithms for the same task, each of them has attractive features. Brute-force search is easy to implement and works well in typical cases (Java’s `indexOf()` method in `String` uses brute-force search); Knuth-Morris-Pratt is guaranteed linear-time with no backup in the input; Boyer-Moore is sublinear (by a factor of  $M$ ) in typical situations; and Rabin-Karp is linear. Each also has drawbacks: brute-force might require time proportional to  $MN$ ; Knuth-Morris-Pratt and Boyer-Moore use extra space; and Rabin-Karp has a relatively long inner loop (several arithmetic operations, as opposed to character compares in the other methods). These characteristics are summarized in the table below.

algorithm	version	operation count guarantee	operation count typical	backup in input?	correct?	extra space
<i>brute force</i>	—	$MN$	$1.1 N$	yes	yes	1
<i>Knuth-Morris-Pratt</i>	<i>full DFA</i> (Algorithm 5.6)	$2N$	$1.1 N$	no	yes	$MR$
	<i>mismatch transitions only</i>	$3N$	$1.1 N$	no	yes	$M$
	<i>full algorithm</i>	$3N$	$N/M$	yes	yes	$R$
<i>Boyer-Moore</i>	<i>mismatched char heuristic only</i> (Algorithm 5.7)	$MN$	$N/M$	yes	yes	$R$
	<i>Monte Carlo</i> (Algorithm 5.8)	$7N$	$7N$	no	$yes^\dagger$	1
<i>Rabin-Karp</i> <sup>†</sup>	<i>Las Vegas</i>	$7N^\dagger$	$7N$	yes	yes	1

<sup>†</sup> probabilistic guarantee, with uniform and independent hash function

**Q&A**

**Q.** This substring search problem seems like a bit of a toy problem. Do I really need to understand these complicated algorithms?

**A.** Well, the factor of  $M$  speedup available with Boyer-Moore can be quite impressive in practice. Also, the ability to stream input (no backup) leads to many practical applications for KMP and Rabin-Karp. Beyond these direct practical applications, this topic provides an interesting introduction to the use of abstract machines and randomization in algorithm design.

**Q.** Why not simplify things by converting each character to binary, treating all text as binary text?

**A.** That idea is not quite effective because of false matches across character boundaries.

**EXERCISES**

**5.3.1** Develop a brute-force substring search implementation `Brute`, using the same API as ALGORITHM 5.6.

**5.3.2** Give the `dfa[][]` array for the Knuth-Morris-Pratt algorithm for the pattern A A A A A A A A, and draw the DFA, in the style of the figures in the text.

**5.3.3** Give the `dfa[][]` array for the Knuth-Morris-Pratt algorithm for the pattern A B R A C A D A B R A, and draw the DFA, in the style of the figures in the text.

**5.3.4** Write an efficient method that takes a string `txt` and an integer `M` as arguments and returns the position of the first occurrence of `M` consecutive blanks in the string, `txt.length` if there is no such occurrence. Estimate the number of character compares used by your method, on typical text and in the worst case.

**5.3.5** Give the `right[]` array computed by the constructor in ALGORITHM 5.7 for the pattern A A B A A B A A B C D A C A B.

**5.3.6** Give the `right[]` array computed by the constructor in ALGORITHM 5.7 for the pattern A B R A C A D A B R A.

**5.3.7** Add to our brute-force implementation of substring search a `count()` method to count occurrences and a `searchAll()` method to print all occurrences.

**5.3.8** Add to KMP a `count()` method to count occurrences and a `searchAll()` method to print all occurrences.

**5.3.9** Add to BoyerMoore a `count()` method to count occurrences and a `searchAll()` method to print all occurrences.

**5.3.10** Add to RabinKarp a `count()` method to count occurrences and a `searchAll()` method to print all occurrences.

**5.3.11** Construct a worst-case example for the Boyer-Moore implementation in ALGORITHM 5.7 (which demonstrates that it is not linear-time).

**5.3.12** Add the code to `check()` in RabinKarp (ALGORITHM 5.8) that turns it into a Las Vegas algorithm (check that the pattern matches the text at the position given as argument).

**5.3.13** In the Boyer-Moore implementation in ALGORITHM 5.7, show that you can set

**EXERCISES (continued)**

`right[c]` to the penultimate occurrence of `c` when `c` is the last character in the pattern.

**5.3.14** Develop versions of the substring search implementations in this section that use `char[]` instead of `String` to represent the pattern and the text.

**5.3.15** Develop a brute-force substring search implementation `BruteForceRL` that processes the pattern from right to left (a simplified version of ALGORITHM 5.7).

**5.3.16** Show the trace of the brute-force algorithm in the style of the figures in the text for the following pattern and text strings

- pattern: AAAAAAAB text: AAAAAAAAAAAAAAAAB
- pattern: ABABABAB text: ABABABABAABABABAABAAA

**5.3.17** Draw the KMP DFA for the following pattern strings.

- AAAAAAB
- AACAAAB
- ABABABAB
- ABAABAAABAAAB
- ABAABCABAABCB

**5.3.18** Suppose that the pattern and text are *random* strings over an alphabet of size  $R$  (which is at least 2). Show that the expected number of character compares for the brute-force method is  $(N - M + 1) (1 - R^{-M}) / (1 - R^{-1}) \leq 2(N - M + 1)$ .

**5.3.19** Construct an example where the Boyer-Moore algorithm (with only the mismatched character heuristic) performs poorly.

**5.3.20** How would you modify the Rabin-Karp algorithm to determine whether any of a subset of  $k$  patterns (say, all of the same length) is in the text?

*Solution:* Compute the hashes of the  $k$  patterns and store the hashes in a `StringSET` (see EXERCISE 5.2.6).

**5.3.21** How would you modify the Rabin-Karp algorithm to search for a given pattern with the additional proviso that the middle character is a “wildcard” (any text character

at all can match it).

**5.3.22** How would you modify the Rabin-Karp algorithm to search for an  $H$ -by- $V$  pattern in an  $N$ -by- $N$  text?

**5.3.23** Write a program that reads characters one at a time and reports at each instant if the current string is a palindrome. *Hint:* Use the Rabin-Karp hashing idea.

## CREATIVE PROBLEMS

**5.3.24** *Find all occurrences.* Add a method `findAll()` to each of the four substring search algorithms given in the text that returns an `Iterable<Integer>` that allows clients to iterate through all offsets of the pattern in the text.

**5.3.25** *Streaming.* Add a `search()` method to `KMP` that takes variable of type `In` as argument, and searches for the pattern in the specified input stream *without* using any extra instance variables. Then do the same for `RabinKarp`.

**5.3.26** *Cyclic rotation check.* Write a program that, given two strings, determines whether one is a cyclic rotation of the other, such as `example` and `ampleex`.

**5.3.27** *Tandem repeat search.* A tandem repeat of a base string `t` in a string `s` is a substring of `s` having at least two consecutive copies `t` (nonoverlapping). Develop and implement a linear-time algorithm that, given two strings `s` and `t`, returns the index of the beginning of the longest tandem repeat of `t` in `s`. For example, your program should return 3 when `t` is `abcab` and `s` is `abcababcababcababcab`.

**5.3.28** *Buffering in brute-force search.* Add a `search()` method to your solution to EXERCISE 5.3.1 that takes an input stream (of type `In`) as argument and searches for the pattern in the given input stream. *Note:* You need to maintain a buffer that can keep at least the previous `M` characters in the input stream. Your challenge is to write efficient code to initialize, update, and clear the buffer for any input stream.

**5.3.29** *Buffering in Boyer-Moore.* Add a `search()` method to ALGORITHM 5.7 that takes an input stream (of type `In`) as argument and searches for the pattern in the given input stream.

**5.3.30** *Two-dimensional search.* Implement a version of the Rabin-Karp algorithm to search for patterns in two-dimensional text. Assume both pattern and text are rectangles of characters.

**5.3.31** *Random patterns.* How many character compares are needed to do a substring search for a random pattern of length 100 in a given text?

*Answer:* None. The method

```
public boolean search(char[] txt)
{   return false; }
```

is quite effective for this problem, since the chances of a random pattern of length 100 appearing in any text are so low that you may consider it to be 0.

**5.3.32 Unique substrings.** Solve EXERCISE 5.2.14 using the idea behind the Rabin-Karp method.

**5.3.33 Random primes.** Implement `longRandomPrime()` for `RabinKarp` (ALGORITHM 5.8). Hint: A random  $n$ -digit number is prime with probability proportional to  $1/n$ .

**5.3.34 Straight-line code.** The Java Virtual Machine (and your computer’s assembly language) support a `goto` instruction so that the search can be “wired in” to machine code, like the program at right (which is exactly equivalent to simulating the DFA for the pattern as in `KMPdfa`, but likely to be much more efficient). To avoid checking whether the end of the text has been reached each time `i` is incremented, we assume that the pattern itself is stored at the end of the text as a sentinel, as the last  $M$  characters of the text. The `goto` labels in this code correspond precisely to the `dfa[]` array. Write a static method that takes a pattern as input and produces as output a straight-line program like this that searches for the pattern.

```
int i = -1;
sm: i++;
s0: if (txt[i]) != 'A' goto sm;
s1: if (txt[i]) != 'A' goto s0;
s2: if (txt[i]) != 'B' goto s0;
s3: if (txt[i]) != 'A' goto s2;
s4: if (txt[i]) != 'A' goto s0;
s5: if (txt[i]) != 'A' goto s3;
return i-8;
```

Straight-line substring search for A A B A A A

**5.3.35 Boyer-Moore in binary strings.** The mismatched character heuristic does not help much for binary strings, because there are only two possibilities for characters that cause the mismatch (and these are both likely to be in the pattern). Develop a substring search class for binary strings that groups bits together to make “characters” that can be used exactly as in ALGORITHM 5.7. Note: If you take  $b$  bits at a time, then you need a `right[]` array with  $2^b$  entries. The value of  $b$  should be chosen small enough so that this table is not too large, but large enough that most  $b$ -bit sections of the text are not likely to be in the pattern—there are  $M-b+1$  different  $b$ -bit sections in the pattern (one starting at each bit position from 1 through  $M-b+1$ ), so we want  $M-b+1$  to be significantly less than  $2^b$ . For example, if you take  $b$  to be about  $\lg(4M)$ , then the `right[]` array will be more than three-quarters filled with  $-1$  entries, but do not let  $b$  become greater than  $M/2$ , since otherwise you could miss the pattern entirely, if it were split between two  $b$ -bit text sections.

## EXPERIMENTS

**5.3.36** *Random text.* Write a program that takes integers  $M$  and  $N$  as arguments, generates a random binary text string of length  $N$ , then counts the number of other occurrences of the last  $M$  bits elsewhere in the string. *Note:* Different methods may be appropriate for different values of  $M$ .

**5.3.37** *KMP for random text.* Write a client that takes integers  $M$ ,  $N$ , and  $T$  as input and runs the following experiment  $T$  times: Generate a random pattern of length  $M$  and a random text of length  $N$ , counting the number of character compares used by KMP to search for the pattern in the text. Instrument KMP to provide the number of compares, and print the average count for the  $T$  trials.

**5.3.38** *Boyer-Moore for random text.* Answer the previous exercise for BoyerMoore.

**5.3.39** *Timings.* Write a program that times the four methods for the task of searching for the substring

it is a far far better thing that i do than i have ever done

in the text of *Tale of Two Cities* (*tale.txt*). Discuss the extent to which your results validate the hypotheses about performance that are stated in the text.

*This page intentionally left blank*



## 5.4 REGULAR EXPRESSIONS

IN MANY APPLICATIONS, we need to do substring searching with somewhat less than complete information about the pattern to be found. A user of a text editor may wish to specify only part of a pattern, or to specify a pattern that could match a few different words, or to specify that any one of a number of patterns would do. A biologist might search for a genomic sequence satisfying certain conditions. In this section we will consider how pattern matching of this type can be done efficiently.

The algorithms in the previous section fundamentally depend on complete specification of the pattern, so we have to consider different methods. The basic mechanisms we will consider make possible a very powerful string-searching facility that can match complicated  $M$ -character patterns in  $N$ -character text strings in time proportional to  $MN$  in the worst case, and much faster for typical applications.

First, we need a way to describe the patterns: a rigorous way to specify the kinds of partial-substring-searching problems suggested above. This specification needs to involve more powerful primitive operations than the “check if the  $i$ th character of the text string matches the  $j$ th character of the pattern” operation used in the previous section. For this purpose, we use *regular expressions*, which describe patterns in combinations of three natural, basic, and powerful operations.

Programmers have used regular expressions for decades. With the explosive growth of search opportunities on the web, their use is becoming even more widespread. We will discuss a number of specific applications at the beginning of the section, not only to give you a feeling for their utility and power, but also to enable you to become more familiar with their basic properties.

As with the KMP algorithm in the previous section, we consider the three basic operations in terms of an abstract machine that can search for patterns in a text string. Then, as before, our pattern-matching algorithm will construct such a machine and then simulate its operation. Naturally, pattern-matching machines are typically more complicated than KMP DFAs, but not as complicated as you might expect.

As you will see, the solution we develop to the pattern-matching problem is intimately related to fundamental processes in computer science. For example, the method we will use in our program to perform the string-searching task implied by a given pattern description is akin to the method used by the Java system to transform a given Java program into a machine-language program for your computer. We also encounter the concept of *nondeterminism*, which plays a critical role in the search for efficient algorithms (see CHAPTER 6).

**Describing patterns with regular expressions** We focus on pattern descriptions made up of characters that serve as operands for three fundamental operations. In this context, we use the word *language* specifically to refer to a set of strings (possibly infinite) and the word *pattern* to refer to a language specification. The rules that we consider are quite analogous to familiar rules for specifying arithmetic expressions.

**Concatenation.** The first fundamental operation is the one used in the last section. When we write  $A B$ , we are specifying the language  $\{AB\}$  that has one two-character string, formed by concatenating  $A$  and  $B$ .

**Or.** The second fundamental operation allows us to specify alternatives in the pattern. If we have an *or* between two alternatives, then both are in the language. We will use the vertical bar symbol  $|$  to denote this operation. For example,  $A | B$  specifies the language  $\{A, B\}$  and  $A | E | I | O | U$  specifies the language  $\{A, E, I, O, U\}$ . Concatenation has higher precedence than *or*, so  $A B | B C D$  specifies the language  $\{AB, BCD\}$ .

**Closure.** The third fundamental operation allows parts of the pattern to be repeated arbitrarily. The *closure* of a pattern is the language of strings formed by concatenating the pattern with itself any number of times (including zero). We denote closure by placing a  $*$  after the pattern to be repeated. Closure has higher precedence than concatenation, so  $A B^*$  specifies the language consisting of strings with an  $A$  followed by 0 or more  $B$ s, while  $A^* B$  specifies the language consisting of strings with 0 or more  $A$ s followed by a  $B$ . The *empty string*, which we denote by  $\epsilon$ , is found in every text string (and in  $A^*$ ).

**Parentheses.** We use parentheses to override the default precedence rules. For example,  $C (A C | B) D$  specifies the language  $\{CACD, CBD\}$ ;  $(A | C) ((B | C) D)$  specifies the language  $\{ABD, CBD, ACD, CCD\}$ ; and  $(A B)^*$  specifies the language of strings formed by concatenating any number of occurrences of  $AB$ , including no occurrences:  $\{\epsilon, AB, ABAB, \dots\}$ .

These simple rules allow us to write down REs that, while complicated, clearly and completely describe languages (see the table at right for a few examples). Often, a language can be simply described in some other way, but discovering such a description can be a challenge. For example, the RE in the bottom row of the table specifies the subset of  $(A | B)^*$  with an even number of  $B$ s.

RE	matches	does not match
$(A   B)(C   D)$	AC AD BC BD	<i>every other string</i>
$A(B   C)^*D$	AD ABD ACD ABCCBD	BCD ADD ABCBC
$A^*   (A^*BA^*BA^*)^*$	AAA BBAABB BABAAA	ABA BBB BABBAAA

Examples of regular expressions

REGULAR EXPRESSIONS ARE EXTREMELY SIMPLE formal objects, even simpler than the arithmetic expressions that you learned in grade school. Indeed, we will take advantage of their simplicity to develop compact and efficient algorithms for processing them. Our starting point will be the following formal definition:

**Definition.** A regular expression (RE) is either

- The empty set  $\emptyset$
- The empty string  $\epsilon$
- A single character
- A regular expression enclosed in parentheses
- Two or more *concatenated* regular expressions
- Two or more regular expressions separated by the *or* operator ( $|$ )
- A regular expression followed by the *closure* operator ( $*$ )

This definition describes the *syntax* of regular expressions, telling us what constitutes a legal regular expression. The *semantics* that tells us the meaning of a given regular expression is the point of the informal descriptions that we have given in this section. For review, we summarize these by continuing the formal definition:

**Definition (continued).** Each RE represents a set of strings, defined as follows:

- The empty set  $\emptyset$  represents the set of strings with 0 elements.
- The empty string  $\epsilon$  represents the set of strings with one element, the string with zero characters.
- A single character represents the set of strings with one element, itself.
- An RE enclosed in parentheses represents the same set of strings as the RE without the parentheses.
- The RE consisting of two *concatenated* REs represents the *cross product* of the sets of strings represented by the individual components (all possible strings that can be formed by taking one string from each and concatenating them, in the same order as the REs).
- The RE consisting of the *or* of two REs represents the *union* of the sets represented by the individual components.
- The RE consisting of the *closure* of an RE represents  $\epsilon$  or the union of the sets represented by the concatenation of any number of copies of the RE.

There are many different ways to describe each language: we must try to specify succinct patterns just as we try to write compact programs and implement efficient algorithms.

**Shortcuts** Typical applications adopt various additions to these basic rules to enable us to develop succinct descriptions of languages of practical interest. From a theoretical standpoint, these are each simply a shortcut for a sequence of operations involving many operands; from a practical standpoint, they are a quite useful extension to the basic operations that enable us to develop compact patterns.

**Set-of-characters descriptors.** It is often convenient to be able to use a single character or a short sequence to directly specify sets of characters. The dot character (.) is a *wildcard* that represents any single character. A sequence of characters within square brackets represents any one of those characters. The sequence may also be specified as a range of characters. If preceded by a ^, a sequence within square brackets represents any character *but* one of those characters. These notations are simply shortcuts for a sequence of *or* operations.

**Closure shortcuts.** The closure operator specifies any number of copies of its operand. In practice, we want the flexibility to specify the number of copies, or a range on the number. In particular, we use the plus sign (+) to specify at least one copy, the question mark (?) to specify zero or one copy, and a count or a range within braces ({} ) to specify a given number of copies. Again, these notations are shortcuts for a sequence of the basic concatenation, *or*, and closure operations.

**Escape sequences.** Some characters, such as \, ., |, \*, (, and ), are *metacharacters* that we use to form regular expressions. We use *escape sequences* that begin with a backslash character \ separating metacharacters from characters in the alphabet. An escape sequence may be a \ followed by a single metacharacter (which represents that character). For example, \\ represents \. Other escape sequences represent special characters and whitespace. For example, \t represents a tab character, \n represents a newline, and \s represents any whitespace character.

	name	notation	example
	wildcard	.	A.B
	specified set	enclosed in []	[AEIOU]*
	range	enclosed in [] separated by -	[A-Z] [0-9]
	complement	enclosed in [] preceded by ^	[^AEIOU]*

#### Set-of-characters descriptors

option	notation	example	shortcut for	in language	not in language
at least 1	+	(AB)+	(AB)(AB)*	AB ABABAB	$\epsilon$ BBBAAA
0 or 1	?	(AB)?	$\epsilon$   AB	$\epsilon$ AB	any other string
specific	count in {}	(AB){3}	(AB)(AB)(AB)	ABABAB	any other string
range	range in {}	(AB){1-2}	(AB)   (AB)(AB)	AB ABAB	any other string

#### Closure shortcuts (for specifying the number of copies of the operand)

**REs in applications** REs have proven to be remarkably versatile in describing languages that are relevant in practical applications. Accordingly, REs are heavily used and have been heavily studied. To familiarize you with regular expressions while at the same time giving you some appreciation for their utility, we consider a number of practical applications before addressing the RE pattern-matching algorithm. REs also play an important role in theoretical computer science. Discussing this role to the extent it deserves is beyond the scope of this book, but we sometimes allude to relevant fundamental theoretical results.

**Substring search.** Our general goal is to develop an algorithm that determines whether a given text string is in the set of strings described by a given regular expression. If a text is in the language described by a pattern, we say that the text *matches* the pattern. Pattern matching with REs vastly generalizes the substring search problem of SECTION 5.3. Precisely, to search for a substring *pat* in a text string *txt* is to check whether *txt* is in the language described by the pattern  $.^* \text{pat}.^*$  or not.

**Validity checking.** You frequently encounter RE matching when you use the web. When you type in a date or an account number on a commercial website, the input-processing program has to check that your response is in the right format. One approach to performing such a check is to write code that checks all the cases: if you were to type in a dollar amount, the code might check that the first symbol is a \$, that the \$ is followed by a set of digits, and so forth. A better approach is to define an RE that describes the set of all legal inputs. Then, checking whether your input is legal is precisely the pattern-matching problem: is your input in the language described by the RE? Libraries of REs for common checks have sprung up on the web as this type of checking has come into widespread use. Typically, an RE is a much more precise and concise expression of the set of all valid strings than would be a program that checks all the cases.

context	regular expression	matches
<i>substring search</i>	$.^* \text{NEEDLE}.^*$	A HAYSTACK NEEDLE IN
<i>phone number</i>	$\backslash([0-9]\{3\}\backslash)\backslash [0-9]\{3\}-[0-9]\{4\}$	(800) 867-5309
<i>Java identifier</i>	$[\$_A\text{-}Za\text{-}z][\$_A\text{-}Za\text{-}z0\text{-}9]^*$	Pattern_Matcher
<i>genome marker</i>	$\text{gcg}(\text{cg}\text{g} \text{agg})^*\text{ctg}$	gcgaggaggcggcggctg
<i>email address</i>	$[\text{a-z}]+\text{@}([\text{a-z}]+\text{\.})+(\text{edu} \text{com})$	rs@cs.princeton.edu

Typical regular expressions in applications (simplified versions)

**Programmer’s toolbox.** The origin of regular expression pattern matching is the Unix command `grep`, which prints all lines matching a given RE. This capability has proven invaluable for generations of programmers, and REs are built into many modern programming systems, from `awk` and `emacs` to Perl, Python, and JavaScript. For example, suppose that you have a directory with dozens of `.java` files, and you want to know which of them has code that uses `StdIn`. The command

```
% grep StdIn *.java
```

will immediately give the answer. It prints all lines that match `.*StdIn.*` for each file.

**Genomics.** Biologists use REs to help address important scientific problems. For example, the human gene sequence has a region that can be described with the RE `gcg(cgg)*ctg`, where the number of repeats of the `cgg` pattern is highly variable among individuals, and a certain genetic disease that can cause mental retardation and other symptoms is known to be associated with a high number of repeats.

**Search.** Web search engines support REs, though not always in their full glory. Typically, if you want to specify alternatives with `|` or repetition with `*`, you can do so.

**Possibilities.** A first introduction to theoretical computer science is to think about the set of languages that can be specified with an RE. For example, you might be surprised to know that you can implement the modulus operation with an RE: for example, `(0 | 1(01*0)*1)*` describes all strings of 0s and 1s that are the binary representations of numbers that are multiples of three (!): `11, 110, 1001`, and `1100` are in the language, but `10, 1011`, and `10000` are not.

**Limitations.** Not all languages can be specified with REs. A thought-provoking example is that no RE can describe the set of all strings that specify legal REs. Simpler versions of this example are that we cannot use REs to check whether parentheses are balanced or to check whether a string has an equal number of As and Bs.

THESE EXAMPLES JUST SCRATCH THE SURFACE. Suffice it to say that REs are a useful part of our computational infrastructure and have played an important role in our understanding of the nature of computation. As with KMP, the algorithm that we describe next is a byproduct of the search for that understanding.

**Nondeterministic finite-state automata** Recall that we can view the Knuth-Morris-Pratt algorithm as a finite-state machine constructed from the search pattern that scans the text. For regular expression pattern matching, we will generalize this idea.

The finite-state automaton for KMP changes from state to state by looking at a character from the text string and then changing to another state, depending on the character. The automaton reports a match if and only if it reaches the accept state. The algorithm itself is a simulation of the automaton. The characteristic of the machine that makes it easy to simulate is that it is *deterministic*: each state transition is completely determined by the next character in the text.

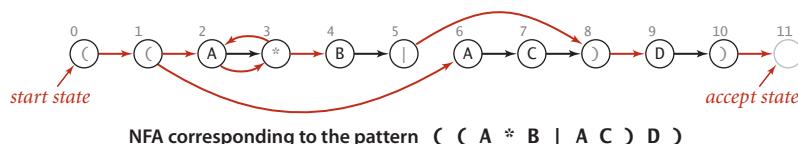
To handle regular expressions, we consider a more powerful abstract machine. Because of the *or* operation, the automaton cannot determine whether or not the pattern could occur at a given point by examining just one character; indeed, because of closure, it cannot even determine *how many* characters might need to be examined before a mismatch is discovered. To overcome these problems, we will endow the automaton with the power of *nondeterminism*: when faced with more than one way to try to match the pattern, the machine can “guess” the right one! This power might seem to you to be impossible to realize, but we will see that it is easy to write a program to build a *nondeterministic finite-state automaton* (NFA) and to efficiently simulate its operation. The overview of our RE pattern matching algorithm is nearly the same as for KMP:

- Build the NFA corresponding to the given RE.
- Simulate the operation of that NFA on the given text.

*Kleene’s Theorem*, a fundamental result of theoretical computer science, asserts that there is an NFA corresponding to any given RE (and vice versa). We will consider a constructive proof of this fact that will demonstrate how to transform any RE into an NFA; then we simulate the operation of the NFA to complete the job.

Before we consider how to build pattern-matching NFAs, we will consider an example that illustrates their properties and the basic rules for operating them. Consider the figure below, which shows an NFA that determines whether a text string is in the language described by the RE  $((A^*B|AC)D)$ . As illustrated in this example, the NFAs that we define have the following characteristics:

- The NFA corresponding to an RE of length  $M$  has exactly one state per pattern character, starts at state 0, and has a (virtual) accept state  $M$ .



- States corresponding to a character from the alphabet have an outgoing edge that goes to the state corresponding to the next character in the pattern (black edges in the diagram).
- States corresponding to the metacharacters  $($ ,  $)$ ,  $|$ , and  $*$  have at least one outgoing edge (red edges in the diagram), which may go to any other state.
- Some states have multiple outgoing edges, but no state has more than one outgoing black edge.

By convention, we enclose all patterns in parentheses, so the first state corresponds to a left parenthesis and the final state corresponds to a right parenthesis (and has a transition to the accept state).

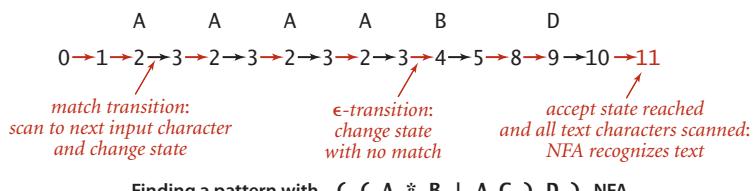
As with the DFAs of the previous section, we start the NFA at state 0, reading the first character of a text. The NFA moves from state to state, sometimes reading text characters, one at a time, from left to right. However, there are some basic differences from DFAs:

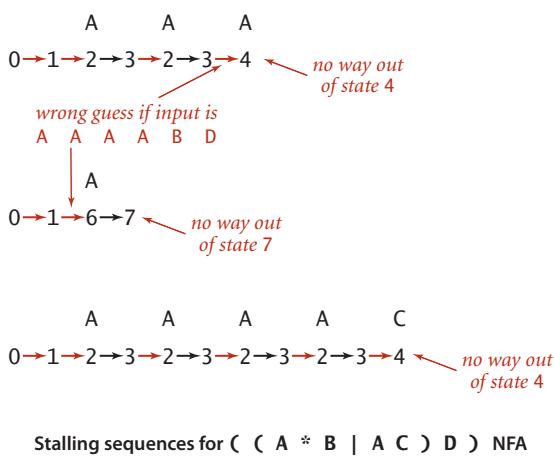
- Characters appear in the nodes, not the edges, in the diagrams.
- Our NFA recognizes a text string only after explicitly reading all its characters, whereas our DFA recognizes a pattern in a text without necessarily reading all the text characters.

These differences are not critical—we have picked the version of each machine that is best suited to the algorithms that we are studying.

Our focus now is on checking whether the text matches the pattern—for that, we need the machine to reach its accept state and consume all the text. The rules for moving from one state to another are also different than for DFAs—an NFA can do so in one of two ways:

- If the current state corresponds to a character in the alphabet *and* the current character in the text string matches the character, the automaton can scan past the character in the text string and take the (black) transition to the next state. We refer to such a transition as a *match transition*.
- The automaton can follow any red edge to another state without scanning any text character. We refer to such a transition as an  $\epsilon$ -*transition*, referring to the idea that it corresponds to “matching” the empty string  $\epsilon$ .





For example, suppose that our NFA for  $((A^* B \mid A C) D)$  is started (at state 0) with the text A A A A B D as input. The figure at the bottom of the previous page shows a sequence of state transitions ending in the accept state. This sequence demonstrates that the text is in the set of strings described by the RE—the text *matches* the pattern. With respect to the NFA, we say that the NFA *recognizes* that text.

The examples shown at left illustrate that it is also possible to find transition sequences that cause the NFA to stall, even for input text such as A A A A B D that it should recognize. For example, if the NFA takes the transition to state

4 before scanning all the As, it is left with nowhere to go, since the only way out of state 4 is to match a B. These two examples demonstrate the nondeterministic nature of the automaton. After scanning an A and finding itself in state 3, the NFA has two choices: it could go on to state 4 or it could go back to state 2. The choices make the difference between getting to the accept state (as in the first example just discussed) or stalling (as in the second example just discussed). This NFA also has a choice to make at state 1 (whether to take an  $\epsilon$ -transition to state 2 or to state 6).

These examples illustrate the key difference between NFAs and DFAs: since an NFA may have multiple edges leaving a given state, the transition from such a state is *not deterministic*—it might take one transition at one point in time and a different transition at a different point in time, without scanning past any text character. To make some sense of the operation of such an automaton, imagine that an NFA has the power to *guess* which transition (if any) will lead to the accept state for the given text string. In other words, we say that *an NFA recognizes a text string if and only if there is some sequence of transitions that scans all the text characters and ends in the accept state when started at the beginning of the text in state 0*. Conversely, an NFA does not recognize a text string if and only if there is no sequence of match transitions and  $\epsilon$ -transitions that can scan all the text characters and lead to the accept state for that string.

As with DFAs, we have been tracing the operation of the NFA on a text string simply by listing the sequence of state changes, ending in the final state. Any such sequence is a proof that the machine recognizes the text string (there may be other proofs). But how do we find such a sequence for a given text string? And how do we prove that there is no such sequence for another given text string? The answers to these questions are easier than you might think: we systematically try all possibilities.

**Simulating an NFA** The idea of an automaton that can guess the state transitions it needs to get to the accept state is like writing a program that can guess the right answer to a problem: it seems ridiculous. On reflection, you will see that the task is conceptually not at all difficult: we make sure that we check all possible sequences of state transitions, so if there is one that gets to the accept state, we will find it.

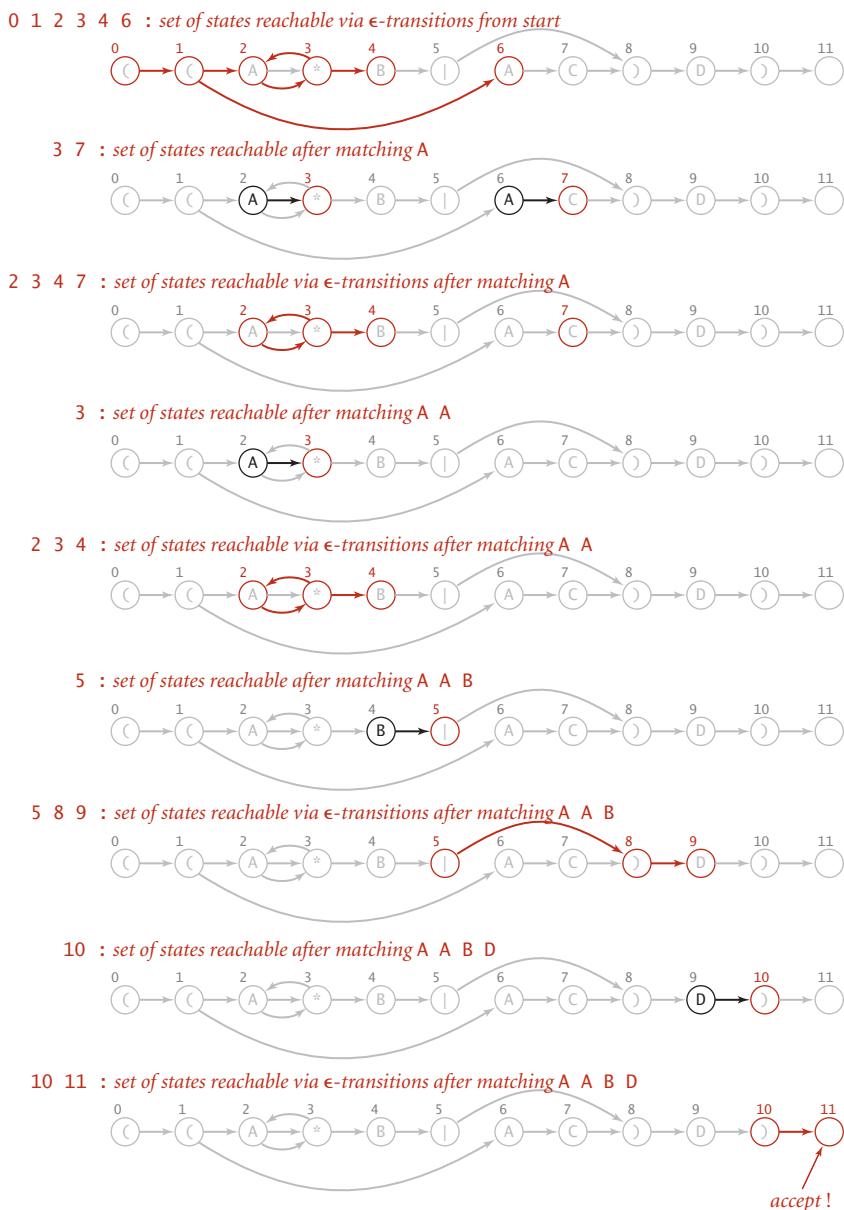
**Representation.** To begin, we need an NFA representation. The choice is clear: the RE itself gives the state names (the integers between 0 and M, where M is the number of characters in the RE). We keep the RE itself in an array `re[]` of `char` values that defines the match transitions (if `re[i]` is in the alphabet, then there is a match transition from i to `i+1`). The natural representation for the  $\epsilon$ -transitions is a *digraph*—they are directed edges (red edges in our diagrams) connecting vertices between 0 and M (one for each state). Accordingly, we represent all the  $\epsilon$ -transitions as a digraph G. We will consider the task of building the digraph associated with a given RE after we consider the simulation process. For our example, the digraph consists of the nine edges

$$0 \rightarrow 1 \quad 1 \rightarrow 2 \quad 1 \rightarrow 6 \quad 2 \rightarrow 3 \quad 3 \rightarrow 2 \quad 3 \rightarrow 4 \quad 5 \rightarrow 8 \quad 8 \rightarrow 9 \quad 10 \rightarrow 11$$

**NFA simulation and reachability.** To simulate an NFA, we keep track of the *set* of states that could possibly be encountered while the automaton is examining the current input character. The key computation is the familiar *multiple-source reachability* computation that we addressed in ALGORITHM 4.4 (page 571). To initialize this set, we find the set of states reachable via  $\epsilon$ -transitions from state 0. For each such state, we check whether a match transition for the first input character is possible. This check gives us the set of possible states for the NFA just after matching the first input character. To this set, we add all states that could be reached via  $\epsilon$ -transitions from one of the states in the set. Given the set of possible states for the NFA just after matching the first character in the input, the solution to the multiple-source reachability problem in the  $\epsilon$ -transition digraph gives the set of states that could lead to match transitions for the *second* character in the input. For example, the initial set of states for our example NFA is 0 1 2 3 4 6; if the first character is an A, the NFA could take a match transition to 3 or 7; then it could take  $\epsilon$ -transitions from 3 to 2 or 3 to 4, so the set of possible states that could lead to a match transition for the second character is 2 3 4 7. Iterating this process until all text characters are exhausted leads to one of two outcomes:

- The set of possible states contains the accept state.
- The set of possible states does not contain the accept state.

The first of these outcomes indicates that there is some sequence of transitions that takes the NFA to the accept state, so we report success. The second of these outcomes indicates that the NFA always stalls on that input, so we report failure. With our SET



Simulation of  $( (A^* B \mid A C) D )$  NFA for input A A B D

data type and the `DirectedDFS` class just described for computing multiple-source reachability in a digraph, the NFA simulation code given below is a straightforward translation of the English-language description just given. You can check your understanding of the code by following the trace on the facing page, which illustrates the full simulation for our example.

**Proposition Q.** Determining whether an  $N$ -character text string is recognized by the NFA corresponding to an  $M$ -character RE takes time proportional to  $NM$  in the worst case.

**Proof:** For each of the  $N$  text characters, we iterate through a set of states of size no more than  $M$  and run a DFS on the digraph of  $\epsilon$ -transitions. The construction that we will consider next establishes that the number of edges in that digraph is no more than  $2M$ , so the worst-case time for each DFS is proportional to  $M$ .

Take a moment to reflect on this remarkable result. This worst-case cost, the product of the text and pattern lengths, is *the same* as the worst-case cost of finding an exact substring match using the elementary algorithm that we started with at the beginning of SECTION 5.3.

```
public boolean recognizes(String txt)
{ // Does the NFA recognize txt?
    Bag<Integer> pc = new Bag<Integer>();
    DirectedDFS dfs = new DirectedDFS(G, 0);
    for (int v = 0; v < G.V(); v++)
        if (dfs.marked(v)) pc.add(v);

    for (int i = 0; i < txt.length(); i++)
    { // Compute possible NFA states for txt[i+1].
        Bag<Integer> match = new Bag<Integer>();
        for (int v : pc)
            if (v < M)
                if (re[v] == txt.charAt(i) || re[v] == '.')
                    match.add(v+1);
        pc = new Bag<Integer>();
        dfs = new DirectedDFS(G, match);
        for (int v = 0; v < G.V(); v++)
            if (dfs.marked(v)) pc.add(v);
    }

    for (int v : pc) if (v == M) return true;
    return false;
}
```

NFA simulation for pattern matching

**Building an NFA corresponding to an RE** From the similarity between regular expressions and familiar arithmetic expressions, you may not be surprised to find that translating an RE to an NFA is somewhat similar to the process of evaluating an arithmetic expression using Dijkstra’s two-stack algorithm, which we considered in SECTION 1.3. The process is a bit different because

- REs do not have an explicit operator for concatenation
- REs have a unary operator, for closure (\*)
- REs have only one binary operator, for *or* (|)

Rather than dwell on the differences and similarities, we will consider an implementation that is tailored for REs. For example, we need only one stack, not two.

From the discussion of the representation at the beginning of the previous subsection, we need only build the digraph  $G$  that consists of all the  $\epsilon$ -transitions. The RE itself and the formal definitions that we considered at the beginning of this section provide precisely the information that we need. Taking a cue from Dijkstra’s algorithm, we will use a stack to keep track of the positions of left parentheses and *or* operators.

**Concatenation.** In terms of the NFA, the concatenation operation is the simplest to implement. Match transitions for states corresponding to characters in the alphabet explicitly implement concatenation.

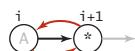
**Parentheses.** We push the RE index of each left parenthesis on the stack. Each time we encounter a right parenthesis, we eventually pop the corresponding left parentheses from the stack in the manner described below. As in Dijkstra’s algorithm, the stack enables us to handle nested parentheses in a natural manner.

**Closure.** A closure (\*) operator must occur either (i) after a single character, when we add  $\epsilon$ -transitions to and from the character, or (ii) after a right parenthesis, when we add  $\epsilon$ -transitions to and from the corresponding left parenthesis, the one at the top of the stack.

**Or expression.** We process an RE of the form  $(A \mid B)$  where  $A$  and  $B$  are both REs by adding two  $\epsilon$ -transitions: one from the state corresponding to the left parenthesis to the state corresponding to the first character of  $B$  and one from the state corresponding to the | operator to the state corresponding to the right parenthesis. We push the RE index corresponding the | operator onto the stack (as well as the index corresponding to the left parenthesis, as described above) so that the information we need is at the top of the stack when needed, at the time we reach the right parenthesis. These  $\epsilon$ -transitions allow the NFA to choose one of the two alternatives. We do not add an  $\epsilon$ -transition from the state corresponding to the | operator to the state with the next higher index, as we have for all other states—the only way for the NFA to leave such a state is to take a transition to the state corresponding to the right parenthesis.

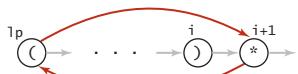
THESE SIMPLE RULES SUFFICE TO build NFAs corresponding to arbitrarily complicated REs. ALGORITHM 5.9 is an implementation whose constructor builds the  $\epsilon$ -transition digraph corresponding to a given RE, and a trace of the construction for our example appears on the following page. You can find other examples at the bottom of this page and in the exercises and are encouraged to enhance your understanding of the process by working your own examples. For brevity and for clarity, a few details (handling metacharacters, set-of-character descriptors, closure shortcuts, and multiway or operations) are left for exercises (see EXERCISES 5.4.16 through 5.4.21). Otherwise, the construction requires remarkably little code and represents one of the most ingenious algorithms that we have seen.

#### single-character closure



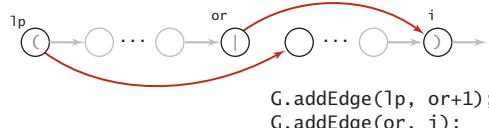
```
G.addEdge(i, i+1);
G.addEdge(i+1, i);
```

#### closure expression



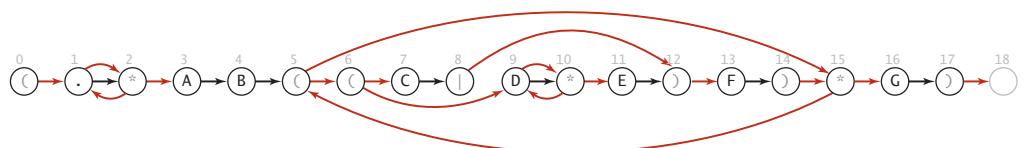
```
G.addEdge(lp, i+1);
G.addEdge(i+1, lp);
```

#### or expression



```
G.addEdge(lp, or+1);
G.addEdge(or, i);
```

#### NFA construction rules



NFA corresponding to the pattern  $(\cdot^* A B ((C \mid D^* E) F)^* G)$

**ALGORITHM 5.9** Regular expression pattern matching (grep)

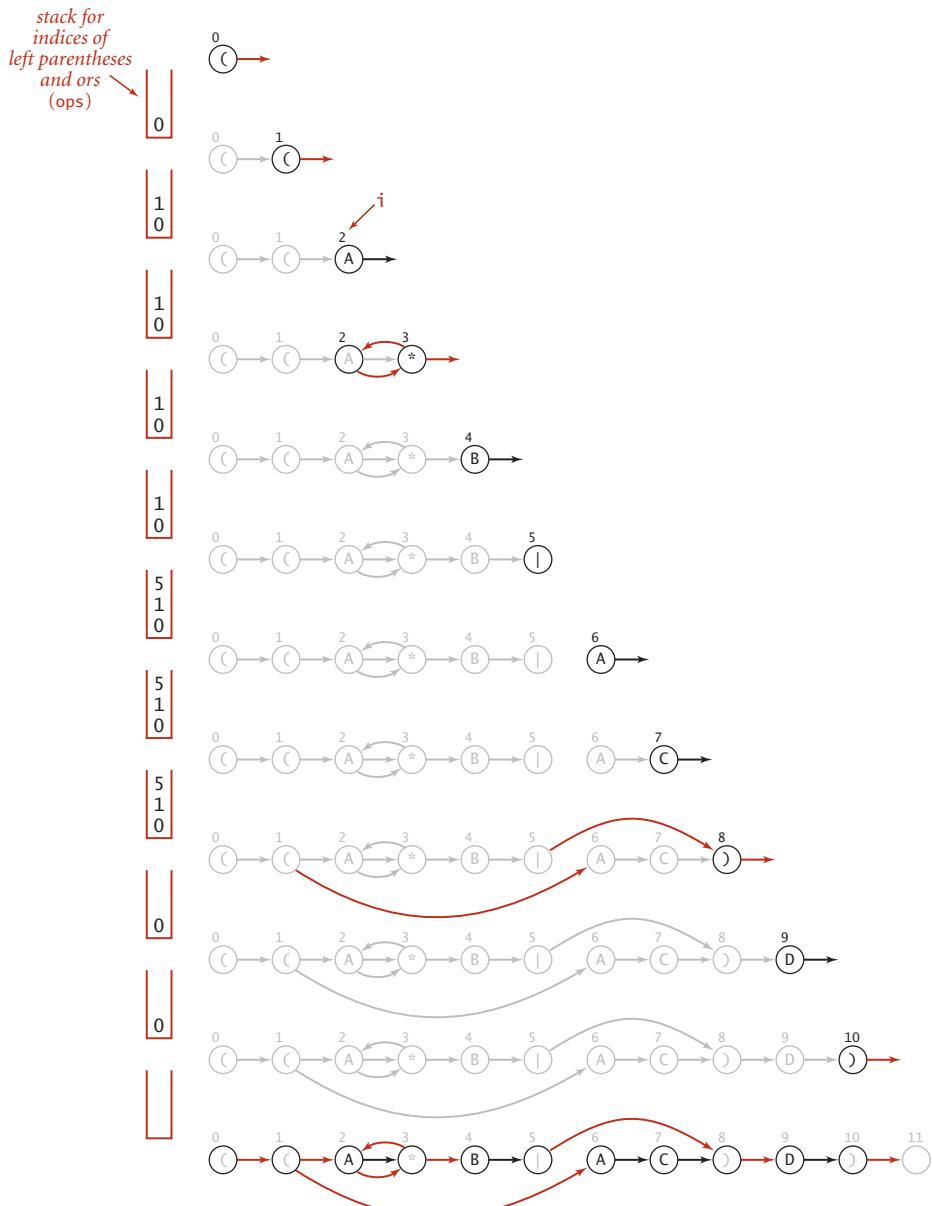
```
public class NFA
{
    private char[] re;           // match transitions
    private Digraph G;          // epsilon transitions
    private int M;              // number of states

    public NFA(String regexp)
    { // Create the NFA for the given regular expression.
        Stack<Integer> ops = new Stack<Integer>();
        re = regexp.toCharArray();
        M = re.length;
        G = new Digraph(M+1);

        for (int i = 0; i < M; i++)
        {
            int lp = i;
            if (re[i] == '(' || re[i] == '|')
                ops.push(i);
            else if (re[i] == ')')
            {
                int or = ops.pop();
                if (re[or] == '|')
                {
                    lp = ops.pop();
                    G.addEdge(lp, or+1);
                    G.addEdge(or, i);
                }
                else lp = or;
            }
            if (i < M-1 && re[i+1] == '*') // lookahead
            {
                G.addEdge(lp, i+1);
                G.addEdge(i+1, lp);
            }
            if (re[i] == '(' || re[i] == '*' || re[i] == ')')
                G.addEdge(i, i+1);
        }
    }

    public boolean recognizes(String txt)
    // Does the NFA recognize txt? (See page 799.)
}
```

This constructor builds an NFA corresponding to a given RE by creating a digraph of  $\epsilon$ -transitions.



**Proposition R.** Building the NFA corresponding to an  $M$ -character RE takes time and space proportional to  $M$  in the worst case.

**Proof.** For each of the  $M$  RE characters in the regular expression, we add at most three  $\epsilon$ -transitions and perhaps execute one or two stack operations.

The classic GREP client for pattern matching, illustrated in the code at left, takes an RE as argument and prints the lines from standard input having some *substring* that is in the language described by the RE.

This client was a feature in the early implementations of Unix and has been an indispensable tool for generations of programmers.

```
public class GREP
{
    public static void main(String[] args)
    {
        String regexp = "(.*" + args[0] + ".*)";
        NFA nfa = new NFA(regexp);
        while (StdIn.hasNextLine())
        {
            String txt = StdIn.readLine();
            if (nfa.recognizes(txt))
                StdOut.println(txt);
        }
    }
}
```

Classic Generalized Regular Expression Pattern-matching NFA client

```
% more tinyL.txt
AC
AD
AAA
ABD
ADD
BCD
ABCCBD
BABAAA
BABBAAA

% java GREP "(A*B|AC)D" < tinyL.txt
ABD
ABCCBD

% java GREP StdIn < GREP.java
      while (StdIn.hasNextLine())
          String txt = StdIn.readLine();
```

**Q&A**

**Q.** What is the difference between  $\emptyset$  and  $\epsilon$ ?

**A.** The former denotes an empty *set*; the latter denotes an empty *string*. You can have a set that contains one element,  $\epsilon$ , and is therefore not an empty set  $\emptyset$ .

## EXERCISES

**5.4.1** Give regular expressions that describe all strings that contain

- Exactly four consecutive As
- No more than four consecutive As
- At least one occurrence of four consecutive As

**5.4.2** Give a brief English description of each of the following REs:

- a.  $\cdot^*$
- b.  $A \cdot^* A \mid A$
- c.  $\cdot^* A B B A B B A \cdot^*$
- d.  $\cdot^* A \cdot^* A \cdot^* A \cdot^* A \cdot^*$

**5.4.3** What is the maximum number of different strings that can be described by a regular expression with *M or* operators and no closure operators (parentheses and concatenation are allowed)?

**5.4.4** Draw the NFA corresponding to the pattern  $((A \mid B)^* \mid C D^* \mid E F G)^*$ .

**5.4.5** Draw the digraph of  $\epsilon$ -transitions for the NFA from EXERCISE 5.4.4.

**5.4.6** Give the sets of states reachable by your NFA from EXERCISE 5.4.4 after each character match and subsequent  $\epsilon$ -transitions for the input **ABBA C E F G E F G C A A B**.

**5.4.7** Modify the GREP client on page 804 to be a client **GREPmatch** that encloses the pattern in parentheses but does *not* add  $\cdot^*$  before and after the pattern, so that it prints out only those lines that are strings in the language described by the given RE. Give the result of typing each of the following commands:

- a. % java GREPmatch "(A|B)(C|D)" < tinyL.txt
- b. % java GREPmatch "A(B|C)^\*D" < tinyL.txt
- c. % java GREPmatch "(A^\*B|AC)D" < tinyL.txt

**5.4.8** Write a regular expression for each of the following sets of binary strings:

- a. Contains at least three consecutive 1s
- b. Contains the substring 110
- c. Contains the substring 1101100
- d. Does not contain the substring 110

**5.4.9** Write a regular expression for binary strings with at least two 0s but not consecutive 0s.

**5.4.10** Write a regular expression for each of the following sets of binary strings:

- a. Has at least 3 characters, and the third character is 0
- b. Number of 0s is a multiple of 3
- c. Starts and ends with the same character
- d. Odd length
- e. Starts with 0 and has odd length, or starts with 1 and has even length
- f. Length is at least 1 and at most 3

**5.4.11** For each of the following regular expressions, indicate how many bitstrings of length exactly 1,000 match:

- a.  $0(0 \mid 1)^*1$
- b.  $0^*101^*$
- c.  $(1 \mid 01)^*$

**5.4.12** Write a Java regular expression for each of the following:

- a. Phone numbers, such as (609) 555-1234
- b. Social Security numbers, such as 123-45-6789
- c. Dates, such as December 31, 1999
- d. IP addresses of the form a.b.c.d where each letter can represent one, two, or three digits, such as 196.26.155.241
- e. License plates that start with four digits and end with two uppercase letters

## CREATIVE PROBLEMS

**5.4.13 Challenging REs.** Construct an RE that describes each of the following sets of strings over the binary alphabet:

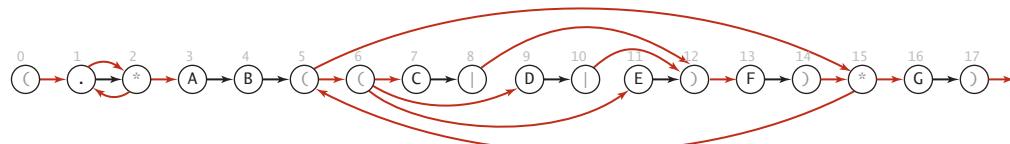
- All strings except 11 or 111
- Strings with 1 in every odd-number bit position
- Strings with at least two 0s and at most one 1
- Strings with no two consecutive 1s

**5.4.14 Binary divisibility.** Construct an RE that describes all binary strings that when interpreted as a binary number are

- Divisible by 2
- Divisible by 3
- Divisible by 123

**5.4.15 One-level REs.** Construct a Java RE that describes the set of strings that are legal REs over the binary alphabet, but with no occurrence of parentheses within parentheses. For example,  $(0.*1)^*$  |  $(1.*0)^*$  is in this language, but  $(1(0 \mid 1)1)^*$  is not.

**5.4.16 Multiway or.** Add multiway *or* to NFA. Your code should produce the machine drawn below for the pattern  $(\cdot^* A B ((C \mid D \mid E) F)^* G)$ .



NFA corresponding to the pattern  $(\cdot^* A B ((C \mid D \mid E) F)^* G)$

- 5.4.17** *Wildcard.* Add to NFA the capability to handle wildcards.
- 5.4.18** *One or more.* Add to NFA the capability to handle the + closure operator.
- 5.4.19** *Specified set.* Add to NFA the capability to handle specified-set descriptors.
- 5.4.20** *Range.* Add to NFA the capability to handle range descriptors.
- 5.4.21** *Complement.* Add to NFA the capability to handle complement descriptors.
- 5.4.22** *Proof.* Develop a version of NFA that prints a *proof* that a given string is in the language recognized by the NFA (a sequence of state transitions that ends in the accept state).



## 5.5 DATA COMPRESSION

The world is awash with data, and algorithms designed to represent data efficiently play an important role in the modern computational infrastructure. There are two primary reasons to compress data: to save storage when saving information and to save time when communicating information. Both of these reasons have remained important through many generations of technology and are familiar today to anyone needing a new storage device or waiting for a long download.

You have certainly encountered compression when working with digital images, sound, movies, and all sorts of other data. The algorithms we will examine save space by exploiting the fact that most data files have a great deal of redundancy: For example, text files have certain character sequences that appear much more often than others; bitmap files that encode pictures have large homogeneous areas; and files for the digital representation of images, movies, sound, and other analog signals have large repeated patterns.

We will look at an elementary algorithm and two advanced methods that are widely used. The compression achieved by these methods varies depending on characteristics of the input. Savings of 20 to 50 percent are typical for text, and savings of 50 to 90 percent might be achieved in some situations. As you will see, the effectiveness of any data compression method is quite dependent on characteristics of the input. *Note:* Usually, in this book, we are referring to time when we speak of performance; with data compression we normally are referring to the compression they can achieve, although we will also pay attention to the time required to do the job.

On the one hand, data-compression techniques are less important than they once were because the cost of computer storage devices has dropped dramatically and far more storage is available to the typical user than in the past. On the other hand, data-compression techniques are more important than ever because, since so much storage is in use, the savings they make possible are greater. Indeed, data compression has come into widespread use with the emergence of the internet, because it is a low-cost way to reduce the time required to transmit large amounts of data.

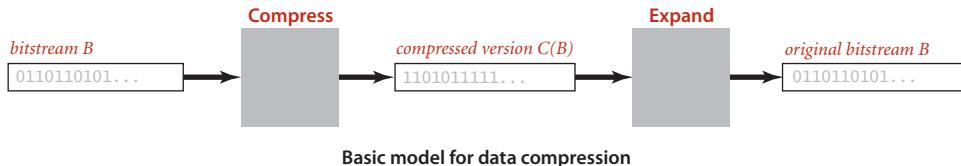
Data compression has a rich history (we will only be providing a brief introduction to the topic), and contemplating its role in the future is certainly worthwhile. Every student of algorithms can benefit from studying data compression because the algorithms are classic, elegant, interesting, and effective.

**Rules of the game** All of the types of data that we process with modern computer systems have something in common: *they are ultimately represented in binary*. We can consider each of them to be simply a sequence of bits (or bytes). For brevity, we use the term *bitstream* in this section to refer to a sequence of bits and *bytestream* when we are referring to the bits being considered as a sequence of fixed-size bytes. A bitstream or a bytestream might be stored as a file on your computer, or it might be a message being transmitted on the internet.

**Basic model.** Accordingly, our basic model for data compression is quite simple, having two primary components, each a black box that reads and writes bitstreams:

- A *compress* box that transforms a bitstream  $B$  into a compressed version  $C(B)$
- An *expand* box that transforms  $C(B)$  back into  $B$

Using the notation  $|B|$  to denote the number of bits in a bitstream, we are interested in minimizing the quantity  $|C(B)| / |B|$ , which is known as the *compression ratio*.



This model is known as *lossless compression*—we insist that no information be lost, in the specific sense that the result of compressing and expanding a bitstream must match the original, bit for bit. Lossless compression is required for many types of files, such as numerical data or executable code. For some types of files (such as images, videos, or music), it is reasonable to consider compression methods that are allowed to lose some information, so the decoder only produces an approximation of the original file. Lossy methods have to be evaluated in terms of a subjective quality standard in addition to the compression ratio. We do not address lossy compression in this book.

**Reading and writing binary data** A full description of how information is encoded on your computer is system-dependent and is beyond our scope, but with a few basic assumptions and two simple APIs, we can separate our implementations from these details. These APIs, `BinaryStdIn` and `BinaryStdOut`, are modeled on the `StdIn` and `StdOut` APIs that you have been using, but their purpose is to read and write *bits*, where `StdIn` and `StdOut` are oriented toward *character streams* encoded in Unicode. An `int` value on `StdOut` is a sequence of characters (its decimal representation); an `int` value on `BinaryStdOut` is a sequence of bits (its binary representation).

**Binary input and output.** Most systems nowadays, including Java, base their I/O on 8-bit bytestreams, so we might decide to read and write bytestreams to match I/O formats with the internal representations of primitive types, encoding an 8-bit char with 1 byte, a 16-bit short with 2 bytes, a 32-bit int with 4 bytes, and so forth. Since *bitstreams* are the primary abstraction for data compression, we go a bit further to allow clients to read and write individual *bits*, intermixed with data of primitive types. The goal is to minimize the necessity for type conversion in client programs and also to take care of operating system conventions for representing data. We use the following API for reading a bitstream from standard input:

---

```
public class BinaryStdIn
```

boolean readBoolean()	<i>read 1 bit of data and return as a boolean value</i>
char readChar()	<i>read 8 bits of data and return as a char value</i>
char readChar(int r)	<i>read r (between 1 and 16) bits of data and return as a char value</i>

*[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]*

boolean isEmpty()	<i>is the bitstream empty?</i>
void close()	<i>close the bitstream</i>

**API for static methods that read from a bitstream on standard input**

---

A key feature of the abstraction is that, in marked contrast to StdIn, *the data on standard input is not necessarily aligned on byte boundaries*. If the input stream is a single byte, a client could read it 1 bit at a time with eight calls to readBoolean(). The close() method is not essential, but, for clean termination, clients should call close() to indicate that no more bits are to be read. As with StdIn/StdOut, we use the following complementary API for writing bitstreams to standard output:

---

```
public class BinaryStdOut
```

void write(boolean b)	<i>write the specified bit</i>
void write(char c)	<i>write the specified 8-bit char</i>
void write(char c, int r)	<i>write the r (between 1 and 16) least significant bits of the specified char</i>

*[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]*

void close()	<i>close the bitstream</i>
--------------	----------------------------

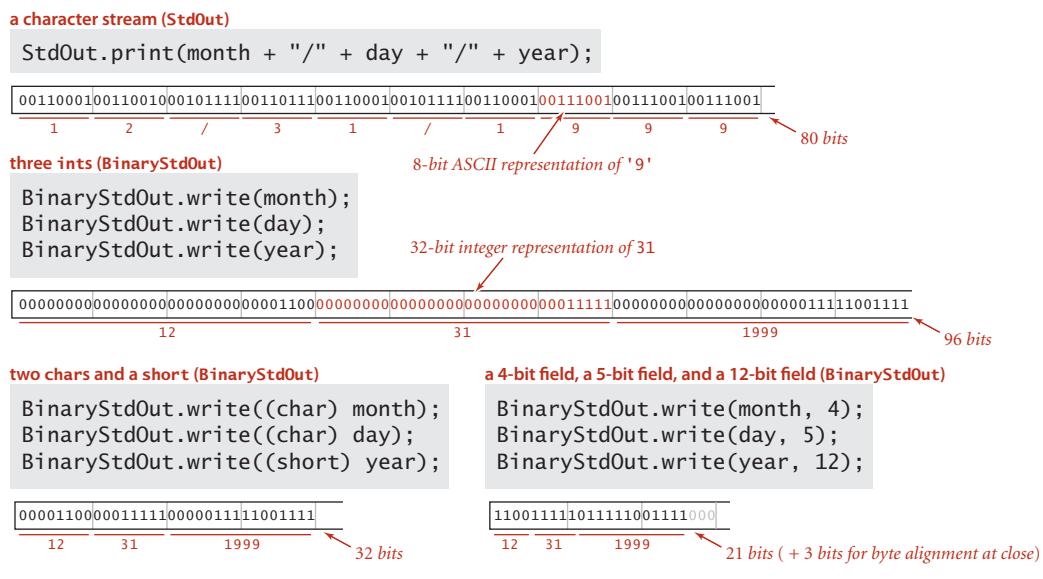
**API for static methods that write to a bitstream on standard output**

---

For output, the `close()` method is essential: clients must call `close()` to ensure that all of the bits specified in prior `write()` calls make it to the bitstream and that the final byte is padded with 0s to byte-align the output for compatibility with the file system. As with the `In` and `Out` APIs associated with `StdIn` and `StdOut`, we also have available `BinaryIn` and `BinaryOut` that allows us to reference binary-encoded files directly.

**Example.** As a simple example, suppose that you have a data type where a date is represented as three `int` values (month, day, year). Using `StdOut` to write those values in the format 12/31/1999 requires 10 characters, or 80 bits. If you write the values directly with `BinaryStdOut`, you would produce 96 bits (32 bits for each of the 3 `int` values); if you use a more economical representation that uses byte values for the month and day and a `short` value for the year, you would produce 32 bits. With `BinaryStdOut` you could also write a 4-bit field, a 5-bit field, and a 12-bit field, for a total of 21 bits (24 bits, actually, because files must be an integral number of 8-bit bytes, so `close()` adds three 0 bits at the end). *Important note:* Such economy, in itself, is a crude form of data compression.

**Binary dumps.** How can we examine the contents of a bitstream or a bytestream while debugging? This question faced early programmers when the only way to find a bug was to examine each of the bits in memory, and the term *dump* has been used since the early days of computing to describe a human-readable view of a bitstream. If you try to open



```

public class BinaryDump
{
    public static void main(String[] args)
    {
        int width = Integer.parseInt(args[0]);
        int cnt;
        for (cnt = 0; !BinaryStdIn.isEmpty(); cnt++)
        {
            if (width == 0) continue;
            if (cnt != 0 && cnt % width == 0)
                StdOut.println();
            if (BinaryStdIn.readBoolean())
                StdOut.print("1");
            else StdOut.print("0");
        }
        StdOut.println();
        StdOut.println(cnt + " bits");
    }
}

```

Printing a bitstream on standard (character) output

a file with an editor or view it in the manner in which you view text files (or just run a program that uses `BinaryStdOut`), you are likely to see gibberish, depending on the system you use. `BinaryStdIn` allows us to avoid such system dependencies by writing our own programs to convert bitstreams such that we can see them with our standard tools. For example, the program `BinaryDump` at left is a `BinaryStdIn` client that prints out the bits from standard input, encoded with the characters 0 and 1. This program is useful for debugging when working with small inputs. The similar client

`HexDump` groups the data into 8-bit bytes and prints each as two hexadecimal digits that each represent 4 bits. The client `PictureDump` displays the bits in a `Picture` with 0 bits represented as white pixels and 1 bits represented as black pixels. This pictorial representation is often useful in identifying patterns in a bitstream. You can download `BinaryDump`, `HexDump`, and `PictureDump` from the booksite. Typically, we use piping and redirection at the command-line level when working with binary files: we can pipe the output of an encoder to `BinaryDump`, `HexDump`, or `PictureDump`, or redirect it to a file.

**standard character stream**

```
% more abra.txt
ABRACADABRA!
```

**bitstream represented as 0 and 1 characters**

```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

**bitstream represented with hex digits**

```
% java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
96 bits
```

**bitstream represented as pixels in a Picture**

```
% java PictureDump 16 6 < abra.txt
```

96 bits

Four ways to look at a bitstream

**ASCII encoding.** When you HexDump a bit-stream that contains ASCII-encoded characters, the table at right is useful for reference. Given a two digit hex number, use the first hex digit as a row index and the second hex digit as a column index to find the character that it encodes. For example, 31 encodes the digit 1, 4A encodes the letter J, and so forth. This table is for 7-bit ASCII, so the first hex digit must be 7 or less. Hex numbers starting with 0 and 1 (and the numbers 20 and 7F) correspond to non-printing control characters. Many of the control characters are left over from the days when physical devices such as typewriters were controlled by ASCII input; the table highlights a few that you might see in dumps. For example, SP is the space character, NUL is the null character, LF is line feed, and CR is carriage return.

IN SUMMARY, working with data compression requires us to reorient our thinking about standard input and standard output to include binary encoding of data. `BinaryStdIn` and `BinaryStdOut` provide the methods that we need. They provide a way for you to make a clear distinction in your client programs between writing out information intended for file storage and data transmission (that will be read by programs) and printing information (that is likely to be read by humans).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	^	_	
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal-to-ASCII conversion table

**Limitations** To appreciate data-compression algorithms, you need to understand fundamental limitations. Researchers have developed a thorough and important theoretical basis for this purpose, which we will consider briefly at the end of this section, but a few ideas will help us get started.

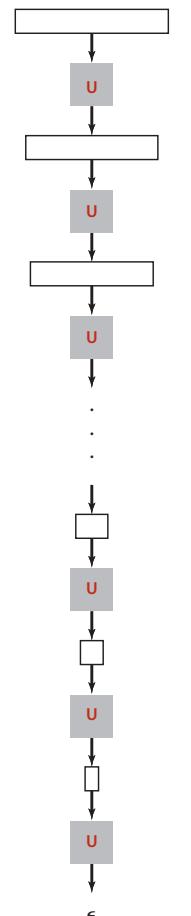
**Universal data compression.** Armed with the algorithmic tools that have proven so useful for so many problems, you might think that our goal should be *universal data compression*: an algorithm that can make any bitstream smaller. Quite to the contrary, we have to adopt more modest goals because universal data compression is impossible.

**Proposition S.** No algorithm can compress every bitstream.

**Proof:** We consider two proofs that each provide some insight. The first is by contradiction: Suppose that you have an algorithm that does compress every bitstream. Then you could use that algorithm to compress its output to get a still shorter bitstream, and continue until you have a bitstream of length 0! The conclusion that your algorithm compresses every bitstream to 0 bits is absurd, and so is the assumption that it can compress every bitstream.

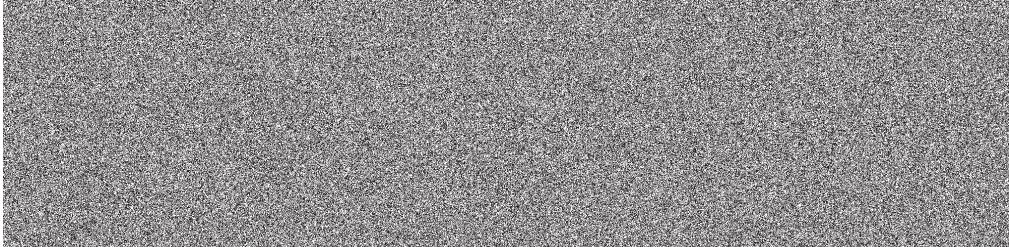
The second proof is a counting argument. Suppose that you have an algorithm that claims lossless compression for every 1,000-bit stream. That is, every such stream must map to a different shorter one. But there are only  $1 + 2 + 4 + \dots + 2^{998} + 2^{999} = 2^{1000} - 1$  bitstreams with fewer than 1,000 bits and  $2^{1000}$  bitstreams with 1,000 bits, so your algorithm cannot compress all of them. This argument becomes more persuasive if we consider stronger claims. Say your goal is to achieve better than a 50 percent compression ratio. You have to know that you will be successful for only about 1 out of  $2^{500}$  of the 1,000-bit streams!

Put another way, you have at most a 1 in  $2^{500}$  chance of being able to compress by half a random 1,000-bit stream with any data-compression algorithm. When you run across a new lossless compression algorithm, it is a sure bet that it will not achieve significant compression for a random bitstream. The insight that we cannot hope to compress random streams is a start to understanding data compression. We regularly process strings of millions or billions of bits but will never process even the tiniest fraction of all possible such strings, so we need



Universal  
data compression?

```
% java RandomBits | java PictureDump 2000 500
```



1000000 bits

A difficult file to compress: 1 million (pseudo-) random bits

not be discouraged by this theoretical result. Indeed, the bitstrings that we regularly process are typically highly structured, a fact that we can exploit for compression.

**Undecidability.** Consider the million-bit string pictured at the top of this page. This string appears to be random, so you are not likely to find a lossless compression algorithm that will compress it. But there is a way to represent that string with just a few thousand bits, because it was produced by the program below. (This program is an example of a pseudo-random number generator, like Java's `Math.random()` method.) A compression algorithm that compresses by writing the program in ASCII and expands by reading the program and then running it achieves a .3 percent compression ratio, which is difficult to beat (and we can drive the ratio arbitrarily low by writing more bits). To compress such a file is to discover the program that produced it. This example is not so far-fetched as it first appears: when you compress a video or an old book that was digitized with a scanner or any of countless other types of files from the web, you are discovering something about the program that produced the file. The realization that much of the data that we process is produced by a program leads to deep issues in the theory of computation and also gives insight into the challenges of data compression. For example, it is possible to prove that optimal data compression (find the shortest program to produce a given string) is an *undecidable* problem: not only can we not have an algorithm that compresses every bitstream, but also we cannot have a strategy for developing the best algorithm!

```
public class RandomBits
{
    public static void main(String[] args)
    {
        int x = 11111;
        for (int i = 0; i < 1000000; i++)
        {
            x = x * 314159 + 218281;
            BinaryStdOut.write(x > 0);
        }
        BinaryStdOut.close();
    }
}
```

A “compressed” million-bit stream

The practical impact of these limitations is that lossless compression methods must be oriented toward taking advantage of *known* structure in the bitstreams to be compressed. The four methods that we consider exploit, in turn, the following structural characteristics:

- Small alphabets
- Long sequences of identical bits/characters
- Frequently used characters
- Long reused bit/character sequences

If you know that a given bitstream exhibits one or more of these characteristics, you can compress it with one of the methods that you are about to learn; if not, trying them each is probably still worth the effort, since the underlying structure of your data may not be obvious, and these methods are widely applicable. As you will see, each method has parameters and variations that may need to be tuned for best compression of a particular bitstream. The first and last recourse is to learn something about the structure of your data yourself and exploit that knowledge to compress it, perhaps using one of the techniques we are about to consider.

**Warmup: genomics** As preparation for more complicated data-compression algorithms, we now consider an elementary (but very important) data-compression task. All of our implementations will use the same conventions that we will now introduce in the context of this example.

**Genomic data.** As a first example of data compression, consider this string:

```
ATAGATGCATAGCGCATAGCTAGATGTGCTAGCAT
```

Using standard ASCII encoding (1 byte, or 8 bits per character), this string is a bitstream of length  $8 \times 35 = 280$ . Strings of this sort are extremely important in modern biology, because biologists use the letters A, C, T, and G to represent the four nucleotides in the DNA of living organisms. A *genome* is a sequence of nucleotides. Scientists know that understanding the properties of genomes is a key to understanding the processes that manifest themselves in living organisms, including life, death, and disease. Genomes for many living things are known, and scientists are writing programs to study the structure of these sequences.

**2-bit code compression.** One simple property of genomes is that they contain only four different characters, so each can be encoded with just 2 bits per character, as in the `compress()` method shown at right. Even though we know the input stream to be character-encoded, we use `BinaryStdIn` to read the input, to emphasize adherence to the standard data-compression model (bitstream to bitstream). We include the number of encoded characters in the compressed file, to ensure proper decoding if the last bit does not fall at the end of a byte. Since it converts each 8-bit character to a 2-bit code and just adds 32 bits for the length, this program approaches a 25 percent compression ratio as the number of characters increases.

**2-bit code expansion.** The `expand()` method at the top of the next page expands a bitstream produced by this `compress()` method. As with compression, this method reads a bitstream and writes a bitstream, in accordance with the basic data-compression model. The bitstream that we produce as output is the original input.

```
public static void compress()
{
    Alphabet DNA = new Alphabet("ACTG");
    String s = BinaryStdIn.readString();
    int N = s.length();
    BinaryStdOut.write(N);
    for (int i = 0; i < N; i++)
    {
        // Write two-bit code for char.
        int d = DNA.toIndex(s.charAt(i));
        BinaryStdOut.write(d, DNA.lgR());
    }
    BinaryStdOut.close();
}
```

Compression method for genomic data

```

public static void expand()
{
    Alphabet DNA = new Alphabet("ACTG");
    int w = DNA.lgR();
    int N = BinaryStdIn.readInt();
    for (int i = 0; i < N; i++)
    {
        // Read 2 bits; write char.
        char c = BinaryStdIn.readChar(w);
        BinaryStdOut.write(DNA.toChar(c), 8);
    }
    BinaryStdOut.close();
}

```

Expansion method for genomic data

is widely reused. Other situations might require including the alphabet in the encoded message (see EXERCISE 5.5.25). The norm in data compression is to include such costs when comparing methods.

In the early days of genomics, learning a genomic sequence was a long and arduous task, so sequences were relatively short and scientists used standard ASCII encoding to store and exchange them. The experimental process has been vastly streamlined, to the point where known genomes are numerous and lengthy (the human genome is over  $10^{10}$  bits), and the 75 percent savings achieved by these simple methods is very significant. Is there room for further compression? That is a very interesting question to contemplate, because it is a *scientific* question: the ability to compress implies the existence of some structure in the data, and a prime focus of modern genomics is to discover structure in genomic data.

Standard data-compression methods like the ones we will consider are ineffective with (2-bit-encoded) genomic data, as with random data.

We package `compress()` and `expand()` as static methods in the same class, along with a simple driver, as shown at right. To test your understanding of the rules of the game and the basic tools that we use for data compression, make sure that you understand the various commands on the facing page that invoke `Genome.compress()` and `Genome.expand()` on our sample data (and their consequences).

THE SAME APPROACH works for other fixed-size alphabets, but we leave this generalization for an (easy) exercise (see EXERCISE 5.5.25).

These methods do not quite adhere to the standard data-compression model, because the compressed bitstream does not contain all the information needed to decode it. The fact that the alphabet is one of the letters A, C, T, or G is agreed upon by the two methods. Such a convention is reasonable in an application such as genomics, where the same code

```

public class Genome
{
    public static void compress()
        // See text.

    public static void expand()
        // See text.

    public static void main(String[] args)
    {
        if (args[0].equals("-")) compress();
        if (args[0].equals("+")) expand();
    }
}

```

Packaging convention for data-compression methods

**tiny test case (264 bits)**

```
% more genomeTiny.txt
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC

% java BinaryDump 64 < genomeTiny.txt
0100000101010100010000010100011101000001010101000100011101000011
0100000101010100010000010100011101000011010001110100001101000001
01010100010000010100011101000011010100010000010100011101000001
0101010001000011101010001000111010000110101000100000101000111
01000011
264 bits

% java Genome - < genomeTiny.txt
?? ← cannot see bitstream on standard output

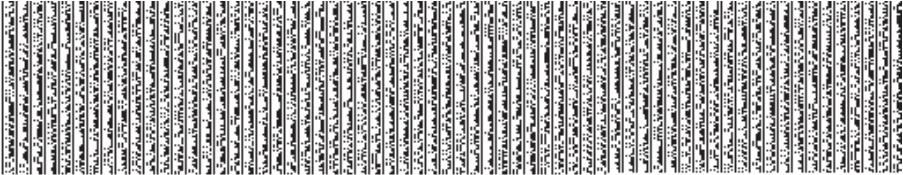
% java Genome - < genomeTiny.txt | java BinaryDump 64
0000000000000000000000000000000010000100100011001011010010001101110100
1000011011000110010111011011000110100000
104 bits

% java Genome - < genomeTiny.txt | java HexDump 8
00 00 00 21 23 2d 23 74
8d 8c bb 63 40
104 bits

% java Genome - < genomeTiny.txt > genomeTiny.2bit
% java Genome + < genomeTiny.2bit
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC ←

% java Genome - < genomeTiny.txt | java Genome +
ATAGATGCATAGCGCATAGCTAGATGTGCTAGC ← → compress-expand cycle
produces original input
```

**an actual virus (50000 bits)**

```
% java PictureDump 512 100 < genomeVirus.txt

50000 bits

% java Genome - < genomeVirus.txt | java PictureDump 512 25

12536 bits

Compressing and expanding genomic sequences with 2-bit encoding
```

**Run-length encoding** The simplest type of redundancy in a bitstream is long runs of repeated bits. Next, we consider a classic method known as *run-length encoding* for taking advantage of this redundancy to compress data. For example, consider the following 40-bit string:

```
00000000000001111110000001111111111
```

This string consists of 15 0s, then 7 1s, then 7 0s, then 11 1s, so we can encode the bitstring with the numbers 15, 7, 7, and 11. All bitstrings are composed of alternating runs of 0s and 1s; we just encode the length of the runs. In our example, if we use 4 bits to encode the numbers and start with a run of 0s, we get the 16-bit string

```
1111011101111011
```

( $15 = 1111$ , then  $7 = 0111$ , then  $7 = 0111$ , then  $11 = 1011$ ) for a compression ratio of  $16/40 = 40$  percent. In order to turn this description into an effective data compression method, we have to consider the following issues:

- How many bits do we use to store the counts?
- What do we do when encountering a run that is longer than the maximum count implied by this choice?
- What do we do about runs that are shorter than the number of bits needed to store their length?

We are primarily interested in long bitstreams with relatively few short runs, so we address these questions by making the following choices:

- Counts are between 0 and 255, all encoded with 8 bits.
- We make all run lengths less than 256 by including runs of length 0 if needed.
- We encode short runs, even though doing so might lengthen the output.

These choices are very easy to implement and also very effective for several kinds of bitstreams that are commonly encountered in practice. They are *not* effective when short runs are numerous—we save bits on a run only when the length of the run is more than the number of bits needed to represent itself in binary.

**Bitmaps.** As an example of the effectiveness of run-length encoding, we consider *bitmaps*, which are widely used to represent pictures and scanned documents. For brevity and simplicity, we consider binary-valued bitmaps organized as bitstreams formed by taking the pixels in row-major order. To view the contents of a bitmap, we use `PictureDump`. Writing a program to convert an image from one of the many common lossless image formats that have been defined for “screen shots” or scanned documents into a bitmap is a simple matter. Our example to demonstrate the effectiveness of run-length encoding comes from screen shots of this book: a letter *q* (at various resolutions). We focus on a binary dump of a 32-by-48-pixel screen shot, shown at right along with

run lengths for each row. Since each row starts and ends with a 0, there is an odd number of run lengths on each row; since the end of one row is followed by the beginning of the next, the corresponding run length in the bitstream is the sum of the last run length in each row and the first run length in the next (with extra additions corresponding to rows that are all 0).

**Implementation.** The informal description just given leads immediately to the `compress()` and `expand()` implementations on the next page. As usual, the `expand()` implementation is the simpler of the two: read a run length, print that many copies of the current bit, complement the current bit, and continue until the input is exhausted. The `compress()` method is not much more difficult, consisting of the following steps while there are bits in the input stream:

- Read a bit.
  - If it differs from the last bit read, write the current count and reset the count to 0.
  - If it is the same as the last bit read, and the count is a maximum, write the count, write a 0 count, and reset the count to 0.
  - Increment the count.

When the input stream empties, writing the count (length of the last run) completes the process.

***Increasing resolution in bitmaps.*** The primary reason that run-length encoding is widely used for bitmaps is that its effectiveness increases dramatically as resolution increases. It is easy to see why this is true. Suppose that we double the resolution for our example. Then the following facts are evident:

- The number of bits increases by a factor of 4.
  - The number of runs increases by about a factor of 2.
  - The run lengths increase by about a factor of 2.
  - The number of bits in the compressed version increases by about a factor of 2.
  - Therefore, the compression ratio is halved!

A typical bitmap, with run lengths for each row

```
public static void expand()
{
    boolean b = false;
    while (!BinaryStdIn.isEmpty())
    {
        char cnt = BinaryStdIn.readChar();
        for (int i = 0; i < cnt; i++)
            BinaryStdOut.write(b);
        b = !b;
    }
    BinaryStdOut.close();
}

public static void compress()
{
    char cnt = 0;
    boolean b, old = false;
    while (!BinaryStdIn.isEmpty())
    {
        b = BinaryStdIn.readBoolean();
        if (b != old)
        {
            BinaryStdOut.write(cnt, 8);
            cnt = 0;
            old = !old;
        }
        else
        {
            if (cnt == 255)
            {
                BinaryStdOut.write(cnt, 8);
                cnt = 0;
                BinaryStdOut.write(cnt, 8);
            }
            cnt++;
        }
    }
    BinaryStdOut.write(cnt);
    BinaryStdOut.close();
}
```

Expand and compress methods for run-length encoding

Without run-length encoding, space requirements increase by a factor of 4 when the resolution is doubled; with run-length encoding, space requirements for the compressed bitstream just double when the resolution is doubled. That is, space grows and the compression ratio drops linearly with resolution. For example, our (low-resolution) letter *q* yields just a 74 percent compression ratio; if we increase the resolution to 64 by 96, the ratio drops to 37 percent. This change is graphically evident in the PictureDump outputs shown in the figure on the facing page. The higher-resolution letter takes four times the space of the lower resolution letter (double in both dimensions), but the compressed version takes just twice the space (double in one dimension). If we further increase the resolution to 128-by-192 (closer to what is needed for print), the ratio drops to 18 percent (see EXERCISE 5.5.5).

RUN-LENGTH ENCODING IS VERY EFFECTIVE in many situations, but there are plenty of cases where the bitstream we wish to compress (for example, typical English-language text) may have no long runs at all. Next, we consider two methods that are effective for a broad variety of files. They are widely used, and you likely have used one or both of these methods when downloading from the web.

**tiny test case (40 bits)**

```
% java BinaryDump 40 < 4runs.bin
0000000000000000111111000000011111111111
40 bits

% java RunLength - < 4runs.bin | java HexDump
0f 07 07 0b
32 bits      compression ratio 32/40 = 80%

% java RunLength - < 4runs.bin | java RunLength + | java BinaryDump 40
0000000000000000111111000000011111111111 ← compress-expand produces original input
40 bits
```

**ASCII text (96 bits)**

```
% java RunLength - < abra.txt | java HexDump 24
01 01 05 01 01 04 01 02 01 01 02 01 02 01 05 01 01 01 01 04 02 01 01
05 01 01 01 03 01 03 01 05 01 01 01 04 01 02 01 01 01 02 01 02 01 05 01
02 01 04 01
416 bits ← compression ratio 416/96 = 433% — do not use run-length encoding for ASCII!
```

**a bitmap (1536 bits)**

```
% java RunLength - < q32x48.bin > q32x48.bin.rle
% java HexDump 16 < q32x48.bin.rle
4f 07 16 0f 0f 04 04 09 0d 04 09 06 0c 03 0c 05
0b 04 0c 05 0a 04 0d 05 09 04 0e 05 09 04 0e 05
08 04 0f 05 08 04 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 05 0f 05 07 05 0f 05
07 05 0f 05 07 05 0f 05 07 06 0e 05 07 06 0e 05
08 06 0d 05 08 06 0d 05 09 06 0c 05 09 07 0b 05
0a 07 0a 05 0b 08 07 06 0c 14 0e 0b 02 05 11 05
05 05 1b 05
1b 05 1b 05 1b 05 1a 07 16 0c 13 0e 41
1144 bits ← compression ratio 1144/1536 = 74%
```

**a higher-resolution bitmap (6144 bits)**

```
% java BinaryDump 0 < q64x96.bin
6144 bits
% java RunLength - < q64x96.bin | java BinaryDump 0
2296 bits ← compression ratio 2296/6144 = 37%
```

% java PictureDump 32 48 < q32x48.bin



1536 bits

% java PictureDump 32 36 < q32x48.bin.rle



1144 bits

% java PictureDump 64 96 < q64x96.bin



6144 bits

% java PictureDump 64 36 < q64x96.bin.rle



2296 bits

**Compressing and expanding bitstreams with run-length encoding**

**Huffman compression** We now examine a data-compression technique that can save a substantial amount of space in natural language files (and many other kinds of files). The idea is to abandon the way in which text files are usually stored: instead of using the usual 7 or 8 bits for each character, we use fewer bits for characters that appear often than for those that appear rarely.

To introduce the basic ideas, we start with a small example. Suppose we wish to encode the string ABRACADABRA!. Encoding it in 7-bit ASCII gives this bitstring:

```
100000110000101010010100000110000111000001-
10001001000001100001010100101000010100001.
```

To decode this bitstring, we simply read off 7 bits at a time and convert according to the ASCII coding table on page 815. In this standard code the D, which appears only once, requires the same number of bits as the A, which appears five times. Huffman compression is based on the idea that we can save bits by encoding frequently used characters with fewer bits than rarely used characters, thereby lowering the total number of bits used.

**Variable-length prefix-free codes.** A *code* associates each character with a bitstring: a symbol table with characters as keys and bitstrings as values. As a start, we might try to assign the shortest bitstrings to the most commonly used letters, encoding A with 0, B with 1, R with 00, C with 01, D with 10, and ! with 11, so ABRACADABRA! would be encoded as 0 1 00 0 01 0 10 0 1 00 0 11. This representation uses only 17 bits compared to the 84 for 7-bit ASCII, but it is not really a code because it depends on the blanks to delimit the characters. Without the blanks, the bitstring would be

```
01000010100100011
```

and could be decoded as CRRDDCRCB or as several other strings. Still, the count of 17 bits plus 11 delimiters is rather more compact than the standard code, primarily because no bits are used to encode letters not appearing in the message. The next step is to take advantage of the fact that *delimiters are not needed if no character code is the prefix of another*. A code with this property is known as a *prefix-free code*. The code just given is not prefix-free because 0, the code for A, is a prefix of 00, the code for R. For example, if we encode A with 0, B with 1111, C with 110, D with 100, R with 1110, and ! with 101, there is only one way to decode the 30-bit string

```
0111111001100100011111100101
```

ABRACADABRA!. All prefix-free codes are *uniquely decodable* (without needing any delimiters) in this way, so prefix-free codes are widely used in practice. Note that fixed-length codes such as 7-bit ASCII are prefix-free.

**Trie representation for prefix-free codes.** One convenient way to represent a prefix-free code is with a trie (see SECTION 5.2). In fact, any trie with  $M$  null links defines a prefix-free code for  $M$  characters: we replace the null links by links to *leaves* (nodes with two null links), each containing a character to be encoded, and define the code for each character with the bitstring defined by the path from the root to the character, in the standard manner for tries where we associate 0 with moving left and 1 with moving right. For example, the figure at right shows two prefix-free codes for the characters in ABRACADABRA!. On top is the variable-length code just considered; below is a code that produces the string

11000111101011100110001111101

which is 29 bits, 1 bit shorter. Is there a trie that leads to even more compression? How do we find the trie that leads to the best prefix-free code? It turns out that there is an elegant answer to these questions in the form of an algorithm that computes a trie which leads to a bitstream of minimal length for any given string. To make a fair comparison with other codes, we also need to count the bits in the code itself, since the string cannot be decoded without it, and, as you will see, the code depends on the string. The general method for finding the optimal prefix-free code was discovered by D. Huffman (while a student!) in 1952 and is called *Huffman encoding*.

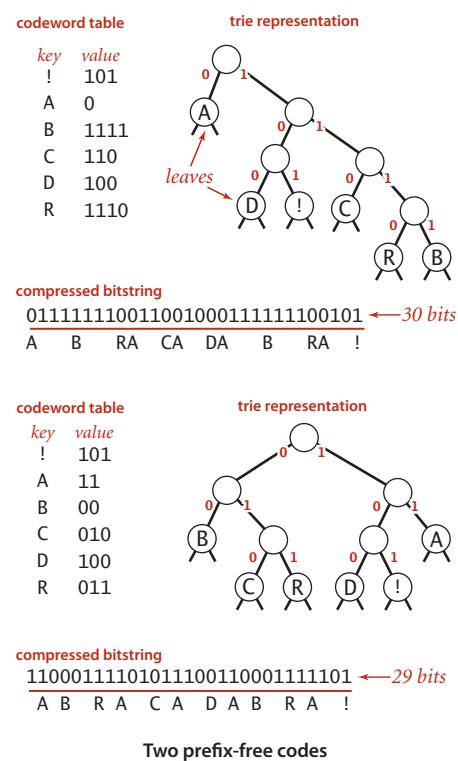
**Overview.** Using a prefix-free code for data compression involves five major steps. We view the bitstream to be encoded as a bytestream and use a prefix-free code for the characters as follows:

- Build an encoding trie.
- Write the trie (encoded as a bitstream) for use in expansion.
- Use the trie to encode the bytestream as a bitstream.

Then expansion requires that we

- Read the trie (encoded at the beginning of the bitstream)
- Use the trie to decode the bitstream

To help you best understand and appreciate the process, we consider these steps in order of difficulty.



Two prefix-free codes

```

private static class Node implements Comparable<Node>
{ // Huffman trie node
    private char ch; // unused for internal nodes
    private int freq; // unused for expand
    private final Node left, right;

    Node(char ch, int freq, Node left, Node right)
    {
        this.ch = ch;
        this.freq = freq;
        this.left = left;
        this.right = right;
    }

    public boolean isLeaf()
    { return left == null && right == null; }

    public int compareTo(Node that)
    { return this.freq - that.freq; }
}

```

Trie node representation

```

public static void expand()
{
    Node root = readTrie();
    int N = BinaryStdIn.readInt();
    for (int i = 0; i < N; i++)
    { // Expand ith codeword.
        Node x = root;
        while (!x.isLeaf())
            if (BinaryStdIn.readBoolean())
                x = x.right;
            else x = x.left;
        BinaryStdOut.write(x.ch, 8);
    }
    BinaryStdOut.close();
}

```

Prefix-free code expansion (decoding)

output A; go back to the root, move right three times, then output B; go back to the root, move right twice, then left, then output R; and so forth. The simplicity of expansion is one reason for the popularity of prefix-free codes in general and Huffman compression in particular.

**Trie nodes.** We begin with the Node class at left. It is similar to the nested classes that we have used before to construct binary trees and tries: each Node has left and right references to Nodes, which define the trie structure. Each Node also has an instance variable freq that is used in construction, and an instance variable ch, which is used in leaves to represent characters to be encoded.

**Expansion for prefix-free codes.** Expanding a bitstream that was encoded with a prefix-free code is simple, given the trie that defines the code. The

expand() method at left is an implementation of this process. After reading the trie from standard input using the readTrie() method to be described later, we use it to expand the rest of the bitstream as follows: Starting at the root, proceed down the trie as directed by the bitstream (read in input bit, move left if it is 0, and move right if it is 1). When you encounter a leaf, output the character at that node and restart at the root. If you study the operation of this method on the small prefix code example on the next page, you will understand and appreciate this process: For example, to decode the bitstring 011111001011... we start at the root, move left because the first bit is 0,

```

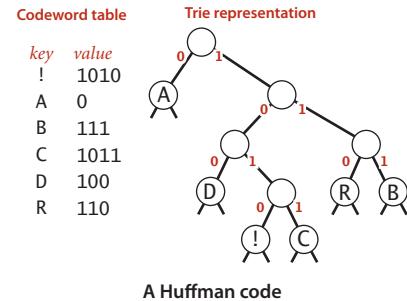
private static String[] buildCode(Node root)
{ // Make a lookup table from trie.
    String[] st = new String[R];
    buildCode(st, root, "");
    return st;
}

private static void buildCode(String[] st, Node x, String s)
{ // Make a lookup table from trie (recursive).
    if (x.isLeaf())
    { st[x.ch] = s; return; }
    buildCode(st, x.left, s + '0');
    buildCode(st, x.right, s + '1');
}

```

Building an encoding table from a (prefix-free) code trie

**Compression for prefix-free codes.** For compression, we use the trie that defines the code to build the code table, as shown in the `buildCode()` method at the top of this page. This method is compact and elegant, but a bit tricky, so it deserves careful study. For any trie, it produces a table giving the bit-string associated with each character in the trie (represented as a `String` of 0s and 1s). The coding table is a symbol table that associates a `String` with each character: we use a character-indexed array `st[]` instead of a general symbol table for efficiency, because the number of characters is not large. To create it, `buildCode()` recursively walks the tree, maintaining a binary string that corresponds to the path from the root to each node (0 for left links and 1 for right links), and setting the codeword corresponding to each character when the character is found in a leaf. Once the coding table is built, compression is a simple matter: just look up the code for each character in the input. To use the encoding at right to compress ABRACADABRA ! we write 0 (the codeword associated with A), then 111 (the codeword associated with B), then 110 (the codeword associated with R), and so forth. The code snippet at right accomplishes this task: we look up the `String` associated with each character in the input, convert it to 0/1 values in a `char` array, and write the corresponding bitstring to the output.



```

for (int i = 0; i < input.length; i++)
{
    String code = st[input[i]];
    for (int j = 0; j < code.length(); j++)
        if (code.charAt(j) == '1')
            BinaryStdOut.write(true);
        else BinaryStdOut.write(false);
}

```

Compression with an encoding table

**Trie construction.** For reference as we describe the process, the figure on the facing page illustrates the process of constructing a Huffman trie for the input

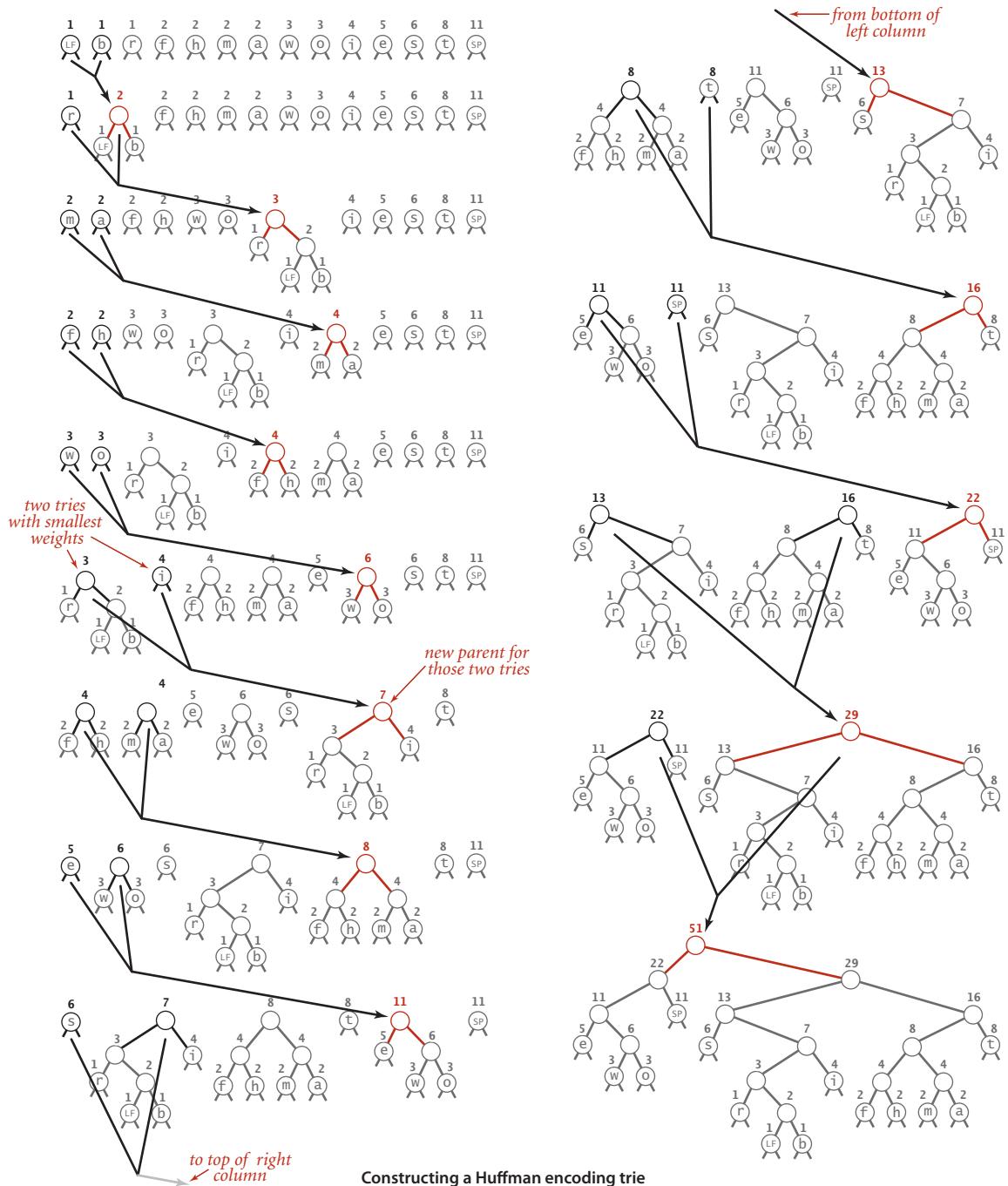
it was the best of times it was the worst of times

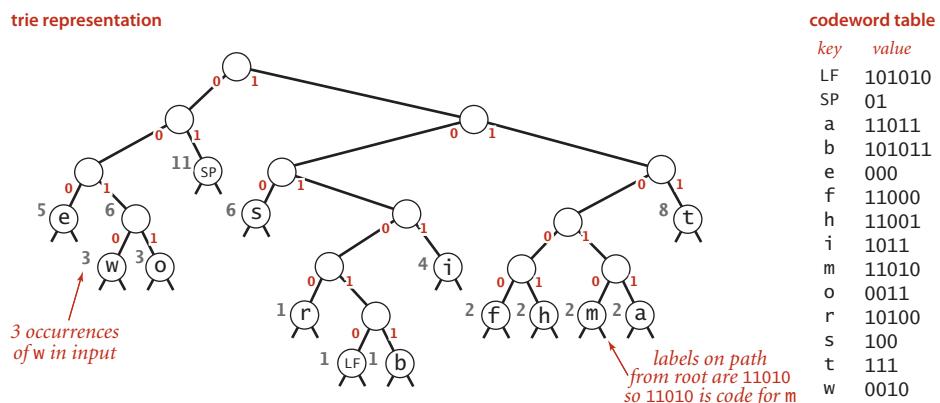
We keep the characters to be encoded in leaves and maintain the `freq` instance variable in each node that represents the frequency of occurrence of all characters in the subtree rooted at that node. The first step is to create a forest of 1-node trees (leaves), one for each character in the input stream, each assigned a `freq` value equal to its frequency of occurrence in the input. In the example, the input has 8 ts, 5 es, 11 spaces, and so forth. (*Important note:* To obtain these frequencies, we need to read the whole input stream—Huffman encoding is a *two-pass* algorithm because we will need to read the input stream a second time to compress it.) Next, we build the coding trie from the bottom up according to the frequencies. When building the trie, we view it as a binary trie with frequencies stored in the nodes; after it has been built, we view it as a trie for coding, as just described. The process works as follows: we find the two nodes with the smallest frequencies and then create a new node with those two nodes as children (and with frequency value set to the sum of the values of the children). This operation reduces the number of tries in the forest by one. Then we iterate the process: find the two nodes with smallest frequency in that forest and a create a new node created in the same way. Implementing the process is straightforward with a priority queue, as shown in the `buildTrie()` method at the bottom of this page. (For clarity, the tries in the figure are kept in sorted order.) Continuing, we build up larger and larger tries and at the same time reduce the number of tries in the forest by one at each step (remove two, add one). Ultimately, all the

```
private static Node buildTrie(int[] freq)
{
    // Initialize priority queue with singleton trees.
    MinPQ<Node> pq = new MinPQ<Node>();
    for (char c = 0; c < R; c++)
        if (freq[c] > 0)
            pq.insert(new Node(c, freq[c], null, null));

    while (pq.size() > 1)
    { // Merge two smallest trees.
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node('\0', x.freq + y.freq, x, y);
        pq.insert(parent);
    }
    return pq.delMin();
}
```

Building a Huffman encoding trie





Huffman code for the character stream “it was the best of times it was the worst of times LF”

nodes are combined together into a single trie. The leaves in this trie have the characters to be encoded and their frequencies in the input; each non-leaf node is the sum of the frequencies of its two children. Nodes with low frequencies end up far down in the trie, and nodes with high frequencies end up near the root of the trie. The frequency in the root equals the number of characters in the input. Since it is a binary trie with characters only in its leaves, it defines a prefix-free code for the characters. Using the codeword table created by `buildCode()` for this example (shown at right in the diagram at the top of this page), we get the output bitstring

```
101111101001011011000111110010000110101100-  
01001110100111100001111101111010000100011011-  
11101001011011100011111100100001001000111010-  
01001110100111100001111101111010000100101010.
```

which is 176 bits, a savings of 57 percent over the 408 bits needed to encode the 51 characters in standard 8-bit ASCII (not counting the cost of including the code, which we will soon consider). Moreover, since it is a *Huffman* code, no other prefix-free code can encode the input with fewer bits.

**Optimality.** We have observed that high-frequency characters are nearer the root of the trie than lower-frequency characters and are therefore encoded with fewer bits, so this is a good code, but why is it an *optimal* prefix-free code? To answer this question, we begin by defining the *weighted external path length* of a tree to be the sum of the weight (associated frequency count) times depth (see page 226) of all of the leaves.

**Proposition T.** For any prefix-free code, the length of the encoded bitstring is equal to the weighted external path length of the corresponding trie.

**Proof:** The depth of each leaf is the number of bits used to encode the character in the leaf. Thus, the weighted external path length is the length of encoded bitstring: it is equivalent to the sum over all letters of the number of occurrences times the number of bits per occurrence.

For our example, there is one leaf at distance 2 (sp, with frequency 11), three leaves at distance 3 (e, s, and t, with total frequency 19), three leaves at distance 4 (w, o, and i, with total frequency 10), five leaves at distance 5 (r, f, h, m, and a, with total frequency 9) and two leaves at distance 6 (lf and b, with total frequency 2), so the sum total is  $2 \cdot 11 + 3 \cdot 19 + 4 \cdot 10 + 5 \cdot 9 + 6 \cdot 2 = 176$ , the length of the output bitstring, as expected.

**Proposition U.** Given a set of  $r$  symbols and frequencies, the Huffman algorithm builds an optimal prefix-free code.

**Proof:** By induction on  $r$ . Assume that the Huffman code is optimal for any set of fewer than  $r$  symbols. Let  $T_H$  be the code computed by Huffman for the set of symbols and associated frequencies  $(s_1, f_1), \dots, (s_r, f_r)$  and denote the length of the code (weighted external path length of the trie) by  $W(T_H)$ . Suppose that  $(s_i, f_i)$  and  $(s_j, f_j)$  are the first two symbols chosen. The algorithm then computes the code  $T_H^*$  for the set of  $r-1$  symbols with  $(s_i, f_i)$  and  $(s_j, f_j)$  replaced by  $(s^*, f_i + f_j)$  where  $s^*$  is a new symbol in a leaf at some depth  $d$ . Note that

$$W(T_H) = W(T_H^*) - d(f_i + f_j) + (d + 1)(f_i + f_j) = W(T_H^*) + (f_i + f_j)$$

Now consider an optimal trie  $T$  for  $(s_1, f_1), \dots, (s_r, f_r)$ , of height  $h$ . Note that  $(s_i, f_i)$  and  $(s_j, f_j)$  must be at depth  $h$  (else we could make a trie with lower external path length by swapping them with nodes at depth  $h$ ). Also, assume  $(s_i, f_i)$  and  $(s_j, f_j)$  are siblings by swapping  $(s_j, f_j)$  with  $(s_i, f_i)$ 's sibling. Now consider the tree  $T^*$  obtained by replacing their parent with  $(s^*, f_i + f_j)$ . Note that (by the same argument as above)  $W(T) = W(T^*) + (f_i + f_j)$ .

By the inductive hypothesis  $T_H^*$  is optimal:  $W(T_H^*) \leq W(T^*)$ . Therefore,

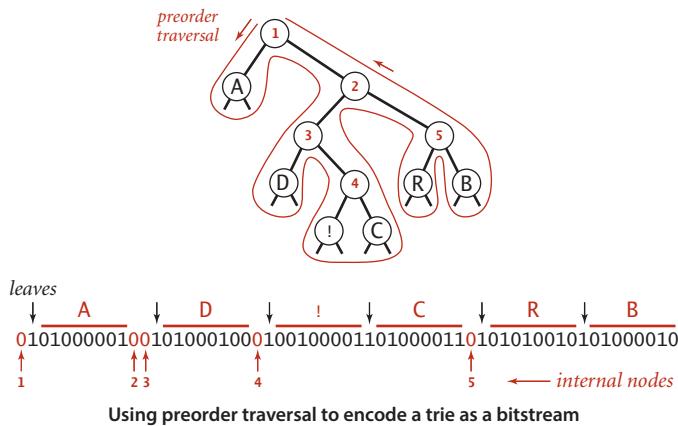
$$W(T_H) = W(T_H^*) + (f_i + f_j) \leq W(T^*) + (f_i + f_j) = W(T)$$

Since  $T$  is optimal, equality must hold, and  $T_H$  is optimal.

Whenever a node is to be picked, it can be the case that there are several nodes with the same weight. Huffman's method does not specify how such ties are to be broken. It also does not specify the left/right positions of the children. Different choices lead to

different Huffman codes, but all such codes will encode the message with the optimal number of bits among prefix-free codes.

**Writing and reading the trie.** As we have emphasized, the savings figure quoted above is not entirely accurate, because the compressed bitstream cannot be decoded without the trie, so we must account for the cost of including the trie in the compressed output, along with the bitstring. For long inputs, this cost is relatively small, but in order for us to have a



full data-compression scheme, we must write the trie onto a bitstream when compressing and read it back when expanding. How can we encode a trie as a bitstream, and then expand it? Remarkably, both tasks can be achieved with simple recursive procedures, based on a *preorder traversal* of the trie. The procedure `writeTrie()` below traverses a trie in preorder: when it visits an internal node, it writes a single 0 bit; when it visits a leaf, it writes a 1 bit, followed by the 8-bit ASCII code of the character in the leaf. The bitstring encoding of the Huffman trie for our ABRACADABRA! example is shown above. The first bit is 0, corresponding to the root; since the leaf containing A is encountered next, the next bit is 1, followed by 01000001, the 8-bit ASCII code for A; the next two bits are 0 because two internal nodes are encountered next, and so forth. The corresponding method `readTrie()` on the next page 835 reconstructs the trie from the bitstring: it reads a single bit to learn which type of node comes next: if a leaf (the bit is 1) it reads the next character and creates a leaf; if an internal node (the bit is 0) it creates an internal node and

```
private static void writeTrie(Node x)
{
    // Write bitstring-encoded trie.
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

Writing a trie as a bitstring

```
private static Node readTrie()
{
    if (BinaryStdIn.readBoolean())
        return new Node(BinaryStdIn.readChar(), 0, null, null);
    return new Node('\0', 0, readTrie(), readTrie());
}
```

#### Reconstructing a trie from the preorder bitstring representation

then (recursively) builds its left and right subtrees. *Be sure that you understand these methods:* their simplicity is somewhat deceiving.

**Huffman compression implementation.** Along with the methods `buildCode()`, `buildTrie()`, `readTrie()` and `writeTrie()` that we have just considered (and the `expand()` method that we considered first), ALGORITHM 5.10 is a complete implementation of Huffman compression. To expand the overview that we considered several pages earlier, we view the bitstream to be encoded as a stream of 8-bit char values and compress it as follows:

- Read the input.
- Tabulate the frequency of occurrence of each char value in the input.
- Build the Huffman encoding trie corresponding to those frequencies.
- Build the corresponding codeword table, to associate a bitstring with each char value in the input.
- Write the trie, encoded as a bitstring.
- Write the count of characters in the input, encoded as a bitstring.
- Use the codeword table to write the codeword for each input character.

To expand a bitstream encoded in this way, we

- Read the trie (encoded at the beginning of the bitstream)
- Read the count of characters to be decoded
- Use the trie to decode the bitstream

With four recursive trie-processing methods and a seven-step compression process, Huffman compression is one of the more involved algorithms that we have considered, but it is also one of the most widely used, because of its effectiveness.

**ALGORITHM 5.10** Huffman compression

```
public class Huffman
{
    private static int R = 256;    // ASCII alphabet
    // See page 828 for inner Node class.
    // See text for helper methods and expand().

    public static void compress()
    {
        // Read input.
        String s = BinaryStdIn.readString();
        char[] input = s.toCharArray();

        // Tabulate frequency counts.
        int[] freq = new int[R];
        for (int i = 0; i < input.length; i++)
            freq[input[i]]++;

        // Build Huffman code trie.
        Node root = buildTrie(freq);

        // Build code table (recursive).
        String[] st = new String[R];
        buildCode(st, root, "");

        // Print trie for decoder (recursive).
        writeTrie(root);

        // Print number of chars.
        BinaryStdOut.write(input.length);

        // Use Huffman code to encode input.
        for (int i = 0; i < input.length; i++)
        {
            String code = st[input[i]];
            for (int j = 0; j < code.length(); j++)
                if (code.charAt(j) == '1')
                    BinaryStdOut.write(true);
                else BinaryStdOut.write(false);
        }
        BinaryStdOut.close();
    }
}
```

This implementation of Huffman encoding builds an explicit coding trie, using various helper methods that are presented and explained in the last several pages of text.

## test case (96 bits)

```
% more abra.txt  
ABRACADABRA!  
  
% java Huffman - < abra.txt | java BinaryDump 60  
010100000100101000100010000110100001101010100101010000100  
00000000000000000000000000000000110001111100101101000111110010100  
120 bits ← compression ratio 120/96 = 125% due to 59 bits for trie and 32 bits for count
```

### example from text (408 bits)

```
% more tinytinyTale.txt
it was the best of times it was the worst of times

% java Huffman - < tinytinyTale.txt | java BinaryDump 64
0001011001010101110110111100100000001011100110010111001001
0000101010110001010110100100010110011010110100001011011011011000
0110111010000000000000000000000000000000000110011101111101001011011100
0111111001000011010110001001110100111100001111101111010000100011
0111110100101101110001111100100001001000111010010011110100111100
00111110111101000010010101000000
352 bits ← compression ratio 352/408 = 86% even with 137 bits for trie and 32 bits for count

% java Huffman - < tinytinyTale.txt | java Huffman +
it was the best of times it was the worst of times
```

## first chapter of *Tale of Two Cities*

```
% java PictureDump 512 90 < medTale.txt
```



45056 hits

```
% java Huffman - < medTale.txt | java PictureDump 512 47
```



23912 bits  $\leftarrow$  compression ratio  $23912/45056 = 53\%$

entire text of *Tale of Two Cities*

```
% java BinaryDump 0 < tale.txt  
5812552 bits
```

```
% java Huffman - < tale.txt > tale.txt.huf  
% java BinaryDump 0 < tale.txt.huf  
3043928 bits ← compression ratio 3043928/5812552 = 52%
```

ONE REASON FOR THE POPULARITY of Huffman compression is that it is effective for various types of files, not just natural language text. We have been careful to code the method so that it can work properly for any 8-bit value in each 8-bit character. In other words, we can apply it to any bytestream whatsoever. Several examples, for file types that we have considered earlier in this section, are shown in the figure at the bottom of this page. These examples show that Huffman compression is competitive with both fixed-length encoding and run-length encoding, even though those methods are designed to perform well for certain types of files. Understanding the reason Huffman encoding performs well in these domains is instructive. In the case of genomic data, Huffman compression essentially discovers a 2-bit code, as the four letters appear with approximately equal frequency so that the Huffman trie is balanced, with each character assigned a 2-bit code. In the case of run-length encoding, 0 0 0 0 0 0 0 and 1 1 1 1 1 1 1 are likely to be the most frequently occurring characters, so they are likely to be encoded with 2 or 3 bits, leading to substantial compression.

**virus (50000 bits)**

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```

  
12536 bits

```
% java Huffman - < genomeVirus.txt | java PictureDump 512 25
```

  
12576 bits ← *Huffman compression needs just 40 more bits than custom 2-bit code***bitmap (1536 bits)**

```
% java RunLength - < q32x48.bin | java BinaryDump 0  
1144 bits
```

```
% java Huffman - < q32x48.bin | java BinaryDump 0  
816 bits ← Huffman compression uses 29% fewer bits than customized method
```

**higher-resolution bitmap (6144 bits)**

```
% java RunLength - < q64x96.bin | java BinaryDump 0  
2296 bits
```

```
% java Huffman - < q64x96.bin | java BinaryDump 0  
2032 bits ← gap narrows to 11% for higher resolution
```

Compressing and expanding genomic data and bitmaps with Huffman encoding

A remarkable alternative to Huffman compression that was developed in the late 1970s and the early 1980s by A. Lempel, J. Ziv, and T. Welch has emerged as one of the most widely used compression methods because it is easy to implement and works well for a variety of file types.

The basic plan complements the basic plan for Huffman coding. Rather than maintain a table of *variable-length* codewords for *fixed-length* patterns in the input, we maintain a table of *fixed-length* codewords for *variable-length* patterns in the input. A surprising added feature of the method is that, by contrast with Huffman encoding, *we do not have to encode the table*.

**LZW compression** To fix ideas, we will consider a compression example where we read the input as a stream of 7-bit ASCII characters and write the output as a stream of 8-bit bytes. (In practice, we typically use larger values for these parameters—our implementations use 8-bit inputs and 12-bit outputs.) We refer to input bytes as *characters*, sequences of input bytes as *strings*, and output bytes as *codewords*, even though these terms have slightly different meanings in other contexts. The LZW compression algorithm is based on maintaining a symbol table that associates string keys with (fixed-length) codeword values. We initialize the symbol table with the 128 possible single-character string keys and associate them with 8-bit codewords obtained by prepending 0 to the 7-bit value defining each character. For economy and clarity, we use hexadecimal to refer to codeword values, so 41 is the codeword for ASCII A, 52 for R, and so forth. We reserve the codeword 80 to signify end of file (*EOF*). We will assign the rest of the codeword values (81 through FF) to various substrings of the input that we encounter, by starting at 81 and incrementing the value for each new key added. To compress, we perform the following steps as long as there are unscanned input characters:

- Find the longest string  $s$  in the symbol table that is a prefix of the unscanned input.
- Write the 8-bit value (codeword) associated with  $s$ .
- Scan one character past  $s$  in the input.
- Associate the next codeword value with  $s + c$  ( $c$  appended to  $s$ ) in the symbol table, where  $c$  is the next character in the input.

In the last of these steps, we look ahead to see the next character in the input to build the next dictionary entry, so we refer to that character  $c$  as the *lookahead* character. For the moment, we simply stop adding entries to the symbol table when we run out of codeword values (after assigning the value FF to some string)—we will later discuss alternate strategies.

**LZW compression example.** The figure below gives details of the operation of LZW compression for the example input AB RACADABRABRABRA. For the first seven characters, the longest prefix match is just one character, so we output the codeword associated with the character and associate the codewords from 81 through 87 to two-character strings. Then we find prefix matches with AB (so we output 81 and add ABR to the table), RA (so we output 83 and add RAB to the table), BR (so we output 82 and add BRA to the table), and ABR (so we output 88 and add ABRA to the table), leaving the last A (so we output its codeword, 41). *EOF*

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A	EOF	
matches	A	B	R	A	C	A	D	A B	R A	B R	A B R								
output	41	42	52	41	43	41	44	81	83	82	88							41	80
	AB 81	AB	AB	AB	AB	AB	AB	AB	81										
	B R 82	BR	BR	BR	BR	BR	BR	BR	82										
	R A 83	RA	RA	RA	RA	RA	RA	RA	83										
	A C 84	AC	AC	AC	AC	AC	AC	AC	84										
	C A 85	CA	CA	CA	CA	CA	CA	CA	85										
	A D 86	AD	AD	AD	AD	AD	AD	AD	86										
	D A 87	DA	DA	DA	DA	DA	DA	DA	87										
	A B R 88	ABR	ABR	ABR	ABR	ABR	ABR	ABR	88										
	R A B 89	RAB	RAB	RAB	RAB	RAB	RAB	RAB	89										
	B R A 8A	BRA	BRA	BRA	BRA	BRA	BRA	BRA	8A										
	A B R A 8B	ABRA	ABRA	ABRA	ABRA	ABRA	ABRA	ABRA	ABRA	8B									

codeword table  
key      value

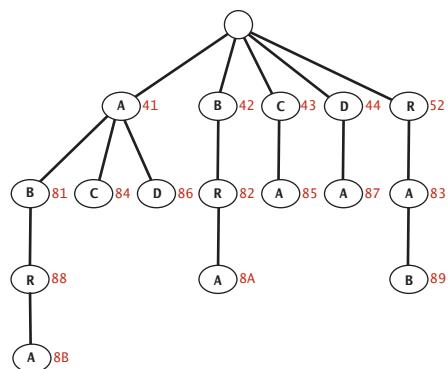
LZW compression for AB RACADABRABRABRA

The input is 17 ASCII characters of 7 bits each for a total of 119 bits; the output is 13 codewords (including *EOF*) of 8 bits each for a total of 104 bits—a compression ratio of 87 percent even for this tiny example.

**LZW trie representation.** LZW compression involves two symbol-table operations:

- Find a longest-prefix match of the input with a symbol-table key.
- Add an entry associating the next codeword with the key formed by appending the lookahead character to that key.

Our trie data structures of SECTION 5.2 are tailor-made for these operations. The trie representation for our example is shown at right. To find a longest prefix match, we traverse the trie from the root, matching node labels with input characters; to add a new codeword, we connect a new node labeled with the next codeword and the lookahead character to the node where the search terminated. In practice, we use a TST for



Trie representation of LZW code table

space efficiency, as described in SECTION 5.2. The contrast with the use of tries in Huffman encoding is worth noting: for Huffman encoding, tries are useful because no prefix of a codeword is also a codeword; for LZW tries are useful because *every* prefix of an input-substring key is also a key.

**LZW expansion.** The input for LZW expansion in our example is a sequence of 8-bit codewords; the output is a string of 7-bit ASCII characters. To implement expansion, we maintain a symbol table that associates strings of characters with codeword values (the inverse of the table used for compression). We fill the table entries from 00 to 7F with one-character strings, one for each ASCII character, set the first unassigned codeword value to 81 (reserving 80 for end of file), set the current string *val* to the one-character string consisting of the first character, and perform the following steps until reading codeword 80 (end of file):

- Write the current string *val*.
- Read a codeword *x* from the input.
- Set *s* to the value associated with *x* in the symbol table.
- Associate the next unassigned codeword value to *val* + *c* in the symbol table, where *c* is the first character of *s*.
- Set the current string *val* to *s*.

This process is more complicated than compression because of the lookahead character: we need to read the next codeword to get the first character in the string associated with it, which puts the process one step out of synch. For the first seven codewords, we just look up and write the appropriate character, then look ahead one character and add a two-character entry to the symbol table, as before. Then we read 81 (so we write A B and add A B R to the table), 83 (so we write R A and add R A B to the table), 82 (so we write B R and add B R A to the table), and 88 (so we write A B R and add A B R A to the table), leaving 41. Finally we read the end-of-file character 80 (so we write A ). At

input	41	42	52	41	43	41	44	81		83	82	88	41	80
output	A	B	R	A	C	A	D	A B	R A	B R	A B R	A	A	
inverse codeword table														
81 A B	A B	A B	A B	A B	A B	A B	A B	A B	A B	B R	B R	A C	81 A B	
82 B R	B R	B R	B R	B R	B R	B R	B R	B R	B R	R A	R A	R A B	82 B R	
83 R A	R A	R A	R A	R A	R A	R A	R A	R A	R A	A C	A C	A C A	83 R A	
84 A C	A C	A C	A C	A C	A C	A C	A C	A C	A C	C A	C A	C A C	84 A C	
85 C A	C A	C A	C A	C A	C A	C A	C A	C A	C A	A D	A D	A D A	85 C A	
86 A D	A D	A D	A D	A D	A D	A D	A D	A D	A D	D A	D A	D A D	86 A D	
87 D A	D A	D A	D A	D A	D A	D A	D A	D A	D A	A B R	A B R	A B R A	87 D A	
88 A B R	A B R	A B R	A B R	A B R	A B R	A B R	A B R	A B R	A B R	R A B	R A B	R A B A	88 A B R	
89 R A B	R A B	R A B	R A B	R A B	R A B	R A B	R A B	R A B	R A B	B R A	B R A	B R A A	89 R A B	
8A B R A	B R A	B R A	B R A	B R A	B R A	B R A	B R A	B R A	B R A	A B R A	A B R A	A B R A A	8A B R A	
8B A B R A	A B R A	A B R A	A B R A	A B R A	A B R A	A B R A	A B R A	A B R A	A B R A	A B R A A	A B R A A	A B R A A A	8B A B R A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

### ALGORITHM 5.11 LZW compression

```

public class LZW
{
    private static final int R = 256;      // number of input chars
    private static final int L = 4096;     // number of codewords = 2^12
    private static final int W = 12;       // codeword width

    public static void compress()
    {
        String input = BinaryStdIn.readString();
        TST<Integer> st = new TST<Integer>();
        for (int i = 0; i < R; i++)
            st.put("", (char) i, i);
        int code = R+1; // R is codeword for EOF.

        while (input.length() > 0)
        {
            String s = st.longestPrefixOf(input); // Find max prefix match.
            BinaryStdOut.write(st.get(s), W); // Print s's encoding.
            int t = s.length();
            if (t < input.length() && code < L) // Add s to symbol table.
                st.put(input.substring(0, t + 1), code++);
            input = input.substring(t); // Scan past s in input.
        }
        BinaryStdOut.write(R, W); // Write EOF.
        BinaryStdOut.close();
    }

    public static void expand()
    // See page 844.
}

```

This implementation of Lempel-Ziv-Welch data compression uses 8-bit input bytes and 12-bit codewords and is appropriate for arbitrary large files. Its codewords for the small example are similar to those discussed in the text: the single-character codewords have a leading 0; the others start at 100. Efficiency of this code depends on a constant-time `substring()` method (see page 202).

```

% more abraLZW.txt
ABRACADABRABRABRA

% java LZW - < abraLZW.txt | java HexDump 20
04 10 42 05 20 41 04 30 41 04 41 01 10 31 02 10 80 41 10 00
160 bits

```

the end of the process, we have written the original input, as expected, and also built the same code table as for compression, but with the key-value roles inverted. Note that we can use a simple array-of-strings representation for the table, indexed by codeword.

**Tricky situation.** There is a subtle bug in the process just described, one that is often discovered by students (and experienced programmers!) only after developing an implementation based on the description above.

The problem, illustrated in the example at right, is that it is possible for the lookahead process to get one character ahead of itself. In the example, the input string

A B A B A B A

is compressed to five output codewords

41 42 81 83 80

as shown in the top part of the figure. To expand, we read the codeword 41, output A, read the codeword 42 to get the lookahead character, add AB as table entry 81, output the B associated with 42, read the codeword 81 to get the lookahead character, add BA as table entry 82, and output the AB associated with 81. So far, so good. But when we read the codeword 83 to get the lookahead character, we are stuck, because the reason that we are reading that codeword is to complete table entry 83! Fortunately, it is easy to test for that condition (it happens precisely when the codeword is the same as the table entry to be completed) and to correct it (the lookahead character must be the first character in that table entry, since that will be the next character to be output). In this example, this logic tells us that the lookahead character must be A (the first character in ABA). Thus, both the next output string and table entry 83 should be ABA.

**Implementation.** With these descriptions, implementing LZW encoding is straightforward, given in ALGORITHM 5.11 on the facing page (the implementation of `expand()` is on the next page). These implementations take 8-bit bytes as input (so we can compress any file, not just strings) and produce 12-bit codewords as output (so that we can get better compression by having a much larger dictionary). These values are specified in the (final) instance variables `R`, `L`, and `W` in the code. We use a TST (see SECTION 5.2) for the code table in `compress()` (taking advantage of the ability of trie data structures to support efficient implementations of `longestPrefixOf()`) and an array of strings

compression	
input	A      B      A      B      A      B      A
matches	A      B      A B      A B A
output	41      42      81      83      80
	codeword table key    value
	AB    81 BA    82 ABA    83

expansion	
input	41      42      81      83      80
output	A      B      A B      ?      80
	must be ABA (see below)
81 AB	AB      AB      BA
82 BA	BA      BA
83 AB?	AB?      ?
	need lookahead character to complete entry
	next character in output—the lookahead character!

LZW expansion: tricky situation

**ALGORITHM 5.11 (continued) LZW expansion**

```
public static void expand()
{
    String[] st = new String[L];
    int i; // next available codeword value
    for (i = 0; i < R; i++)           // Initialize table for chars.
        st[i] = "" + (char) i;
    st[i++] = " "; // (unused) lookahead for EOF
    int codeword = BinaryStdIn.readInt(W);
    String val = st[codeword];
    while (true)
    {
        BinaryStdOut.write(val);      // Write current substring.
        codeword = BinaryStdIn.readInt(W);
        if (codeword == R) break;
        String s = st[codeword];      // Get next codeword.
        if (i == codeword)           // If lookahead is invalid,
            s = val + val.charAt(0); // make codeword from last one.
        if (i < L)
            st[i++] = val + s.charAt(0); // Add new entry to code table.
        val = s;                     // Update current codeword.
    }
    BinaryStdOut.close();
}
```

This implementation of expansion for the Lempel-Ziv-Welch algorithm is a bit more complicated than compression because of the need to extract the lookahead character from the next codeword and because of a tricky situation where lookahead is invalid (see text).

```
% java LZW - < abraLZW.txt | java LZW +
ABRACADABRABRABRA
% more ababLZW.txt
ABABABA
% java LZW - < ababLZW.txt | java LZW +
ABABABA
```

for the inverse code table in `expand()`. With these choices, the code for `compress()` and `expand()` is little more than a line-by-line translation of the descriptions in the text. These methods are very effective as they stand. For certain files, they can be further improved by emptying the codeword table and starting over each time all the codeword values are used. These improvements, along with experiments to evaluate their effectiveness, are addressed in the exercises at the end of this section.

AS USUAL, it is worth your while to study carefully the examples given with the programs and at the bottom of this page of LZW compression in action. Over the several decades since its invention, it has proven to be a versatile and effective data-compression method.

**virus (50000 bits)**

```
% java Genome - < genomeVirus.txt | java PictureDump 512 25
```



12536 bits

```
% java LZW - < genomeVirus.txt | java PictureDump 512 36
```



18232 bits ← *not as good as 2-bit code because repetitive data is rare*

**bitmap (6144 bits)**

```
% java RunLength - < q64x96.bin | java BinaryDump 0  
2296 bits
```

```
% java LZW - < q64x96.bin | java BinaryDump 0  
2824 bits ← not as good as run-length code because file size is too small
```

**entire text of *Tale of Two Cities* (5812552 bits)**

```
% java BinaryDump 0 < tale.txt  
5812552 bits
```

```
% java Huffman - < tale.txt | java BinaryDump 0  
3043928 bits
```

```
% java LZW - < tale.txt | java BinaryDump 0  
2667952 bits ← compression ratio 2667952/5812552 = 46% (best yet)
```

Compressing and expanding various files with LZW 12-bit encoding

**Q&A**

**Q.** Why `BinaryStdIn` and `BinaryStdOut`?

**A.** It's a tradeoff between efficiency and convenience. `StdIn` can handle 8 bits at a time; `BinaryStdIn` has to handle each bit. Most applications are bytestream-oriented; data compression is a special case.

**Q.** Why `close()`?

**A.** This requirement stems from the fact that standard output is actually a bytestream, so `BinaryStdOut` needs to know when to write the last byte.

**Q.** Can we mix `StdIn` and `BinaryStdIn`?

**A.** That is not a good idea. Because of system and implementation dependencies, there is no guarantee of what might happen. Our implementations will raise an exception. On the other hand, there is no problem with mixing `StdOut` and `BinaryStdOut` (we do it in our code).

**Q.** Why is the `Node` class `static` in `Huffman`?

**A.** Our data-compression algorithms are organized as collections of static methods, not data-type implementations.

**Q.** Can I at least guarantee that my compression algorithm will not increase the length of a bitstream?

**A.** You can just copy it from input to output, but you still need to signify not to use a standard compression scheme. Commercial implementations sometimes make this guarantee, but it is quite weak and far from universal compression. Indeed, typical compression algorithms do not even make it past the second step of our first proof of PROPOSITION S: few algorithms will further compress a bitstring produced by that same algorithm.

## EXERCISES

**5.5.1** Consider the four variable-length codes shown in the table at right. Which of the codes are prefix-free? Uniquely decodable? For those that are uniquely decodable, give the decoding of 1000000000000.

**5.5.2** Give an example of a uniquely decodable code that is not prefix-free.

*Answer:* Any *suffix-free* code is uniquely decodable.

**5.5.3** Give an example of a uniquely decodable code that is not prefix free or suffix free.

*Answer:* {0011, 011, 11, 1110} or {01, 10, 011, 110}

**5.5.4** Are {1, 100000, 00} and {01, 1001, 1011, 111, 1110} uniquely decodable? If not, find a string with two encodings.

**5.5.5** Use RunLength on the file q128x192.bin from the booksite. How many bits are there in the compressed file?

**5.5.6** How many bits are needed to encode  $N$  copies of the symbol a (as a function of  $N$ )?  $N$  copies of the sequence abc?

**5.5.7** Give the result of encoding the strings a, aa, aaa, aaaa, ... (strings consisting of  $N$  a's) with run-length, Huffman, and LZW encoding. What is the compression ratio as a function of  $N$ ?

**5.5.8** Give the result of encoding the strings ab, abab, ababab, abababab, ... (strings consisting of  $N$  repetitions of ab) with run-length, Huffman, and LZW encoding. What is the compression ratio as a function of  $N$ ?

**5.5.9** Estimate the compression ratio achieved by run-length, Huffman, and LZW encoding for a *random* ASCII string of length  $N$  (all characters equally likely at each position, independently).

**5.5.10** In the style of the figure in the text, show the Huffman coding tree construction process when you use Huffman for the string "it was the age of foolishness". How many bits does the compressed bitstream require?

symbol	code 1	code 2	code 3	code 4
A	0	0	1	1
B	100	1	01	01
C	10	00	001	001
D	11	11	0001	000

**EXERCISES (continued)**

**5.5.11** What is the Huffman code for a string whose characters are all from a two-character alphabet? Give an example showing the maximum number of bits that could be used in a Huffman code for an  $N$ -character string whose characters are all from a two-character alphabet.

**5.5.12** Suppose that all of the symbol probabilities are negative powers of 2. Describe the Huffman code.

**5.5.13** Suppose that all of the symbol frequencies are equal. Describe the Huffman code.

**5.5.14** Suppose that the frequencies of the occurrence of all the characters to be encoded are different. Is the Huffman encoding tree unique?

**5.5.15** Huffman coding could be extended in a straightforward way to encode in 2-bit characters (using 4-way trees). What would be the main advantage and the main disadvantage of doing so?

**5.5.16** What is the LZW encoding of the following inputs?

- a. T O B E O R N O T T O B E
- b. Y A B B A D A B B A D A B B A D O O
- c. A

**5.5.17** Characterize the tricky situation in LZW coding.

*Solution:* Whenever it encounters  $cScS$ , where  $c$  is a symbol and  $S$  is a string,  $cS$  is in the dictionary already but  $cSc$  is not.

**5.5.18** Let  $F_k$  be the  $k$ th Fibonacci number. Consider  $N$  symbols, where the  $k$ th symbol has frequency  $F_k$ . Note that  $F_1 + F_2 + \dots + F_N = F_{N+2} - 1$ . Describe the Huffman code.  
*Hint:* The longest codeword has length  $N - 1$ .

**5.5.19** Show that there are at least  $2^{N-1}$  different Huffman codes corresponding to a given set of  $N$  symbols.

**5.5.20** Give a Huffman code where the frequency of 0s in the output is much, much higher than the frequency of 1s.

**5.5.21** Prove that the two longest codewords in a Huffman code have the same length.

**5.5.22** Prove the following fact about Huffman codes: If the frequency of symbol  $i$  is strictly larger than the frequency of symbol  $j$ , then the length of the codeword for symbol  $i$  is less than or equal to the length of the codeword for symbol  $j$ .

**5.5.23** What would be the result of breaking up a Huffman-encoded string into five-bit characters and Huffman-encoding that string?

**5.5.24** In the style of the figures in the text, show the encoding trie and the compression and expansion processes when LZW is used for the string

```
it was the best of times it was the worst of times
```

## CREATIVE PROBLEMS

**5.5.25 Fixed-length code.** Implement a class RLE that uses fixed-length encoding, to compress ASCII bytestreams using relatively few different characters, including the code as part of the encoded bitstream. Add code to `compress()` to make a string `alpha` with all the distinct characters in the message and use it to make an `Alphabet` for use in `compress()`, prepend `alpha` (8-bit encoding plus its length) to the compressed bit-stream, then add code to `expand()` to read the alphabet before expansion.

**5.5.26 Rebuilding the LZW dictionary.** Modify LZW to empty the dictionary and start over when it is full. This approach is recommended in some applications because it better adapts to changes in the general character of the input.

**5.5.27 Long repeats.** Estimate the compression ratio achieved by run-length, Huffman, and LZW encoding for a string of length  $2N$  formed by concatenating *two copies* of a random ASCII string of length  $N$  (see EXERCISE 5.5.9).

*This page intentionally left blank*

SIX



# Context

**C**OMPUTING DEVICES ARE UBIQUITOUS in the modern world. In the last several decades, we have evolved from a world where computing devices were virtually unknown to a world where billions of people use them regularly. Moreover, today's cellphones are orders of magnitude more powerful than the supercomputers that were available only to the privileged few as little as 30 years ago. But many of the underlying algorithms that enable these devices to work effectively are the same ones that we have studied in this book. Why? *Survival of the fittest*. Scalable (linear and linearithmic) algorithms have played a central role in the process and validate the idea that efficient algorithms are important. Researchers of the 1960s and 1970s built the basic infrastructure that we now enjoy with such algorithms. They knew that scalable algorithms are the key to the future; the developments of the past several decades have validated that vision. Now that the infrastructure is built, people are beginning to *use* it, for all sorts of purposes. As B. Chazelle has famously observed, the 20th century was the century of the equation, but the 21st century is the century of the *algorithm*.

Our treatment of fundamental algorithms in this book is only a starting point. The day is soon coming (if it is not already here) when one could build a college major around the study of algorithms. In commercial applications, scientific computing, engineering, operations research (OR), and countless other areas of inquiry too diverse to even mention, efficient algorithms make the difference between being able to solve problems in the modern world and not being able to address them at all. Our emphasis throughout this book has been to study *important* and *useful* algorithms. In this chapter, we reinforce this orientation by considering examples that illustrate the role of the algorithms that we have studied (and our approach to the study of algorithms) in

several advanced contexts. To indicate the scope of the impact of the algorithms, we begin with a very brief description of several important areas of application. To indicate the depth, we later consider specific representative examples in detail and introduce the theory of algorithms. In both cases, this brief treatment at the end of a long book can only be indicative, not inclusive. For every area of application that we mention, there are dozens of others, equally broad in scope; for every point that we describe within an application, there are scores of others, equally important; and for every detailed example we consider, there are hundreds if not thousands of others, equally impactful.

**Commercial applications.** The emergence of the internet has underscored the central role of algorithms in *commercial applications*. All of the applications that you use regularly benefit from the classic algorithms that we have studied:

- Infrastructure (operating systems, databases, communications)
- Applications (email, document processing, digital photography)
- Publishing (books, magazines, web content)
- Networks (wireless networks, social networks, the internet)
- Transaction processing (financial, retail, web search)

As a prominent example, we consider in this chapter *B-trees*, a venerable data structure that was developed for mainstream computers of the 1960s but still serve as the basis for modern database systems. We will also discuss *suffix arrays*, for text indexing.

**Scientific computing.** Since von Neumann developed mergesort in 1950, algorithms have played a central role in *scientific computing*. Today's scientists are awash in experimental data and are using both mathematical and computational models to understand the natural world for:

- Mathematical calculations (polynomials, matrices, differential equations)
- Data processing (experimental results and observations, especially genomics)
- Computational models and simulation

All of these can require complex and extensive computing with huge amounts of data. As a detailed example of an application in scientific computing, we consider in this chapter a classic example of *event-driven simulation*. The idea is to maintain a model of a complicated real-world system, controlling changes in the model over time. There are a vast number of applications of this basic approach. We also consider a fundamental data-processing problem in computational genomics.

**Engineering.** Almost by definition, modern *engineering* is based on technology. Modern technology is computer-based, so algorithms play a central role for

- Mathematical calculations and data processing
- Computer-aided design and manufacturing

- Algorithm-based engineering (networks, control systems)
- Imaging and other medical systems

Engineers and scientists use many of the same tools and approaches. For example, scientists develop computational models and simulations for the purpose of understanding the natural world; engineers develop computational models and simulations for the purpose of designing, building, and controlling the artifacts they create.

**Operations research.** Researchers and practitioners in OR develop and apply mathematical models for problem solving, including

- Scheduling
- Decision making
- Assignment of resources

The shortest-paths problem of SECTION 4.4 is a classic OR problem. We revisit this problem and consider the *maxflow* problem, illustrate the importance of *reduction*, and discuss implications for general problem-solving models, in particular the *linear programming* model that is central in OR.

ALGORITHMS PLAY AN IMPORTANT ROLE in numerous subfields of computer science with applications in all of these areas, including, but certainly not limited to

- Computational geometry
- Cryptography
- Databases
- Programming languages and systems
- Artificial intelligence

In each field, articulating problems and finding efficient algorithms and data structures for solving them play an essential role. Some of the algorithms we have studied apply directly; more important, the general approach of designing, implementing, and analyzing algorithms that lies at the core of this book has proven successful in all of these fields. This effect is spreading beyond computer science to many other areas of inquiry, from games to music to linguistics to finance to neuroscience.

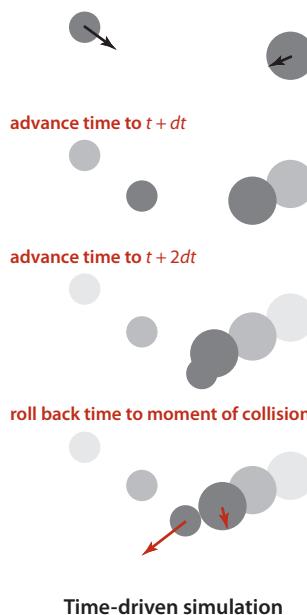
So many important and useful algorithms have been developed that learning and understanding relationships among them are essential. We finish this section (and this book!) with an introduction to the *theory of algorithms*, with particular focus on *intractability* and the **P=NP?** question that still stands as the key to understanding the practical problems that we aspire to solve.

**Event-driven simulation** Our first example is a fundamental scientific application: simulate the motion of a system of moving particles that behave according to the laws of elastic collision. Scientists use such systems to understand and predict properties of physical systems. This paradigm embraces the motion of molecules in a gas, the dynamics of chemical reactions, atomic diffusion, sphere packing, the stability of the rings around planets, the phase transitions of certain elements, one-dimensional self-gravitating systems, front propagation, and many other situations. Applications range from molecular dynamics, where the objects are tiny subatomic particles, to astrophysics, where the objects are huge celestial bodies.

Addressing this problem requires a bit of high-school physics, a bit of software engineering, and a bit of algorithmics. We leave most of the physics for the exercises at the end of this section so that we can concentrate on the topic at hand: using a fundamental algorithmic tool (heap-based priority queues) to address an application, enabling calculations that would not otherwise be possible.

**Hard-disc model.** We begin with an idealized model of the motion of atoms or molecules in a container that has the following salient features:

- Moving *particles* interact via elastic collisions with each other and with *walls*.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces are exerted.



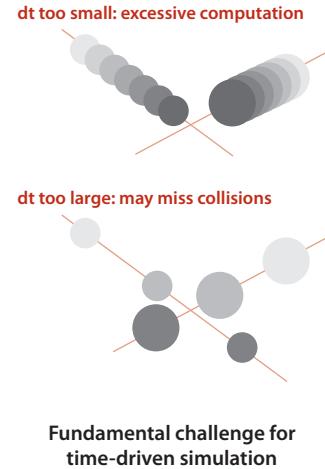
This simple model plays a central role in *statistical mechanics*, a field that relates macroscopic observables (such as temperature and pressure) to microscopic dynamics (such as the motion of individual atoms and molecules). Maxwell and Boltzmann used the model to derive the distribution of speeds of interacting molecules as a function of temperature; Einstein used the model to explain the Brownian motion of pollen grains immersed in water. The assumption that no other forces are exerted implies that particles travel in straight lines at constant speed between collisions. We could also extend the model to add other forces. For example, if we add friction and spin, we can more accurately model the motion of familiar physical objects such as billiard balls on a pool table.

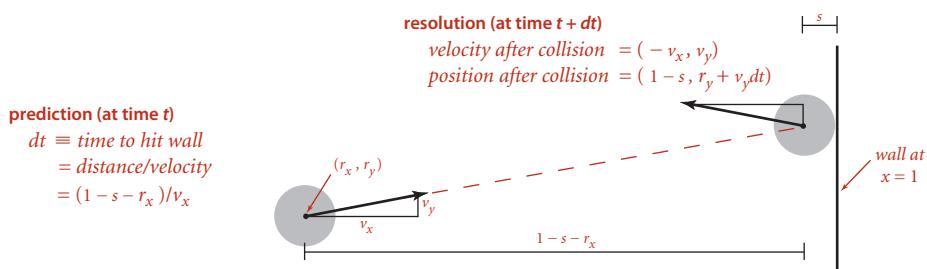
**Time-driven simulation.** Our primary goal is simply to maintain the model: that is, we want to be able to keep track of the positions and velocities of all the particles as time passes. The basic calculation that we have to do is the following: given the positions and velocities for a specific time  $t$ , update them to reflect the situation at

a future time  $t+dt$  for a specific amount of time  $dt$ . Now, if the particles are sufficiently far from one another and from the walls that no collision will occur before  $t+dt$ , then the calculation is easy: since particles travel in a straight-line trajectory, we use each particle's velocity to update its position. The challenge is to take the collisions into account. One approach, known as *time-driven simulation*, is based on using a fixed value of  $dt$ . To do each update, we need to check all pairs of particles, determine whether or not any two occupy the same position, and then back up to the moment of the first such collision. At that point, we are able to properly update the velocities of the two particles to reflect the collision (using calculations that we will discuss later). This approach is computationally intensive when simulating a large number of particles: if  $dt$  is measured in seconds (fractions of a second, usually), it takes time proportional to  $N^2/dt$  to simulate an  $N$ -particle system for 1 second. This cost is prohibitive (even worse than usual for quadratic algorithms)—in the applications of interest,  $N$  is very large and  $dt$  is very small. The challenge is that if we make  $dt$  too small, the computational cost is high, and if we make  $dt$  too large, we may miss collisions.

**Event-driven simulation.** We pursue an alternative approach that focuses only on those times at which collisions occur. In particular, we are always interested in the *next* collision (because the simple update of all of the particle positions using their velocities is valid until that time). Therefore, we maintain a priority queue of *events*, where an event is a potential collision sometime in the future, either between two particles or between a particle and a wall. The priority associated with each event is its time, so when we *remove the minimum* from the priority queue, we get the next potential collision.

**Collision prediction.** How do we identify potential collisions? The particle velocities provide precisely the information that we need. For example, suppose that we have, at time  $t$ , a particle of radius  $s$  at position  $(r_x, r_y)$  moving with velocity  $(v_x, v_y)$  in the unit box. Consider the vertical wall at  $x=1$  with  $y$  between 0 and 1. Our interest is in the horizontal component of the motion, so we can concentrate on the  $x$ -component of the position  $r_x$  and the  $x$ -component of the velocity  $v_x$ . If  $v_x$  is negative, the particle is not on a collision course with the wall, but if  $v_x$  is positive, there is a potential collision with the wall. Dividing the horizontal distance to the wall ( $1 - s - r_x$ ) by the magnitude of the horizontal component of the velocity ( $|v_x|$ ) we find that the particle will hit the wall after  $dt = (1 - s - r_x)/|v_x|$  time units, when the particle will be at  $(1 - s, r_y + v_y dt)$ , unless it hits some other particle or a horizontal wall before that time. Accordingly, we

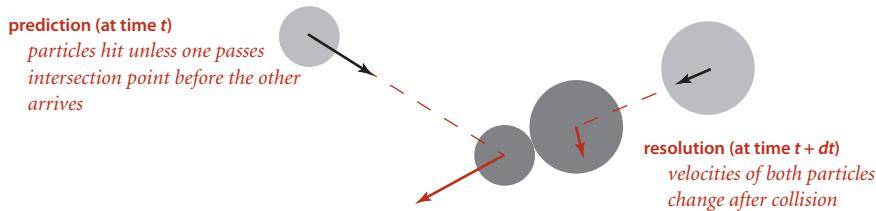




Predicting and resolving a particle-wall collision

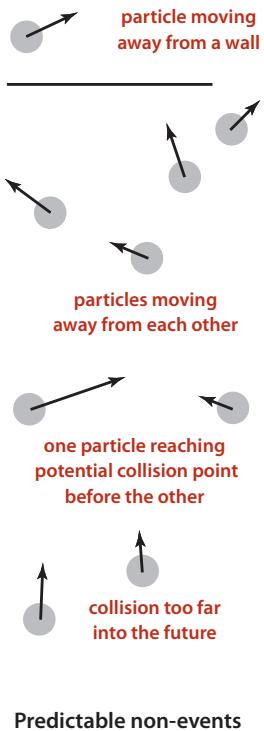
put an entry on the priority queue with priority  $t + dt$  (and appropriate information describing the particle-wall collision event). The collision-prediction calculations for other walls are similar (see EXERCISE 6.1). The calculation for two particles colliding is also similar, but more complicated. Note that it is often the case that the calculation leads to a prediction that the collision will *not* happen (if the particle is moving away from the wall, or if two particles are moving away from one another)—we do not need to put anything on the priority queue in such cases. To handle another typical situation where the predicted collision might be too far in the future to be of interest, we include a parameter `limit` that specifies the time period of interest, so we can also ignore any events that are predicted to happen at a time later than `limit`.

**Collision resolution.** When a collision does occur, we need to resolve it by applying the physical formulas that specify the behavior of a particle after an elastic collision with a reflecting boundary or with another particle. In our example where the particle hits the vertical wall, if the collision does occur, the velocity of the particle will change from  $(v_x, v_y)$  to  $(-v_x, v_y)$  at that time. The collision-resolution calculations for other walls are similar, as are the calculations for two particles colliding, but these are more complicated (see EXERCISE 6.1).



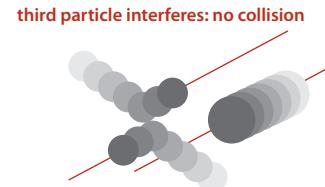
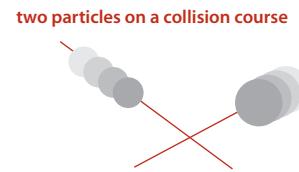
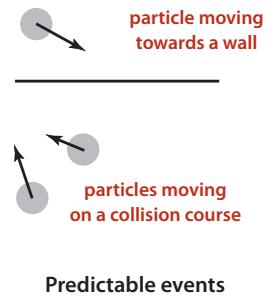
Predicting and resolving a particle-particle collision

**Invalidated events.** Many of the collisions that we predict do not actually happen because some other collision intervenes. To handle this situation, we maintain an instance variable for each particle that counts the number of collisions in which it has been involved. When we remove an event from the priority queue for processing, we check whether the counts corresponding to its particle(s) have changed since the event was created. This approach to handling invalidated collisions is the so-called *lazy* approach:



when a particle is involved in a collision, we leave the now-invalid events associated with it on the priority queue and essentially ignore them when they come off. An alternative approach, the so-called *eager* approach, is to remove from the priority queue all events involving any colliding particle before calculating all of the new potential collisions for that particle. This approach requires a more sophisticated priority queue (that implements the *remove* operation).

THIS DISCUSSION sets the stage for a full event-driven simulation of particles in motion, interacting according to the physical laws of elastic collisions. The software architecture is to encapsulate the implementation in three classes: a `Particle` data type that encapsulates calculations that involve particles, an `Event` data type for predicted events, and a `CollisionSystem` client that does the simulation. The centerpiece of the simulation is a `MinPQ` that contains events, ordered by time. Next, we consider implementations of `Particle`, `Event`, and `CollisionSystem`.



An invalidated event

**Particles.** EXERCISE 6.1 outlines the implementation of a data type particles, based on a direct application of Newton's laws of motion. A simulation client needs to be able to move particles, draw them, and perform a number of calculations related to collisions, as detailed in the following API:

---

<code>public class Particle</code>		
	<code>Particle()</code>	<i>create a new random particle in unit square</i>
	<code>Particle(     double rx, double ry,     double vx, double vy,     double s,     double mass)</code>	<i>create a particle with the given position, velocity, radius, and mass</i>
	<code>void draw()</code>	<i>draw the particle</i>
	<code>void move(double dt)</code>	<i>change position to reflect passage of time dt</i>
	<code>int count()</code>	<i>number of collisions involving this particle</i>
	<code>double timeToHit(Particle b)</code>	<i>time until this particle hits particle b</i>
	<code>double timeToHitHorizontalWall()</code>	<i>time until this particle hits a horizontal wall</i>
	<code>double timeToHitVerticalWall()</code>	<i>time until this particle hits a vertical wall</i>
	<code>void bounceOff(Particle b)</code>	<i>change particle velocities to reflect collision</i>
	<code>void bounceOffHorizontalWall()</code>	<i>change velocity to reflect hitting horizontal wall</i>
	<code>void bounceOffVerticalWall()</code>	<i>change velocity to reflect hitting vertical wall</i>
	<b>API for moving-particle objects</b>	

The three `timeToHit*`() methods all return `Double.POSITIVE_INFINITY` for the (rather common) case when there is no collision course. These methods allow us to predict all future collisions that are associated with a given particle, putting an event on a priority queue corresponding to each one that happens before a given time limit. We use the `bounceOff()` method each time that we process an event that corresponds to two particles colliding to change the velocities (of both particles) to reflect the collision, and the `bounceOff*`() methods for events corresponding to collisions between a particle and a wall.

**Events.** We encapsulate in a private class the description of the objects to be placed on the priority queue (events). The instance variable `time` holds the time when the event is predicted to happen, and the instance variables `a` and `b` hold the particles associated with the event. We have three different types of events: a particle may hit a vertical wall, a horizontal wall, or another particle. To develop a smooth dynamic display of the particles in motion, we add a fourth event type, a redraw event that is a command to draw all the particles at their current positions. A slight twist in the implementation of `Event` is that we use the fact that particle values may be null to encode these four different types of events, as follows:

- Neither `a` nor `b` null: particle-particle collision
- `a` not null and `b` null: collision between `a` and a vertical wall
- `a` null and `b` not null: collision between `b` and a horizontal wall
- Both `a` and `b` null: redraw event (draw all particles)

While not the finest object-oriented programming, this convention is a natural one that enables straightforward client code and leads to the implementation shown below.

```
private static class Event implements Comparable<Event>
{
    private final double time;
    private final Particle a, b;
    private final int countA, countB;

    public Event(double t, Particle a, Particle b)
    { // Create a new event to occur at time t involving a and b.
        this.time = t;
        this.a = a;
        this.b = b;
        if (a != null) countA = a.count(); else countA = -1;
        if (b != null) countB = b.count(); else countB = -1;
    }

    public int compareTo(Event that)
    {
        if (this.time < that.time) return -1;
        else if (this.time > that.time) return +1;
        else return 0;
    }

    public boolean isValid()
    {
        if (a != null && a.count() != countA) return false;
        if (b != null && b.count() != countB) return false;
        return true;
    }
}
```

Event class for particle simulation

A second twist in the implementation of Event is that we maintain the instance variables countA and countB to record the number of collisions involving each of the particles *at the time the event is created*. If these counts are unchanged when the event is removed from the priority queue, we can go ahead and simulate the occurrence of the event, but if one of the counts changes between the time an event goes on the priority queue and the time it leaves, we know that the event has been invalidated and can ignore it. The method `isValid()` allows client code to test this condition.

**Simulation code.** With the computational details encapsulated in `Particle` and `Event`, the simulation itself requires remarkably little code, as you can see in the implementation in the class `CollisionSystem` (see page 863 and page 864). Most of the calculations are encapsulated in the `predictCollisions()` method shown on this page. This method

calculates all potential future collisions involving particle a (either with another particle or with a wall) and puts an event corresponding to each onto the priority queue.

The heart of the simulation is the `simulate()` method shown on page 864. We initialize by calling `predictCollisions()` for each particle to fill the priority queue with the potential collisions involving all particle-wall and all

```
private void predictCollisions(Particle a, double limit)
{
    if (a == null) return;
    for (int i = 0; i < particles.length; i++)
    { // Put collision with particles[i] on pq.
        double dt = a.timeToHit(particles[i]);
        if (t + dt <= limit)
            pq.insert(new Event(t + dt, a, particles[i]));
    }
    double dtX = a.timeToHitVerticalWall();
    if (t + dtX <= limit)
        pq.insert(new Event(t + dtX, a, null));
    double dtY = a.timeToHitHorizontalWall();
    if (t + dtY <= limit)
        pq.insert(new Event(t + dtY, null, a));
}
```

#### Predicting collisions with other particles

particle-particle pairs. Then we enter the main event-driven simulation loop, which works as follows:

- Delete the impending event (the one with minimum priority  $t$ ).
- If the event is invalid, ignore it.
- Advance all particles to time  $t$  on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Use `predictCollisions()` to predict future collisions involving the colliding particle(s) and insert onto the priority queue an event corresponding to each.

This simulation can serve as the basis for computing all manner of interesting properties of the system, as explored in the exercises. For example, one fundamental property

## Event-driven simulation of colliding particles (scaffolding)

```
public class CollisionSystem
{
    private class Event implements Comparable<Event>
    { /* See text. */ }

    private MinPQ<Event> pq;           // the priority queue
    private double t = 0.0;              // simulation clock time
    private Particle[] particles;       // the array of particles

    public CollisionSystem(Particle[] particles)
    { this.particles = particles; }

    private void predictCollisions(Particle a, double limit)
    { /* See text. */ }

    public void redraw(double limit, double Hz)
    { // Redraw event: redraw all particles.
        StdDraw.clear();
        for (int i = 0; i < particles.length; i++) particles[i].draw();
        StdDraw.show(20);
        if (t < limit)
            pq.insert(new Event(t + 1.0 / Hz, null, null));
    }

    public void simulate(double limit, double Hz)
    { /* See next page. */ }

    public static void main(String[] args)
    {
        StdDraw.show(0);
        int N = Integer.parseInt(args[0]);
        Particle[] particles = new Particle[N];
        for (int i = 0; i < N; i++)
            particles[i] = new Particle();
        CollisionSystem system = new CollisionSystem(particles);
        system.simulate(10000, 0.5);
    }
}
```

---

This class is a priority-queue client that simulates the motion of a system of particles over time. The `main()` test client takes a command-line argument  $N$ , creates  $N$  random particles, creates a `CollisionSystem` consisting of the particles, and calls `simulate()` to do the simulation. The instance variables are a priority queue for the simulation, the time, and the particles.

## Event-driven simulation of colliding particles (primary loop)

```

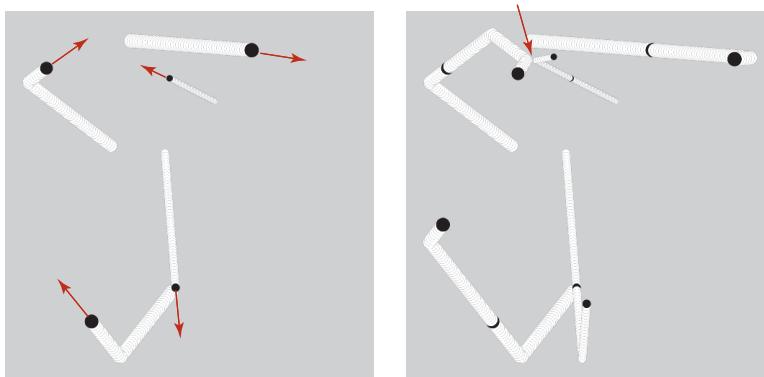
public void simulate(double limit, double Hz)
{
    pq = new MinPQ<Event>();
    for (int i = 0; i < particles.length; i++)
        predictCollisions(particles[i], limit);
    pq.insert(new Event(0, null, null)); // Add redraw event.

    while (!pq.isEmpty())
    { // Process one event to drive simulation.
        Event event = pq.delMin();
        if (!event.isValid()) continue;
        for (int i = 0; i < particles.length; i++)
            particles[i].move(event.time - t); // Update particle positions
        t = event.time; // and time.
        Particle a = event.a, b = event.b;
        if (a != null && b != null) a.bounceOff(b);
        else if (a != null && b == null) a.bounceOffVerticalWall();
        else if (a == null && b != null) b.bounceOffHorizontalWall();
        else if (a == null && b == null) redraw(limit, Hz);
        predictCollisions(a, limit);
        predictCollisions(b, limit);
    }
}

```

This method represents the main event-driven simulation. First, the priority queue is initialized with events representing all predicted future collisions involving each particle. Then the main loop takes an event from the queue, updates time and particle positions, and adds new events to reflect changes.

% java CollisionSystem 5



of interest is the amount of pressure exerted by the particles against the walls. One way to calculate the pressure is to keep track of the number and magnitude of wall collisions (an easy computation based on particle mass and velocity) so that we can easily compute the total. Temperature involves a similar calculation.

**Performance.** As described at the outset, our interest in event-driven simulation is to avoid the computationally intensive inner loop intrinsic in time-driven simulation.

**Proposition A.** An event-driven simulation of  $N$  colliding particles requires at most  $N^2$  priority queue operations for initialization, and at most  $N$  priority queue operations per collision (with one extra priority queue operation for each invalid collision).

**Proof** Immediate from the code.

Using our standard guaranteed-logarithmic-time-per operation priority-queue implementation from SECTION 2.4, the time needed per collision is linearithmic. Simulations with large numbers of particles are therefore quite feasible.

EVENT-DRIVEN SIMULATION applies to countless other domains that involve physical modeling of moving objects, from molecular modeling to astrophysics to robotics. Such applications may involve extending the model to add other kinds of bodies, to operate in three dimensions, to include other forces, and in many other ways. Each extension involves its own computational challenges. This event-driven approach results in a more robust, accurate, and efficient simulation than many other alternatives that we might consider, and the efficiency of the heap-based priority queue enables calculations that might not otherwise be possible.

Simulation plays a vital role in helping researchers to understand properties of the natural world in all fields of science and engineering. Applications ranging from manufacturing processes to biological systems to financial systems to complex engineered structures are too numerous to even list here. For a great many of these applications, the extra efficiency afforded by the heap-based priority queue data type or an efficient sorting algorithm can make a substantial difference in the quality and extent that are possible in the simulation.

**B-trees** In CHAPTER 3, we saw that algorithms that are appropriate for accessing items from huge collections of data are of immense practical importance. Searching is a fundamental operation on huge data sets, and such searching consumes a significant fraction of the resources used in many computing environments. With the advent of the web, we have the ability to access a vast amount of information that might be relevant to a task—our challenge is to be able to search through it efficiently. In this section, we describe a further extension of the balanced-tree algorithms from SECTION 3.3 that can support *external search* in symbol tables that are kept on a disk or on the web and are thus potentially far larger than those we have been considering (which have to fit in addressable memory). Modern software systems are blurring the distinction between local files and web pages, which may be stored on a remote computer, so the amount of data that we might wish to search is virtually unlimited. Remarkably, the methods that we shall study can support search and insert operations on symbol tables containing trillions of items or more using only four or five references to small blocks of data.

**Cost model.** Data storage mechanisms vary widely and continue to evolve, so we use a simple model to capture the essentials. We use the term *page* to refer to a contiguous block of data and the term *probe* to refer to the first access to a page. We assume that accessing a page involves reading its contents into local memory, so that subsequent accesses are relatively inexpensive. A page could be a file on your local computer or a web page on a distant computer or part of a file on a server, or whatever. Our goal is to develop search implementations that use a small number of probes to find any given key. We avoid making specific assumptions about the page size and about the ratio of the time required for a probe (which presumably requires communicating with a distant device) to the time required, subsequently, to access items within the block (which presumably happens in a local processor). In typical situations, these values are likely to be on the order of 100 or 1,000 or 10,000; we do not need to be more precise because the algorithms are not highly sensitive to differences in the values in the ranges of interest.

**B-tree cost model.** When studying algorithms for external searching, we count *page accesses* (the number of times a page is accessed, for read or write).

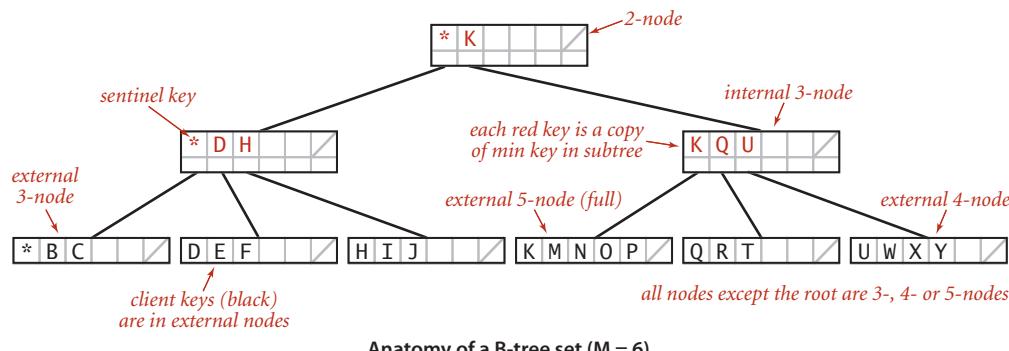
**B-trees.** The approach is to extend the 2-3 tree data structure described in SECTION 3.3, with a crucial difference: rather than store the data in the tree, we build a tree with *copies* of the keys, each key copy associated with a link. This approach enables us to more easily separate the index from the table itself, much like the index in a book. As with 2-3 trees, we enforce upper and lower bounds on the number of key-link pairs

that can be in each node: we choose a parameter  $M$  (an even number, by convention) and build multiway trees where every node must have *at most*  $M - 1$  key-link pairs (we assume that  $M$  is sufficiently small that an  $M$ -way node will fit on a page) and *at least*  $M/2$  key-link pairs (to provide the branching that we need to keep search paths short), except possibly the root, which can have fewer than  $M/2$  key-link pairs but must have at least 2. Such trees were named *B-trees* by Bayer and McCreight, who, in 1970, were the first researchers to consider the use of multiway balanced trees for external searching. Some people reserve the term *B-tree* to describe the exact data structure built by the algorithm suggested by Bayer and McCreight; we use it as a generic term for data structures based on multiway balanced search trees with a fixed page size. We specify the value of  $M$  by using the terminology “*B-tree of order  $M$* .” In a B-tree of order 4, each node has at most 3 and at least 2 key-link pairs; in a B-tree of order 6, each node has at most 5 and at least 3 key-link pairs (except possibly the root, which could have 2 key-link pairs), and so forth. The reason for the exception at the root for larger  $M$  will become clear when we consider the construction algorithm in detail.

**Conventions.** To illustrate the basic mechanisms, we consider an (ordered) SET implementation (with keys and no values). Extending to provide an ordered ST to associate keys with values is an instructive exercise (see EXERCISE 6.16). Our goal is to support `add()` and `contains()` for a set of keys that could be huge. We use ordered keys because we are generalizing search trees, which are based on ordered keys. Extending our implementation to support other ordered operations is also an instructive exercise. In external searching applications, it is common to keep the index separate from the data. For B-trees, we do so by using two different kinds of nodes:

- *Internal* nodes, which associate copies of keys with pages
- *External* nodes, which have references to the actual data

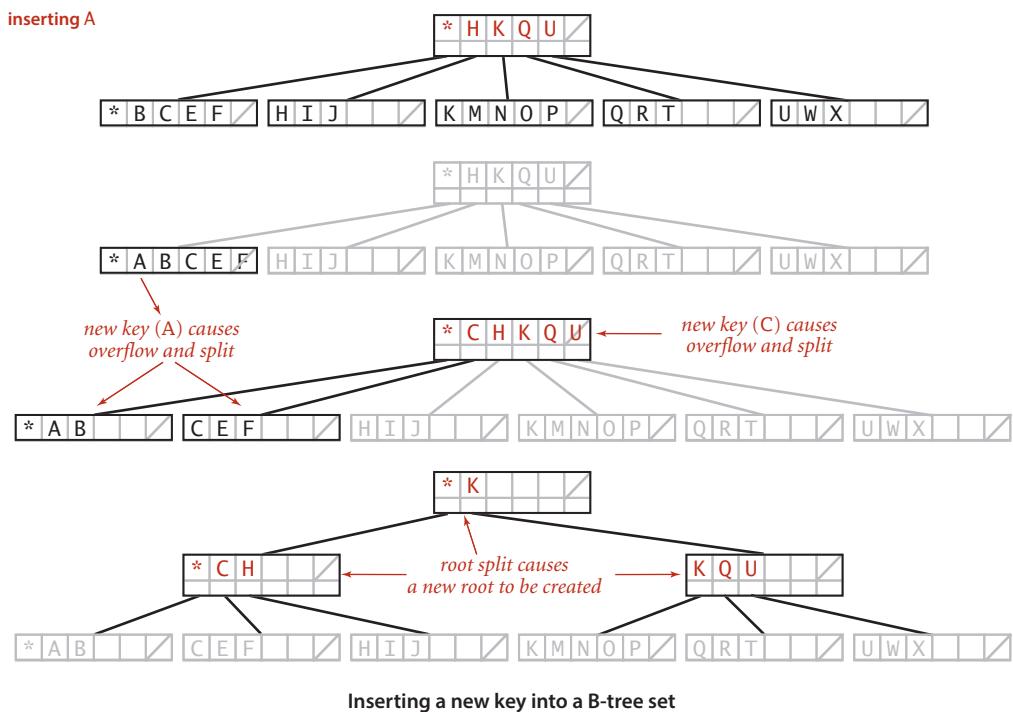
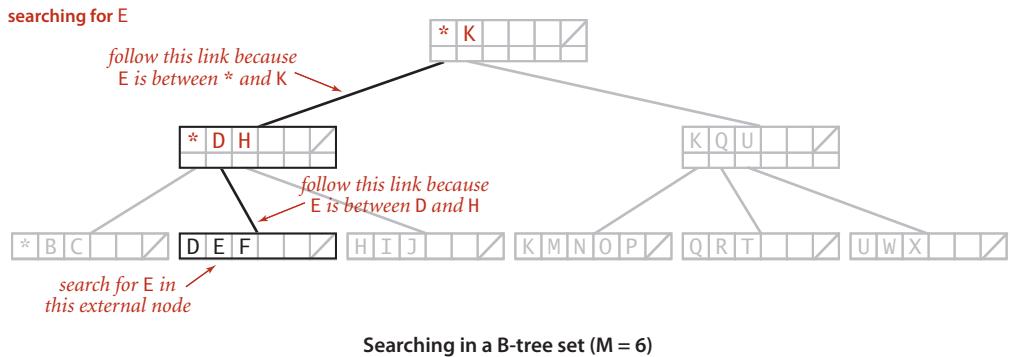
Every key in an internal node is associated with another node that is the root of a tree containing all keys *greater than or equal to* that key and *less than* the next largest key, if



any. It is convenient to use a special key, known as a *sentinel*, that is defined to be less than all other keys, and to start with a root node containing that key. The symbol table does not contain duplicate keys, but we use copies of keys (in internal nodes) to guide the search. (In our examples, we use single-letter keys and the character \* as the sentinel that is less than all other keys.) These conventions simplify the code somewhat and thus represent a convenient (and widely used) alternative to mixing all the data with links in the internal nodes, as we have done for other search trees.

**Search and insert.** Search in a B-tree is based on recursively searching in the unique subtree that could contain the search key. Every search ends in an external node that contains the key if and only if it is in the set. We might also terminate a *search hit* when encountering a copy of the search key in an internal node, but we always search to an external node because doing so simplifies extending the code to an ordered symbol-table implementation (also, this event rarely happens when  $M$  is large). To be specific, consider searching in a B-tree of order 6: it consists of 3-nodes with 3 key-link pairs, 4-nodes with 4 key-link pairs, and 5-nodes with 5 key-link pairs, with possibly a 2-node at the root. To search, we start at the root and move from one node to the next by finding the proper interval for the search key in the current node and then exiting through the corresponding link to get to the next node. Eventually, the search process leads us to a page containing keys at the bottom of the tree. We terminate the search with a search hit if the search key is in that page; we terminate with a search miss if it is not. As with 2-3 trees, we can use recursive code to insert a new key at the bottom of the tree. If there is no room for the key, we allow the node at the bottom to temporarily overflow (become a 6-node) and then split 6-nodes on the way up the tree, after the recursive call. If the root is an 6-node, we split it into a 2-node connected to two 3-nodes; elsewhere in the tree, we replace any  $k$ -node attached to a 6-node by a  $(k+1)$ -node attached to two 3-nodes. Replacing 3 by  $M/2$  and 6 by  $M$  in this description converts it into a description of search and insert for B-trees of order  $M$  and leads to the following definition:

**Definition.** A *B-tree of order  $M$*  (where  $M$  is an even positive integer) is a tree that either is an external  $k$ -node (with  $k$  keys and associated information) or comprises internal  $k$ -nodes (each with  $k$  keys and  $k$  links to B-trees representing each of the  $k$  intervals delimited by the keys), having the following structural properties: every path from the root to an external node must be the same length (perfect balance); and  $k$  must be between 2 and  $M - 1$  at the root and between  $M/2$  and  $M - 1$  at every other node.



**Representation.** As just discussed, we have a great deal of freedom in choosing concrete representations for nodes in B-trees. We encapsulate these choices in a Page API that associates keys with links to Page objects and supports the operations that we need to test for overfull pages, split them, and distinguish between internal and external pages. You can think of a Page as a symbol table, kept externally (in a file on your computer or on the web). The terms *open* and *close* in the API refer to the process of bringing an external page into internal memory and writing its contents back out (if necessary). The add() method for internal pages is a symbol-table operation that associates the given page with the minimum key in the tree rooted at that page. The add() and contains() methods for external pages are like their corresponding SET operations. The workhorse of any implementation is the split() method, which splits a full page by moving the  $M/2$  key-value pairs of rank greater than  $M/2$  to a new Page and returns a reference to that page. EXERCISE 6.15 discusses an implementation of Page using BinarySearchST, which implements B-trees in memory, like our other search implementations. On some systems, this might suffice as an external searching implementation because a virtual-memory system might take care of disk references. More typical practical implementations might involve hardware-specific code that reads and

---

public class Page<Key>	
Page(boolean bottom)	<i>create and open a page</i>
void close()	<i>close a page</i>
void add(Key key)	<i>add key into the (external) page</i>
void add(Page p)	<i>open p and add an entry into this (internal) page that associates the smallest key in p with p</i>
boolean isExternal()	<i>is this page external?</i>
boolean contains(Key key)	<i>is key in the page?</i>
Page next(Key key)	<i>the subtree that could contain the key</i>
boolean isFull()	<i>has the page overflowed?</i>
Page split()	<i>move the highest-ranking half of the keys in the page to a new page</i>
Iterable<Key> keys()	<i>iterator for the keys on the page</i>

API for a B-tree page

writes pages. EXERCISE 6.19 encourages you to think about implementing Page using web pages. We ignore such details here in the text to emphasize the utility of the B-tree concept in a broad variety of settings.

With these preparations, the code for BTREESET onpage 872 is remarkably simple. For `contains()`, we use a recursive method that takes a `Page` as argument and handles three cases:

- If the page is external and the key is in the page, return `true`.
- If the page is external and the key is not in the page, return `false`.
- Otherwise, do a recursive call for the subtree that could contain the key.

For `add()` we use the same recursive structure, but insert the key at the bottom if it is not found during the search and then split any full nodes on the way up the tree.

**Performance.** The most important property of B-trees is that for reasonable values of the parameter  $M$  the search cost is *constant*, for all practical purposes:

**Proposition B.** A search or an insertion in a B-tree of order  $M$  with  $N$  items requires between  $\log_M N$  and  $\log_{M/2} N$  probes—a constant number, for practical purposes.

**Proof** This property follows from the observation that all the nodes in the interior of the tree (nodes that are not the root and are not external) have between  $M/2$  and  $M - 1$  links, since they are formed from a split of a full node with  $M$  keys and can only grow in size (when a child is split). In the best case, these nodes form a complete tree of branching factor  $M - 1$ , which leads immediately to the stated bound. In the worst case, we have a root with two entries each of which refers to a complete tree of degree  $M/2$ . Taking the logarithm to the base  $M$  results in a very small number—for example, when  $M$  is 1,000, the height of the tree is less than 4 for  $N$  less than 62.5 billion.

In typical situations, we can reduce the cost by one probe by keeping the root in internal memory. For searching on disk or on the web, we might take this step explicitly before embarking on any application involving a huge number of searches; in a virtual memory with caching, the root node will be the one most likely to be in fast memory, because it is the most frequently accessed node.

**Space.** The space usage of B-trees is also of interest in practical applications. By construction, the pages are at least half full, so, in the worst case, B-trees use about double the space that is absolutely necessary for keys, plus extra space for links. For random keys, A. Yao proved in 1979 (using mathematical analysis that is beyond the scope of

**ALGORITHM 6.1 B-tree set implementation**

```
public class BTreeSET<Key extends Comparable<Key>>
{
    private Page root = new Page(true);

    public BTreeSET(Key sentinel)
    { add(sentinel); }

    public boolean contains(Key key)
    { return contains(root, key); }

    private boolean contains(Page h, Key key)
    {
        if (h.isExternal()) return h.contains(key);
        return contains(h.next(key), key);
    }

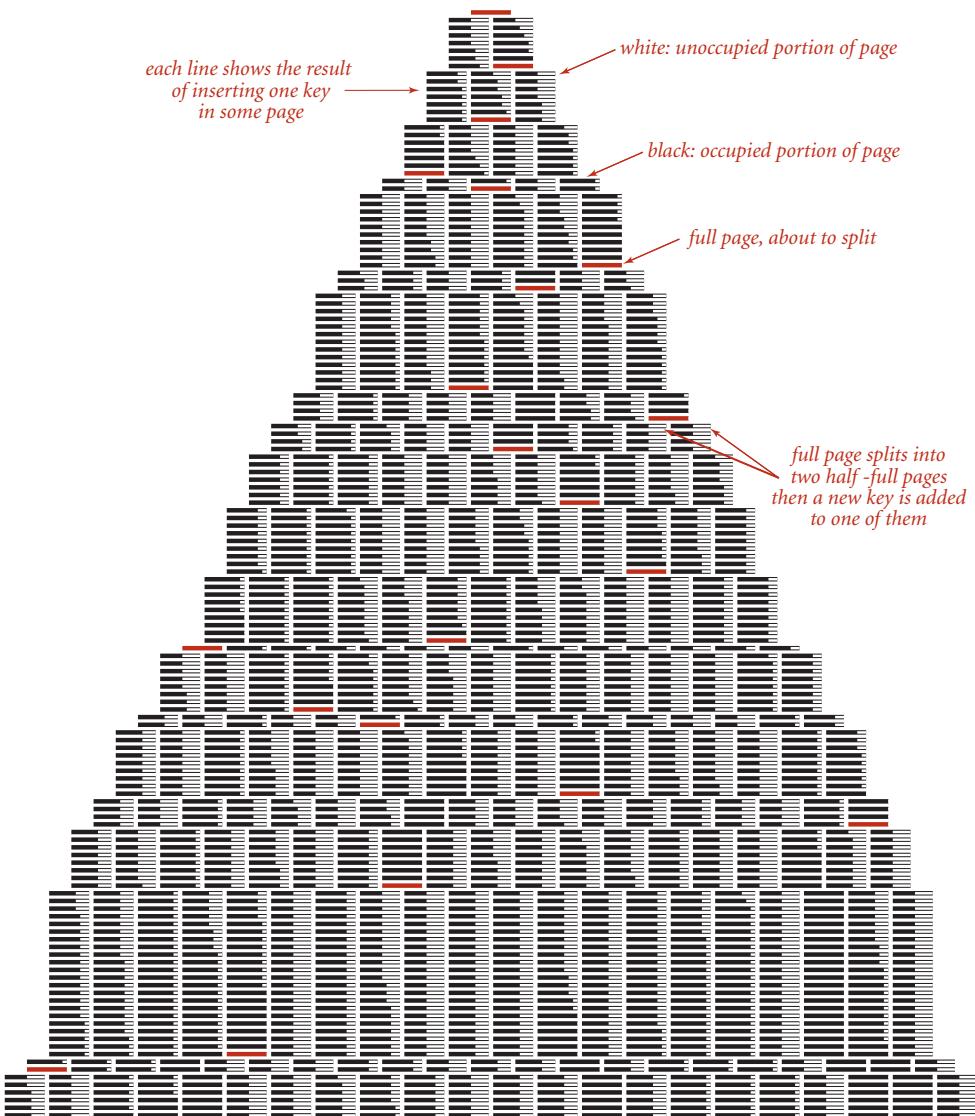
    public void add(Key key)
    {
        add(root, key);
        if (root.isFull())
        {
            Page lefthalf = root;
            Page righthalf = root.split();
            root = new Page(false);
            root.add(lefthalf);
            root.add(righthalf);
        }
    }

    public void add(Page h, Key key)
    {
        if (h.isExternal()) { h.add(key); return; }

        Page next = h.next(key);
        add(next, key);
        if (next.isFull())
            h.add(next.split());
        next.close();
    }
}
```

---

This B-tree implementation implements multiway balanced search trees as described in the text, using a `Page` data type that supports search by associating keys with subtrees that could contain the key and supports insertion by including a test for overflow and a page split method.



Building a large B-tree

this book) that the average number of keys in a node is about  $M \ln 2$ , so about 44 percent of the space is unused. As with many other search algorithms, this random model reasonably predicts results for key distributions that we observe in practice.

THE IMPLICATIONS OF PROPOSITION B ARE PROFOUND and worth contemplating. Would you have guessed that you can develop a search implementation that can guarantee a cost of four or five probes for search and insert in files as large as you can reasonably contemplate needing to process? B-trees are widely used because they allow us to achieve this ideal. In practice, the primary challenge to developing an implementation is ensuring that space is available for the B-tree nodes, but even that challenge becomes easier to address as available storage space increases on typical devices.

Many variations on the basic B-tree abstraction suggest themselves immediately. One class of variations saves time by packing as many page references as possible in internal nodes, thereby increasing the branching factor and flattening the tree. Another class of variations improves storage efficiency by combining nodes with siblings before splitting. The precise choice of variant and algorithm parameter can be engineered to suit particular devices and applications. Although we are limited to getting a small constant factor improvement, such an improvement can be of significant importance for applications where the table is huge and/or huge numbers of transactions are involved, precisely the applications for which B-trees are so effective.

**Suffix arrays** Efficient algorithms for string processing play a critical role in commercial applications and in scientific computing. From the countless strings that define web pages that are searched by billions of users to the extensive genomic databases that scientists are studying to unlock the secret of life, computing applications of the 21st century are increasingly string-based. As usual, some classic algorithms are effective, but remarkable new algorithms are being developed. Next, we describe a data structure and an API that support some of these algorithms. We begin by describing a typical (and a classic) string-processing problem.

**Longest repeated substring.** What is the longest substring that appears at least twice in a given string? For example, the longest repeated substring in the string "to be or not to be" is the string "to be". Think briefly about how you might solve it. Could you find the longest repeated substring in a string that has millions of characters? This problem is simple to state and has many important applications, including data compression, cryptography, and computer-assisted music analysis. For example, a standard technique used in the development of large software systems is *refactoring code*. Programmers often put together new programs by cutting and pasting code from old programs. In a large program built over a long period of time, replacing duplicate code by function calls to a single copy of the code can make the program much easier to understand and maintain. This improvement can be accomplished by finding long repeated substrings in the program. Another application is found in computational biology. Are substantial identical fragments to be found within a given genome? Again, the basic computational problem underlying this question is to find the longest repeated substring in a string. Scientists are typically interested in more detailed questions (indeed, the nature of the repeated substrings is precisely what scientists seek to understand), but such questions are certainly no easier to answer than the basic question of finding the longest repeated substring.

**Brute-force solution.** As a warmup, consider the following simple task: given two strings, find their longest common *prefix* (the longest substring that is a prefix of both strings). For example, the longest common prefix of acctgttaac and accgttaa is acc. The code at right is a useful starting point for addressing more complicated tasks: it takes time proportional to the length of the match. Now, how do we find the longest repeated substring in a given string? With `lcp()`, the following

```
private static int lcp(String s, String t)
{
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i)) return i;
    return N;
}
```

Longest common prefix of two strings

brute-force solution immediately suggests itself: we compare the substring starting at each string position  $i$  with the substring starting at each other starting position  $j$ , keeping track of the longest match found. This code is not useful for long strings, because its running time is at least *quadratic* in the length of the string: the number of different pairs  $i$  and  $j$  is  $N(N-1)/2$ , so the number of calls on `lcp()` for this approach would be  $\sim N^2/2$ . Using this solution for a genomic sequence with millions of characters would require trillions of `lcp()` calls, which is infeasible.

**Suffix sort solution.** The following clever approach, which takes advantage of sorting in an unexpected way, is an effective way to find the longest repeated substring, even in a huge string: we make an array of the  $N$  suffixes of  $s$  (the substrings starting at each position and going to the end), and then we sort this array. The key to the algorithm's correctness is that every substring appears somewhere as a prefix of one of the suffixes in the array. After sorting, the longest repeated substrings will appear in adjacent positions in the array. Thus, we can make a single pass through the sorted array, keeping track of the longest matching prefixes between adjacent strings. The key to the algorithm's efficiency is to form the  $N$  suffixes implicitly (storing only the original string and the index of the first character in each suffix) instead of explicitly (since that would require quadratic time and space). This suffix sorting approach is significantly more efficient than the brute-force method, but before implementing and analyzing it, we consider another application of suffix sorting.

```

input string
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
 a a c a a g t t t a c a a g c

suffixes
0 a a c a a g t t t a c a a g c
1 a c a a g t t t a c a a g c
2 c a a g t t t a c a a g c
3 a a g t t t a c a a g c
4 a g t t t a c a a g c
5 g t t t a c a a g c
6 t t t a c a a g c
7 t t a c a a g c
8 t a c a a g c
9 a c a a g c
10 c a a g c
11 a a g c
12 a g c
13 g c
14 c

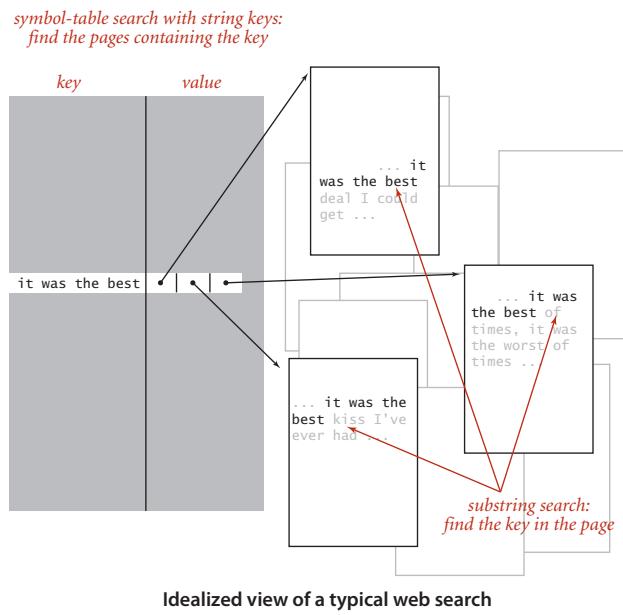
sorted suffixes
0 a a c a a g t t t a c a a g c
11 a a g c
3 a a g t t t a c a a g c
9 a c a a g c
1 a c a a g t t t a c a a g c
12 a g c
4 a g t t t a c a a g c
14 c
10 c a a g c
2 c a a g t t t a c a a g c
13 g c
5 g t t t a c a a g c
8 t a c a a g c
7 t t a c a a g c
6 t t t a c a a g c

longest repeated substring
 1           9
a a c a a g t t t a c a a g c

```

Computing the LRS by sorting suffixes

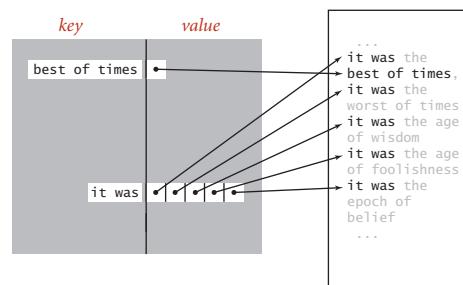
**Indexing a string.** When you are trying to find a particular substring within a large text—for example, while working in a text editor or within a page you are viewing with a browser—you are doing a *substring search*, the problem we considered in SECTION 5.3. For that problem, we assume the text to be relatively large and focus on preprocessing the *substring*, with the goal of being able to efficiently find that substring in any given text. When you type search keys into your web browser, you are doing a *search with string keys*, the subject of SECTION 5.2. Your search engine must precompute an index, since it cannot afford to scan all the pages in the web for your keys. As we discussed in SECTION 3.5 (see `FileIndex` on page 501), this would ideally be an inverted index associating each possible search string with all web pages that contain it—a symbol table where each entry is a string key and each value is a set of pointers (each pointer giving the information necessary to locate an occurrence of the key on the web—perhaps a URL that names a web page and an integer offset within that page). In practice, such a symbol table would be far too big, so your search engine uses various sophisticated algorithms to reduce its size. One approach is to rank web pages by importance (perhaps using an algorithm like the PageRank algorithm that we discussed on page 502) and work only with highly-ranked pages, not all pages. Another approach to cutting down on the size of a symbol table to support search with string keys is to associate URLs with *words* (substrings delimited by whitespace) as keys in the precomputed index. Then, when you search for a word, the search engine can use the index to find the (important) pages containing your search keys (words) and then use substring search within each page to find them. But with this approach, if the text were to contain "everything" and you were to search for "thing", you would not find it. For some applications, it *is* worthwhile to build an index to help find *any substring* within a given text. Doing so might be justified for a linguistic study of an important piece of literature, for a genomic sequence that might be an object of study for many scientists, or just for a widely accessed web page. Again,



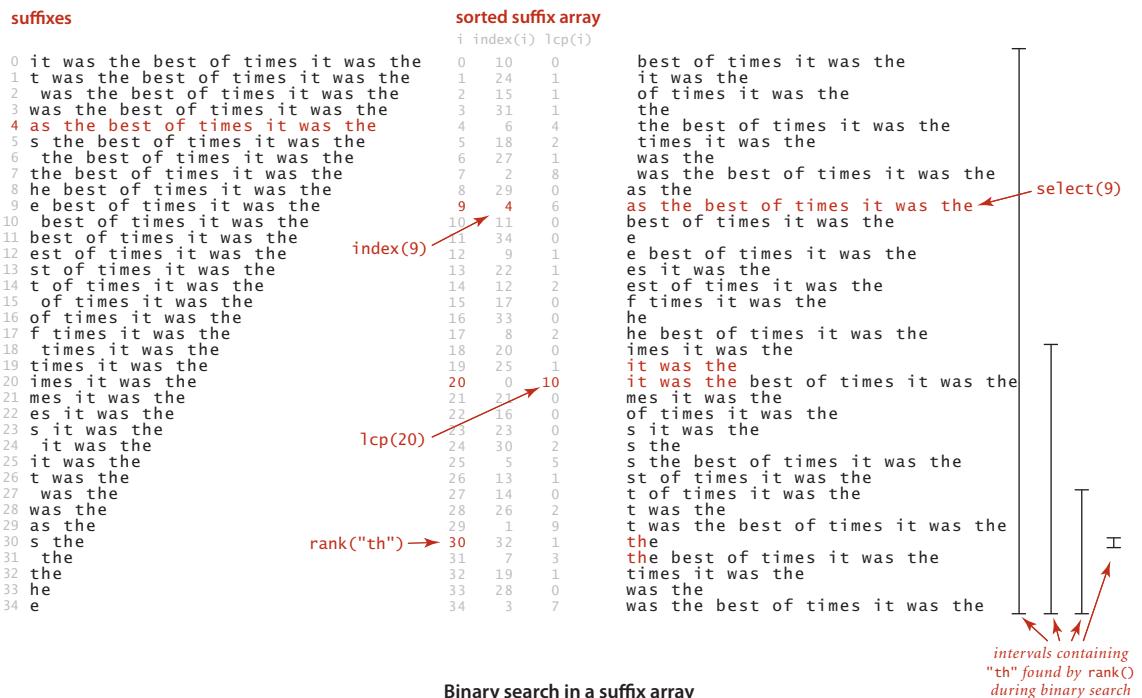
Idealized view of a typical web search

whitespace) as keys in the precomputed index. Then, when you search for a word, the search engine can use the index to find the (important) pages containing your search keys (words) and then use substring search within each page to find them. But with this approach, if the text were to contain "everything" and you were to search for "thing", you would not find it. For some applications, it *is* worthwhile to build an index to help find *any substring* within a given text. Doing so might be justified for a linguistic study of an important piece of literature, for a genomic sequence that might be an object of study for many scientists, or just for a widely accessed web page. Again,

ideally, the index would associate all possible substrings of the text string with each position where it occurs in the text string, as depicted at right. The basic problem with this ideal is that the number of possible substrings is too large to have a symbol-table entry for each of them (an  $N$ -character text has  $N(N+1)/2$  substrings). The table for the example at right would need entries for b, be, bes, best, best o, best of, e, es, est, est o, est of, s, st, st o, st of, t, t o, t of, o, of, and many, many other substrings. Again, we can use a suffix sort to address this problem in a manner analogous to our first symbol-table implementation using binary search, in SECTION 3.1. We consider each of the  $N$  suffixes to be keys, create a sorted array of our keys (the suffixes), and use binary search to search in that array, comparing the search key with each suffix.



Idealized view of a text-string index



**API and client code.** To support client code to solve these two problems, we articulate the API shown below. It includes a constructor; a `length()` method; methods `select()` and `index()`, which give the string and index of the suffix of a given rank in the sorted list of suffixes; a method `lcp()` that gives the length of the longest common prefix of each suffix and the one preceding it in the sorted list; and a method `rank()` that gives the number of suffixes less than the given key (just as we have been using since we first examined binary search in CHAPTER 1). We use the term *suffix array* to describe the abstraction of a sorted list of suffix strings, without committing to use an array of strings as the underlying data structure.

---

<code>public class SuffixArray</code>	
	<code>SuffixArray(String text)</code> <i>build suffix array for text</i>
<code>int length()</code>	<i>length of text</i>
<code>String select(int i)</code>	<i>i<sup>th</sup> in the suffix array (i between 0 and N-1)</i>
<code>int index(int i)</code>	<i>index of select(i) (i between 0 and N-1)</i>
<code>int lcp(int i)</code>	<i>length of longest common prefix of select(i) and select(i-1) (i between 1 and N-1)</i>
<code>int rank(String key)</code>	<i>number of suffixes less than key</i>
	<b>Suffix array API</b>

---

In the example on the facing page, `select(9)` is "as the best of times...", `index(9)` is 4, `lcp(20)` is 10 because "it was the best of times..." and "it was the" have the common prefix "it was the" which is of length 10, and `rank("th")` is 30. Note also that the `select(rank(key))` is the first possible suffix in the sorted suffix list that has key as prefix and that all other occurrences of key in the text immediately follow (see the figure on the opposite page). With this API, the client code on the next two pages is immediate. LRS (page 880) finds the longest repeated substring in the text on standard input by building a suffix array and then scanning through the sorted suffixes to find the maximum `lcp()` value. KWIC (page 881) builds a suffix array for the text named as command-line argument, takes queries from standard input, and prints all occurrences of each query in the text (including a specified number of characters before and after to give context). The name KWIC stands for *keyword-in-context* search, a term dating at least to the 1960s. The simplicity and efficiency of this client code for these typical string-processing applications is remarkable, and testimony to the importance of careful API design (and the power of a simple but ingenious idea).

```
public class LRS
{
    public static void main(String[] args)
    {
        String text = StdIn.readAll().replaceAll("\\s+", " ");
        int N = text.length();
        SuffixArray sa = new SuffixArray(text);
        String lrs = "";
        for (int i = 1; i < N; i++)
        {
            int length = sa.lcp(i);
            if (length > lrs.length())
                lrs = text.substring(sa.index(i), sa.index(i) + length);
        }
        StdOut.println("'" + lrs + "'");
    }
}
```

Longest repeated substring client

```
% more tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair

% java LRS < tinyTale.txt
'st of times it was the '

% java LRS < mobyDick.txt
',- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th'
```

```

public class KWIC
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int context = Integer.parseInt(args[1]);

        String text = in.readAll().replaceAll("\\s+", " ");
        int N = text.length();
        SuffixArray sa = new SuffixArray(text);

        while (StdIn.hasNextLine())
        {
            String query = StdIn.readLine();
            for (int i = sa.rank(query); i < N; i++)
            {
                // Check if sorted suffix i is a match.
                int from1 = sa.index(i);
                int to1 = Math.min(N, sa.index(i) + query.length());
                if (!query.equals(text.substring(from1, to1))) break;

                // Print context surrounding sorted suffix i.
                int from2 = Math.max(0, sa.index(i) - context);
                int to2 = Math.min(N, sa.index(i) + context + query.length());
                StdOut.println(text.substring(from2, to2));
            }
            StdOut.println();
        }
    }
}

```

Keyword-in-context indexing client

```

% java KWIC tale.txt 15
search
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
dispersing in search of other carri
n that bed and search the straw hold

better thing
t is a far far better thing that i do than
some sense of better things else forgotte
was capable of better things mr carton ent

```

**Implementation.** The code on the facing page is an elementary implementation of the SuffixArray API. The key to the implementation is a nested class `Suffix` that represents a suffix of a text string. A `Suffix` has two instance variables: a `String` reference to the text string and an `int` index of its first character. It provides four utility methods: `length()` returns the length of the suffix; `charAt(i)` returns the `i`th character in the suffix; `toString()` returns a string representation of the suffix; and `compareTo()` compares two suffixes, for use in sorting. Using this nested class, it is straightforward to complete the implementation. The constructor builds an array of `Suffix` objects and sorts them, so `index(i)` just returns the index associated with `suffixes[i]`. The implementations of `length()` and `select()` are also one-liners. The implementation of `lcp()` is similar to the `lcp()` on page 875, and `rank()` is virtually the same as our implementation of binary search for symbol tables, on page 381. Again, the simplicity and elegance of this implementation should not mask the fact that it is a sophisticated algorithm that enables the solution of important problems like the longest repeated substring problem that would otherwise seem to be infeasible.

**Performance.** The efficiency of our suffix sorting implementation depends on the fact that we form the suffixes implicitly—each suffix is represented by a reference to the text string and the index of its first character. Thus, the space to store the array of suffixes is linear in the length of the text string. This point is a bit counterintuitive because the total number of characters in the  $N$  suffixes is  $\sim N^2/2$ , a quadratic function of the length of the string. Moreover, that quadratic factor gives one pause when considering the cost of sorting the array of suffixes. It is very important to bear in mind that this approach is effective for long strings because of our implicit representation for suffixes: when we exchange two suffixes, we are exchanging only references, not the whole suffixes. Now,

```
public int compareTo(Suffix that)
{
    if (this == that) return 0;
    int N = Math.min(this.length(), that.length());
    for (int i = 0; i < N; i++)
    {
        if (this.charAt(i) < that.charAt(i)) return -1;
        if (this.charAt(i) > that.charAt(i)) return +1;
    }
    return this.length() - that.length();
}
```

Comparing two suffixes

the cost of comparing two suffixes may be proportional to the length of the suffixes in the case when their common prefix is very long, but most comparisons in typical applications involve only a few characters. If so, the running time of the suffix sort is linearithmic.

**ALGORITHM 6.2 Suffix array (elementary implementation)**

```
import java.util.Arrays;
public class SuffixArray
{
    private Suffix[] suffixes; // array of suffixes
    public SuffixArray(String text)
    {
        int N = text.length();
        this.suffixes = new Suffix[N];
        for (int i = 0; i < N; i++)
            suffixes[i] = new Suffix(text, i);
        Arrays.sort(suffixes);
    }
    private static class Suffix implements Comparable<Suffix>
    {
        private final String text; // reference to text string
        private final int index; // index of suffix's first character
        private Suffix(String s, int index)
        {
            this.text = text;
            this.index = index;
        }
        private int length() { return text.length() - index; }
        private char charAt(int i) { return text.charAt(index + i); }
        public String toString() { return text.substring(index); }
        public int compareTo(Suffix that) // See page 882.
    }
    public int index(int i) { return suffixes[i].index; }
    public int length() { return suffixes.length; }
    public String select(int i) { return suffixes[i].toString(); }

    public int lcp(int i) // See Exercise 6.28.
    public int rank(String key) // See Exercise 6.28.
}
```

---

This implementation of our `SuffixArray` API depends for its efficiency on the fact that the suffixes are represented implicitly (see text), using the nested class `Suffix`.

For example, in many applications, it is reasonable to use a random string model:

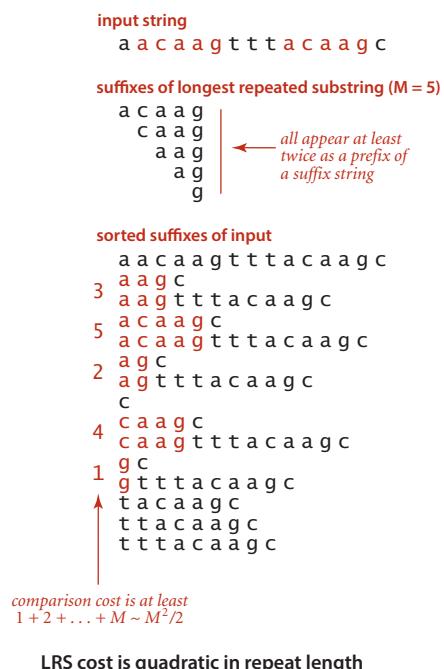
**Proposition C.** Using 3-way string quicksort, we can build a suffix array from a random string of length  $N$  with space proportional to  $N$  and  $\sim 2N\ln N$  character compares, on the average.

**Discussion:** The space bound is immediate, but the time bound follows from a detailed and difficult research result by P. Jacquet and W. Szpankowski, which implies that the cost of sorting the suffixes is asymptotically the same as the cost of sorting  $N$  random strings (see PROPOSITION E on page 723).

**Improved implementations.** Our elementary implementation of SuffixArray (ALGORITHM 6.2) has poor worst-case performance. For example, if all the characters are equal, the sort examines every character in each suffix and thus takes *quadratic* time. For strings of the type we have been using as examples, such as genomic sequences or natural-language text, this is not likely to be problematic, but the algorithm can be slow for texts with long runs of identical characters. Another way of looking at the problem is to observe that the cost of finding the longest repeated substring is (at least) *quadratic in the length of the longest repeated substring* because all of the prefixes of the repeat need to be checked (see the diagram at right). This is not a problem for a text such as *A Tale of Two Cities*, where the longest repeated substring

"s dropped because it would have  
been a bad thing for me in a  
worldly point of view i"

has just 84 characters, but it is a serious problem for genomic data, where long repeated substrings are not unusual. How can this quadratic behavior for repeat searching be avoided? Remarkably, research by P. Weiner in 1973 showed that *it is possible to solve the longest repeated substring problem in guaranteed linear time*. Weiner's algorithm was based on building a suffix tree data structure (essentially a



trie for suffixes). With multiple pointers per character, suffix trees consume too much space for many practical problems, which led to the development of suffix arrays. In the 1990s, U. Manber and E. Myers presented a linearithmic algorithm for building suffix arrays directly and a method that does preprocessing at the same time as the suffix sort to support *constant-time* `lcp()`. Several linear-time suffix sorting algorithms have been developed since. With a bit more work, the Manber-Myers implementation can also support a two-argument `lcp()` that finds the longest common prefix of two given suffixes that are not necessarily adjacent in guaranteed constant time, again a remarkable improvement over the straightforward implementation. These results are quite surprising, as they achieve efficiencies quite beyond what you might have expected.

**Proposition D.** With suffix arrays, we can solve both the suffix sorting and longest repeated substring problems in linear time.

**Proof:** The remarkable algorithms for these tasks are just beyond our scope, but you can find on the booksite code that implements the `SuffixArray` constructor in linear time and `lcp()` queries in constant time.

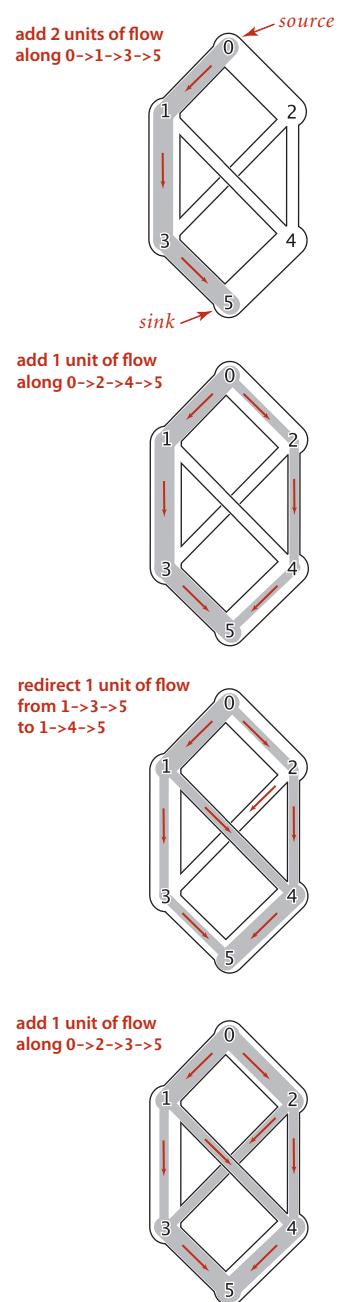
A `SuffixArray` implementation based on these ideas supports efficient solutions of numerous string-processing problems, with simple client code, as in our LRS and KWIC examples.

SUFFIX ARRAYS ARE THE CULMINATION of decades of research that began with the development of tries for KWIC indices in the 1960s. The algorithms that we have discussed were worked out by many researchers over several decades in the context of solving practical problems ranging from putting the *Oxford English Dictionary* online to the development of the first web search engines to sequencing the human genome. This story certainly helps put the importance of algorithm design and analysis in context.

**Network-flow algorithms** Next, we consider a graph model that has been successful not just because it provides us with a simply stated problem-solving model that is useful in many practical applications but also because we have efficient algorithms for solving problems within the model. The solution that we consider illustrates the tension between our quest for implementations of general applicability and our quest for efficient solutions to specific problems. The study of network-flow algorithms is fascinating because it brings us tantalizingly close to compact and elegant implementations that achieve both goals. As you will see, we have straightforward implementations that are guaranteed to run in time proportional to a polynomial in the size of the network.

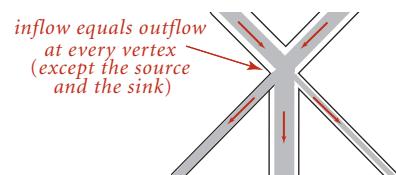
The classical solutions to network-flow problems are closely related to other graph algorithms that we studied in CHAPTER 4, and we can write surprisingly concise programs that solve them, using the algorithmic tools we have developed. As we have seen in many other situations, good algorithms and data structures can lead to substantial reductions in running times. Development of better implementations and better algorithms is still an area of active research, and new approaches continue to be discovered.

**A physical model.** We begin with an idealized physical model in which several of the basic concepts are intuitive. Specifically, imagine a collection of interconnected oil pipes of varying sizes, with switches controlling the direction of flow at junctions, as in the example illustrated at right. Suppose further that the network has a single *source* (say, an oil field) and a single *sink* (say, a large refinery) to which all the pipes ultimately connect. At each vertex, the flowing oil reaches an equilibrium where the amount of oil flowing in is equal to the amount flowing out. We measure both flow and pipe capacity in the same units (say, gallons per second). If every switch has the property that the total capacity of the ingoing pipes is equal to the total capacity of the outgoing pipes, then there is no problem to solve: we

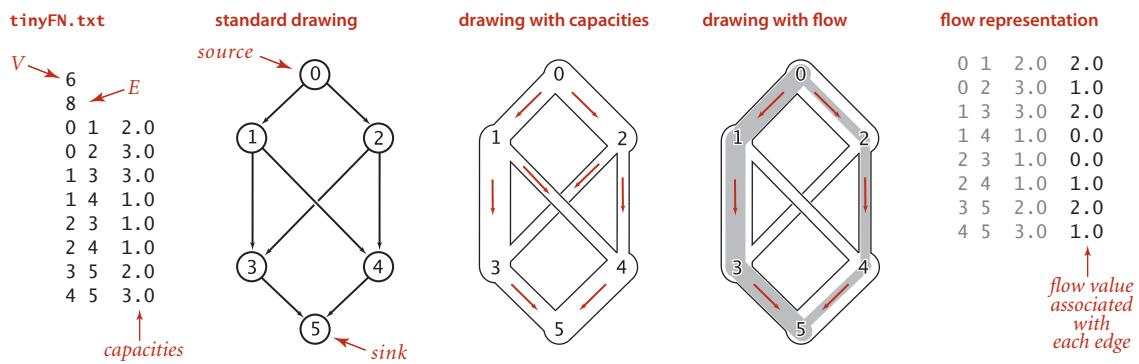


Adding flow to a network

simply fill all pipes to full capacity. Otherwise, not all pipes are full, but oil flows through the network, controlled by switch settings at the junctions, satisfying a *local equilibrium* condition at the junctions: the amount of oil flowing into each junction is equal to the amount of oil flowing out. For example, consider the network in the diagram on the opposite page. Operators might start the flow by opening the switches along the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$ , which can handle 2 units of flow, then open switches along the path  $0 \rightarrow 2 \rightarrow 4 \rightarrow 5$  to get another unit of flow in the network. Since  $0 \rightarrow 1$ ,  $2 \rightarrow 4$ , and  $3 \rightarrow 5$  are full, there is no direct way to get more flow from 0 to 5, but if we change the switch at 1 to redirect enough flow to fill  $1 \rightarrow 4$ , we open up enough capacity in  $3 \rightarrow 5$  to allow us to add a unit of flow on  $0 \rightarrow 2 \rightarrow 3 \rightarrow 5$ . Even for this simple network, finding switch settings that increase the flow is not an easy task; for a complicated network, we are clearly interested in the following question: What switch settings will maximize the amount of oil flowing from source to sink? We can model this situation directly with an edge-weighted digraph that has a single source and a single sink. The edges in the network correspond to the oil pipes, the vertices correspond to the junctions with switches that control how much oil goes into each outgoing edge, and the weights on the edges correspond to the capacity of the pipes. We assume that the edges are directed, specifying that oil can flow in only one direction in each pipe. Each pipe has a certain amount of flow, which is less than or equal to its capacity, and every vertex satisfies the equilibrium condition that the flow in is equal to the flow out. This flow-network abstraction is a useful problem-solving model that applies directly to a variety of applications and indirectly to still more. We sometimes appeal to the idea of oil flowing through pipes for intuitive support of basic ideas,



Local equilibrium in a flow network



Anatomy of a network-flow problem

but our discussion applies equally well to goods moving through distribution channels and to numerous other situations. As with our use of distance in shortest-paths algorithms, we are free to abandon any physical intuition when convenient because all the definitions, properties, and algorithms that we consider are based entirely on an abstract model that does not necessarily obey physical laws. Indeed, a prime reason for our interest in the network-flow model is that it allows us to solve numerous other problems through reduction, as we see in the next section.

**Definitions.** Because of this broad applicability, it is worthwhile to consider precise statements of the terms and concepts that we have just informally introduced:

**Definition.** A *flow network* is an edge-weighted digraph with positive edge weights (which we refer to as *capacities*). An *st-flow network* has two identified vertices, a source  $s$  and a sink  $t$ .

We sometimes refer to edges as having infinite capacity or, equivalently, as being uncapacitated. That might mean that we do not compare flow against capacity for such edges, or we might use a sentinel value that is guaranteed to be larger than any flow value. We refer to the total flow into a vertex (the sum of the flows on its incoming edges) as the vertex's *inflow*, the total flow out of a vertex (the sum of the flows on its outgoing edges) as the vertex's *outflow*, and the difference between the two (inflow minus outflow) as the vertex's *netflow*. To simplify the discussion, we also assume that there are no edges leaving  $t$  or entering  $s$ .

**Definition.** An *st-flow* in an *st-flow network* is a set of nonnegative values associated with each edge, which we refer to as *edge flows*. We say that a flow is *feasible* if it satisfies the condition that no edge's flow is greater than that edge's capacity and the local equilibrium condition that every vertex's netflow is zero (except  $s$  and  $t$ ).

We refer to the sink's inflow as the *st-flow value*. We will see in PROPOSITION E that the value is also equal to the source's outflow. With these definitions, the formal statement of our basic problem is straightforward:

**Maximum st-flow.** Given an *st-flow network*, find an *st-flow* such that no other flow from  $s$  to  $t$  has a larger value.

For brevity, we refer to such a flow as a *maxflow* and the problem of finding one in a network as the *maxflow problem*. In some applications, we might be content to know

just the maxflow value, but we generally want to know a flow (edge flow values) that achieves that value.

**APIs.** The `FlowEdge` and `FlowNetwork` APIs shown on page 890 are straightforward extensions of APIs from CHAPTER 4. We will consider on page 896 an implementation of `FlowEdge` that is based on adding an instance variable containing the flow to our `Edge` class from page 610. Flows have a direction, but we do not base `FlowEdge` on `DirectedEdge` because we work with a more general abstraction known as the *residual network* that is described below, and we need each edge to appear in the adjacency lists of both its vertices to implement the residual network. The residual network allows us to both add and subtract flow and to test whether an edge is full to capacity (no more flow can be added) or empty (no flow can be subtracted). This abstraction is implemented via the methods `residualCapacity()` and `addResidualFlow()` that we will consider later. The implementation of `FlowNetwork` is virtually identical to our `EdgeWeightedGraph` implementation on page 611, so we omit it. To simplify the file format, we adopt the convention that the source is 0 and the sink is  $V - 1$ . These APIs leave a straightforward goal for maxflow algorithms:

build a network, then assign values to the flow instance variables in the client's edges that maximize flow through the network. Shown at right are client methods for certifying whether a flow is feasible. Typically, we might do such a check as the final action of a maxflow algorithm.

```

private boolean localEq(FlowNetwork G, int v)
{ // Check local equilibrium at vertex v.
    double EPSILON = 1E-11;
    double netflow = 0.0;
    for (FlowEdge e : G.adj(v))
        if (v == e.from()) netflow -= e.flow();
        else                netflow += e.flow();
    return Math.abs(netflow) < EPSILON;
}

private boolean isFeasible(FlowNetwork G)
{
    // Check that flow on each edge is nonnegative
    // and not greater than capacity.
    for (int v = 0; v < G.V(); v++)
        for (FlowEdge e : G.adj(v))
            if (e.flow() < 0 || e.flow() > e.capacity())
                return false;

    // Check local equilibrium at each vertex.
    for (int v = 0; v < G.V(); v++)
        if (v != s && v != t && !localEq(v))
            return false;
    return true;
}

```

Checking that a flow is feasible in a flow network

```
public class FlowEdge
    FlowEdge(int v, int w, double cap)
        int from()
        int to()
        int other(int v)
        double capacity()
        double flow()
        double residualCapacityTo(int v)
        void addResidualFlowTo(int v, double delta)
        String toString()
```

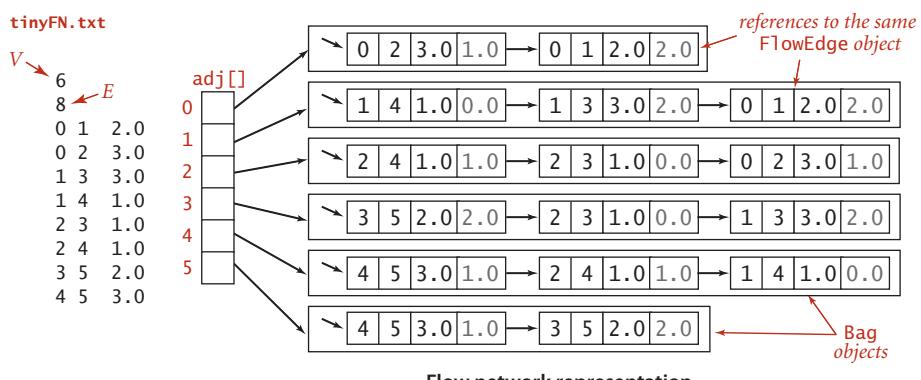
*vertex this edge points from*  
*vertex this edge points to*  
*other endpoint*  
*capacity of this edge*  
*flow in this edge*  
*residual capacity toward v*  
*add delta flow toward v*  
*string representation*

## API for edges in a flow network

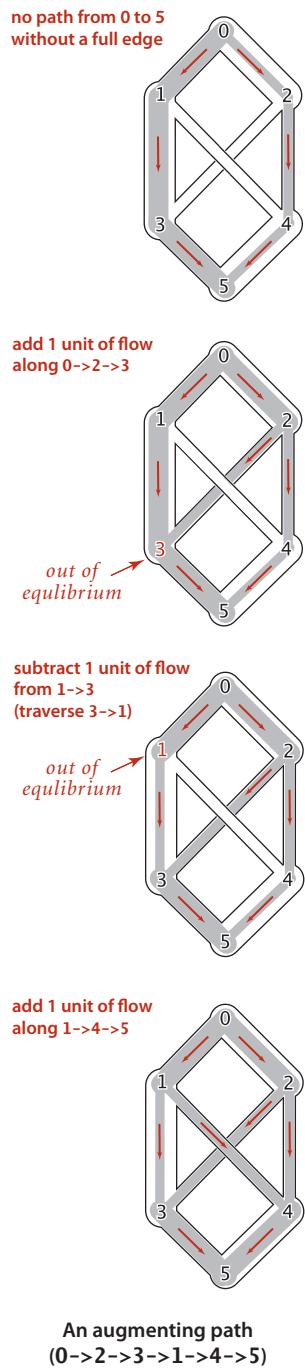
```
public class FlowNetwork
    FlowNetwork(int V)
    FlowNetwork(In in)
    int V()
    int E()
    void addEdge(FlowEdge e)
    Iterable<FlowEdge> adj(int v)
    Iterable<FlowEdge> edges()
    String toString()
```

*empty V-vertex flow network*  
*construct from input stream*  
*number of vertices*  
*number of edges*  
*add e to this flow network*  
*edges pointing from v*  
*all edges in this flow network*  
*string representation*

## Flow network API



**Ford-Fulkerson algorithm.** An effective approach to solving max-flow problems was developed by L. R. Ford and D. R. Fulkerson in 1962. It is a generic method for increasing flows incrementally along paths from source to sink that serves as the basis for a family of algorithms. It is known as the *Ford-Fulkerson algorithm* in the classical literature; the more descriptive term *augmenting-path algorithm* is also widely used. Consider any directed path from source to sink through an *st*-flow network. Let  $x$  be the minimum of the unused capacities of the edges on the path. We can increase the network's flow value by at least  $x$  by increasing the flow in all edges on the path by that amount. Iterating this action, we get a first attempt at computing flow in a network: find another path, increase the flow along that path, and continue until all paths from source to sink have at least one full edge (so that we can no longer increase flow in this way). This algorithm will compute the maxflow in some cases but will fall short in other cases. Our introductory example on page 886 is such an example. To improve the algorithm such that it always finds a maxflow, we consider a more general way to increase the flow, along a path from source to sink through the network's underlying *undirected* graph. The edges on any such path are either *forward* edges, which go with the flow (when we traverse the path from source to sink, we traverse the edge from its source vertex to its destination vertex), or *backward* edges, which go against the flow (when we traverse the path from source to sink, we traverse the edge from its destination vertex to its source vertex). Now, for any path from source to sink with no full forward edges and no empty backward edges, we can increase the amount of flow in the network by increasing flow in forward edges and decreasing flow in backward edges. The amount by which the flow can be increased is limited by the minimum of the unused capacities in the forward edges and the flows in the backward edges. Such a path is called an *augmenting path*. An example is shown at right. In the new flow, at least one of the forward edges along the path becomes full or at least one of the backward edges along the path becomes empty. The process just sketched is the basis for the classical Ford-Fulkerson maxflow algorithm (augmenting-path method). We summarize it as follows:



**Ford-Fulkerson maxflow algorithm.** Start with zero flow everywhere. Increase the flow along any augmenting path from source to sink (with no full forward edges or empty backward edges), continuing until there are no such paths in the network.

Remarkably (under certain technical conditions about numeric properties of the flow), this method always finds a maxflow, no matter how we choose the paths. Like the greedy MST algorithm discussed in SECTION 4.3 and the generic shortest-paths method discussed in SECTION 4.4, it is a generic algorithm that is useful because it establishes the correctness of a whole family of more specific algorithms. We are free to use any method whatsoever to choose the path. Several algorithms that compute sequences of augmenting paths have been developed, all of which lead to a maxflow. The algorithms differ in the number of augmenting paths they compute and the costs of finding each path, but they all implement the Ford-Fulkerson algorithm and find a maxflow.

**Maxflow-mincut theorem.** To show that any flow computed by any implementation of the Ford-Fulkerson algorithm is indeed a maxflow, we prove a key fact known as the *maxflow-mincut theorem*. Understanding this theorem is a crucial step in understanding network-flow algorithms. As suggested by its name, the theorem is based on a direct relationship between flows and cuts in networks, so we begin by defining terms that relate to cuts. Recall from SECTION 4.3 that a *cut* in a graph is a partition of the vertices into two disjoint sets, and a crossing edge is an edge that connects a vertex in one set to a vertex in the other set. For flow networks, we refine these definitions as follows:

**Definition.** An *st-cut* is a cut that places vertex  $s$  in one of its sets and vertex  $t$  in the other.

Each crossing edge corresponding to an *st-cut* is either an *st-edge* that goes from a vertex in the set containing  $s$  to a vertex in the set containing  $t$ , or a *ts-edge* that goes in the other direction. We sometimes refer to the set of crossing *st*-edges as a *cut set*. The *capacity* of an *st-cut* in a flow network is the sum of the capacities of that cut's *st*-edges, and the *flow across* an *st-cut* is the difference between the sum of the flows in that cut's *st*-edges and the sum of the flows in that cut's *ts*-edges. Removing all the *st*-edges (the cut set) in an *st-cut* of a network leaves no path from  $s$  to  $t$ , but adding any one of them back could create such a path. Cuts are the appropriate abstraction for many applications. For our oil-flow model, a cut provides a way to completely stop the flow of oil

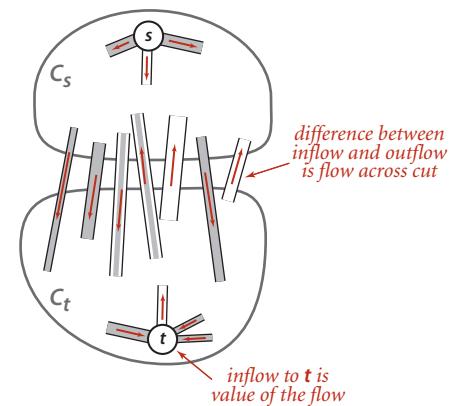
from the source to the sink. If we view the capacity of the cut as the cost of doing so, to stop the flow in the most economical manner is to solve the following problem:

**Minimum st-cut.** Given an *st*-network, find an *st*-cut such that the capacity of no other cut is smaller. For brevity, we refer to such a cut as a *mincut* and to the problem of finding one in a network as the *mincut problem*.

The statement of the mincut problem includes no mention of flows, and these definitions might seem to digress from our discussion of the augmenting-path algorithm. On the surface, computing a mincut (a set of edges) seems easier than computing a maxflow (an assignment of weights to all the edges). On the contrary, the maxflow and mincut problems are intimately related. The augmenting-path method itself provides a proof. That proof rests on the following basic relationship between flows and cuts, which immediately gives a proof that local equilibrium in an *st*-flow implies global equilibrium as well (the first corollary) and an upper bound on the value of any *st*-flow (the second corollary):

**Proposition E.** For any *st*-flow, the flow across each *st*-cut is equal to the value of the flow.

**Proof:** Let  $C_s$  be the vertex set containing  $s$  and  $C_t$  the vertex set containing  $t$ . This fact follows immediately by induction on the size of  $C_t$ . The property is true by definition when  $C_t$  is  $t$  and when a vertex is moved from  $C_s$  to  $C_t$ , local equilibrium at that vertex implies that the stated property is preserved. Any *st*-cut can be created by moving vertices in this way.



**Corollary.** The outflow from  $s$  is equal to the inflow to  $t$  (the value of the *st*-flow).

**Proof:** Let  $C_s$  be  $\{s\}$ .

**Corollary.** No *st*-flow's value can exceed the capacity of any *st*-cut.

**Proposition F. (Maxflow-mincut theorem)** Let  $f$  be an  $st$ -flow. The following three conditions are equivalent:

- i. There exists an  $st$ -cut whose capacity equals the value of the flow  $f$ .
- ii.  $f$  is a maxflow.
- iii. There is no augmenting path with respect to  $f$ .

**Proof:** Condition *i.* implies condition *ii.* by the corollary to PROPOSITION E. Condition *ii.* implies condition *iii.* because the existence of an augmenting path implies the existence of a flow with a larger flow value, contradicting the maximality of  $f$ .

It remains to prove that condition *iii.* implies condition *i.* Let  $C_s$  be the set of all vertices that can be reached from  $s$  with an undirected path that does not contain a full forward or empty backward edge, and let  $C_t$  be the remaining vertices. Then,  $t$  must be in  $C_t$ , so  $(C_s, C_t)$  is an  $st$ -cut, whose cut set consists entirely of full forward or empty backward edges. The flow across this cut is equal to the cut's capacity (since forward edges are full and the backward edges are empty) and also to the value of the flow (by PROPOSITION E).

**Corollary. (Integrality property)** When capacities are integers, there exists an integer-valued maxflow, and the Ford-Fulkerson algorithm finds it.

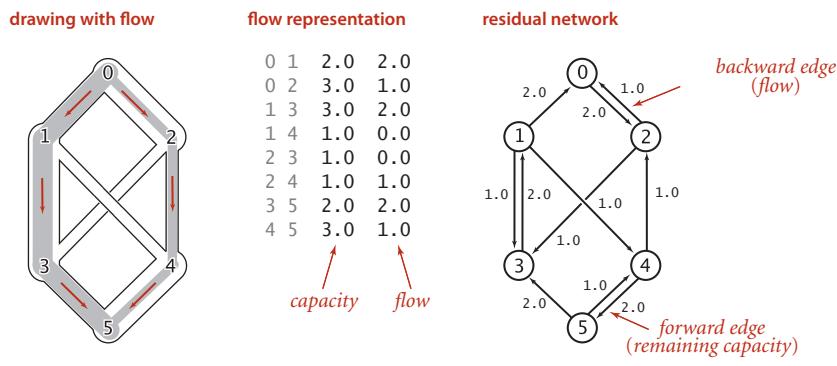
**Proof:** Each augmenting path increases the flow by a positive integer (the minimum of the unused capacities in the forward edges and the flows in the backward edges, all of which are always positive integers).

It is possible to design a maxflow with noninteger flows, even when capacities are all integers, but we do not need to consider such flows. From a theoretical standpoint, this observation is important: allowing capacities and flows that are real numbers, as we have done and as is common in practice, can lead to unpleasant anomalous situations. For example, it is known that the Ford-Fulkerson algorithm could, in principle, lead to an infinite sequence of augmenting paths that does not even converge to the maxflow value. The version of the algorithm that we consider is known to always converge, even when capacities and flows are real-valued. No matter what method we choose to find an augmenting path and no matter what paths we find, we always end up with a flow that does not admit an augmenting path, which therefore must be a maxflow.

**Residual network.** The generic Ford-Fulkerson algorithm does not specify any particular method for finding an augmenting path. How can we find a path with no full forward edges and no empty backward edges? To this end, we begin with the following definition:

**Definition.** Given a  $st$ -flow network and an  $st$ -flow, the *residual network* for the flow has the same vertices as the original and one or two edges in the residual network for each edge in the original, defined as follows: For each edge  $e$  from  $v$  to  $w$  in the original, let  $f_e$  be its flow and  $c_e$  its capacity. If  $f_e$  is positive, include an edge  $w \rightarrow v$  in the residual with capacity  $f_e$ ; and if  $f_e$  is less than  $c_e$ , include an edge  $v \rightarrow w$  in the residual with capacity  $c_e - f_e$ .

If an edge  $e$  from  $v$  to  $w$  is empty ( $f_e$  is equal to 0), there is a single corresponding edge  $v \rightarrow w$  with capacity  $c_e$  in the residual; if it is full ( $f_e$  is equal to  $c_e$ ), there is a single corresponding edge  $w \rightarrow v$  with capacity  $f_e$  in the residual; and if it is neither empty nor full, both  $v \rightarrow w$  and  $w \rightarrow v$  are in the residual with their respective capacities. An example is shown at the bottom of this page. At first, the residual network representation is a bit confusing because the edges corresponding to flow go in the *opposite* direction of the flow itself. The forward edges represent the remaining capacity (the amount of flow we can add if traversing that edge); the backward edges represent the flow (the amount of flow we can remove if traversing that edge). The code on page 896 gives the methods in the `FlowEdge` class that we need to implement the residual network abstraction. With these implementations, our algorithms work with the residual network, but they are actually examining capacities and changing flow (through edge references) in the client's edges. The methods `from()` and `other()` allow us to process edges in



## Flow edge data type (residual network)

```

public class FlowEdge
{
    private final int v;                                // edge source
    private final int w;                                // edge target
    private final double capacity;                      // capacity
    private double flow;                               // flow

    public FlowEdge(int v, int w, double capacity)
    {
        this.v = v;
        this.w = w;
        this.capacity = capacity;
        this.flow = 0.0;
    }

    public int from()         { return v; }
    public int to()          { return w; }
    public double capacity() { return capacity; }
    public double flow()     { return flow; }

    public int other(int vertex)
    // same as for Edge

    public double residualCapacityTo(int vertex)
    {
        if      (vertex == v) return flow;
        else if (vertex == w) return capacity - flow;
        else throw new RuntimeException("Inconsistent edge");
    }

    public void addResidualFlowTo(int vertex, double delta)
    {
        if      (vertex == v) flow -= delta;
        else if (vertex == w) flow += delta;
        else throw new RuntimeException("Inconsistent edge");
    }

    public String toString()
    { return String.format("%d->%d %.2f %.2f", v, w, capacity, flow); }
}

```

This `FlowEdge` implementation adds to the `Edge` implementation of SECTION 4.3 (see page 610) a `flow` instance variable and two methods to implement the residual flow network.

either orientation: `e.other(v)` returns the endpoint of `e` that is not `v`. The methods `residualCapacityTo()` and `addResidualFlowTo()` implement the residual network. Residual networks allow us to use graph search to find an augmenting path, since any path from source to sink in the residual network corresponds directly to an augmenting path in the original network. Increasing the flow along the path implies making changes in the residual network: for example, at least one edge on the path becomes full or empty, so at least one edge in the residual network changes direction or disappears (but our use of an abstract residual network means that we just check for positive capacity and do not need to actually insert and delete edges).

**Shortest-augmenting-path method.** Perhaps the simplest Ford-Fulkerson implementation is to use a *shortest* augmenting path (as measured by the number of edges on the path, not flow or capacity). This method was suggested by J. Edmonds and R. Karp in 1972. In this case, the search for an augmenting path amounts to breadth-first search (BFS) in the residual network, precisely as described in SECTION 4.1, as you can see by comparing the `hasAugmentingPath()` implementation below to our breadth-first search implementation in ALGORITHM 4.2 on page 540 (the residual graph is a digraph, and this is fundamentally a digraph processing algorithm, as mentioned on page 685). This method forms the basis for the full implementation in ALGORITHM 6.3 on the next page, a remarkably concise implementation based on the tools we have developed. For brevity, we refer to this method as the *shortest-augmenting-path* maxflow algorithm. A trace for our example is shown in detail on page 899.

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
{
    marked = new boolean[G.V()]; // Is path to this vertex known?
    edgeTo = new FlowEdge[G.V()]; // last edge on path
    Queue<Integer> q = new Queue<Integer>();
    marked[s] = true;           // Mark the source
    q.enqueue(s);              // and put it on the queue.
    while (!q.isEmpty())
    {
        int v = q.dequeue();
        for (FlowEdge e : G.adj(v))
        {
            int w = e.other(v);
            if (e.residualCapacityTo(w) > 0 && !marked[w])
                { // For every edge to an unmarked vertex (in residual)
                    edgeTo[w] = e; // Save the last edge on a path.
                    marked[w] = true; // Mark w because a path is known
                    q.enqueue(w); // and add it to the queue.
                }
        }
    }
    return marked[t];
}
```

Finding an augmenting path in the residual network via breadth-first search

---

**ALGORITHM 6.3 Ford-Fulkerson shortest-augmenting path maxflow algorithm**


---

```

public class FordFulkerson
{
    private boolean[] marked;      // Is s->v path in residual graph?
    private FlowEdge[] edgeTo;     // last edge on shortest s->v path
    private double value;          // current value of maxflow

    public FordFulkerson(FlowNetwork G, int s, int t)
    { // Find maxflow in flow network G from s to t.

        while (hasAugmentingPath(G, s, t))
        { // While there exists an augmenting path, use it.

            // Compute bottleneck capacity.
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v))
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));

            // Augment flow.
            for (int v = t; v != s; v = edgeTo[v].other(v))
                edgeTo[v].addResidualFlowTo(v, bottle);

            value += bottle;
        }
    }

    public double value() { return value; }
    public boolean inCut(int v) { return marked[v]; }

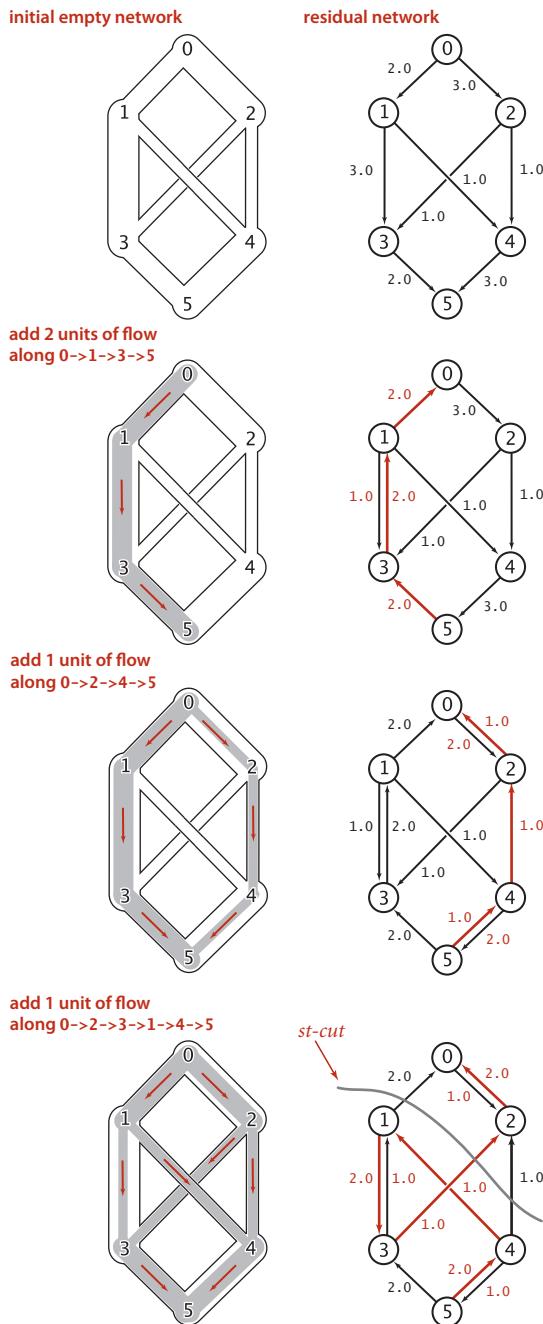
    public static void main(String[] args)
    {
        FlowNetwork G = new FlowNetwork(new In(args[0]));
        int s = 0, t = G.V() - 1;
        FordFulkerson maxflow = new FordFulkerson(G, s, t);

        StdOut.println("Max flow from " + s + " to " + t);
        for (int v = 0; v < G.V(); v++)
            for (FlowEdge e : G.adj(v))
                if ((v == e.from()) && e.flow() > 0)
                    StdOut.println("    " + e);
        StdOut.println("Max flow value = " + maxflow.value());
    }
}

```

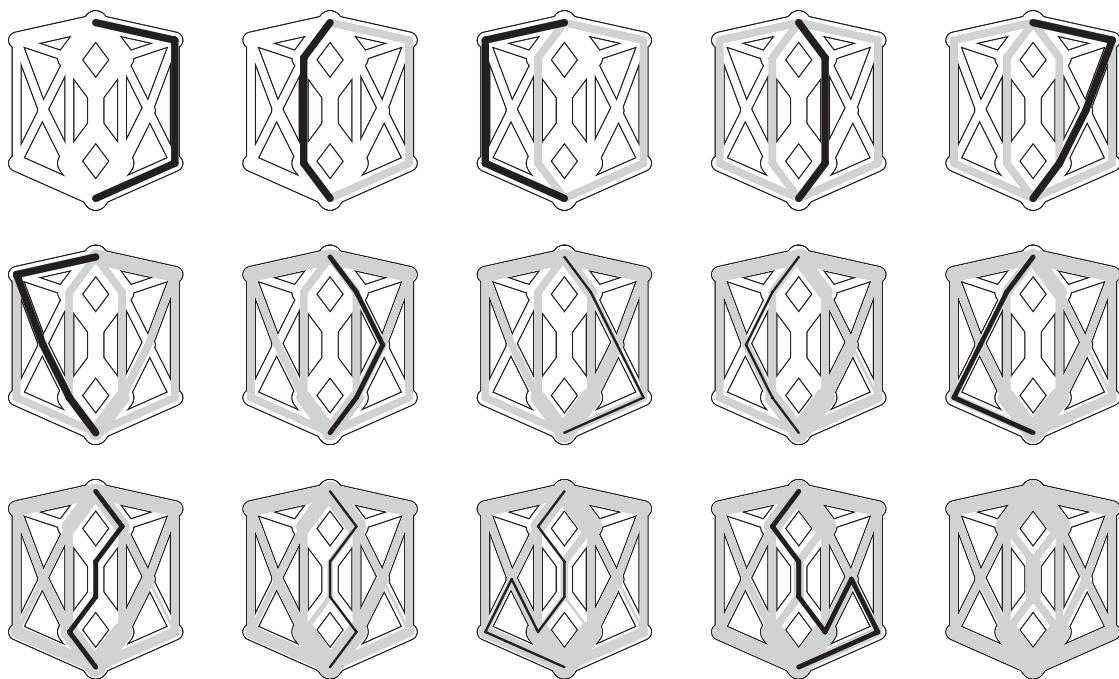
---

This implementation of the Ford-Fulkerson algorithm finds the shortest augmenting path in the residual network, finds the bottleneck capacity in that path, and augments the flow along that path, continuing until no path from source to sink exists.



Trace of augmenting-path Ford-Fulkerson algorithm

```
% java FordFulkerson tinyFN.txt
Max flow from 0 to 5
0->2 3.0 2.0
0->1 2.0 2.0
1->4 1.0 1.0
1->3 3.0 1.0
2->3 1.0 1.0
2->4 1.0 1.0
3->5 2.0 2.0
4->5 3.0 2.0
Max flow value = 4.0
```



Shortest augmenting paths in a larger flow network

**Performance.** A larger example is shown in the figure above. As is evident from the figure, the lengths of the augmenting paths form a nondecreasing sequence. This fact is a first key to analyzing the performance of the algorithm.

**Proposition G.** The number of augmenting paths needed in the shortest-augmenting-path implementation of the Ford-Fulkerson maxflow algorithm for a flow network with  $V$  vertices and  $E$  edges is at most  $EV/2$ .

**Proof sketch:** Every augmenting path has a *critical edge*—an edge that is deleted from the residual network because it corresponds either to a forward edge that becomes filled to capacity or a backward edge that is emptied. Each time an edge is a critical edge, the length of the augmenting path through it must increase by 2 (see EXERCISE 6.39). Since an augmenting path is of length at most  $V$  each edge can be on at most  $V/2$  augmenting paths, and the total number of augmenting paths is at most  $EV/2$ .

**Corollary.** The shortest-augmenting-path implementation of the Ford-Fulkerson maxflow algorithm takes time proportional to  $VE^2$  in the worst case.

**Proof:** Breadth-first search examines at most  $E$  edges.

The upper bound of PROPOSITION G is very conservative. For example, the graph shown in the figure at the top of page 900 has 14 vertices and 27 edges, so the bound says that the algorithm uses no more than 189 augmenting paths. In fact, it uses 14.

**Other implementations.** Another Ford-Fulkerson implementation, suggested by Edmonds and Karp, is the following: Augment along the path that increases the flow by the largest amount. For brevity, we refer to this method as the *maximum-capacity-augmenting-path* maxflow algorithm. We can implement this (and other approaches) by using a priority queue and slightly modifying our implementation of Dijkstra's shortest-paths algorithm, choosing edges from the priority queue to give the maximum amount of flow that can be pushed through a forward edge or diverted from a backward edge. Or, we might look for a longest augmenting path, or make a random choice. A complete analysis establishing which method is best is a complex task, because their running times depend on

- The number of augmenting paths needed to find a maxflow
- The time needed to find each augmenting path

These quantities can vary widely, depending on the network being processed and on the graph-search strategy. Several other approaches to solving the maxflow problem have also been devised, some of which compete well with the Ford-Fulkerson algorithm in practice. Developing a mathematical model of maxflow algorithms that can validate such hypotheses, however, is a significant challenge. The analysis of maxflow algorithms remains an interesting and active area of research. From a theoretical standpoint, worst-case performance bounds for numerous maxflow algorithms have been developed, but the bounds are generally substantially higher than the actual costs observed in applications and also quite a bit higher than the trivial (linear-time) lower bound. This gap between what is known and what is possible is larger than for any other problem that we have considered (so far) in this book.

**Computing a minimum st-cut.** Remarkably, the Ford-Fulkerson algorithm computes not only a maximum *st*-flow but also a minimum *st*-cut. The augmenting path algorithm terminates when there are no more augmenting paths with respect to the flow  $f$ . Upon termination, let  $C_s$  be the set of all vertices that can be reached from  $s$  with an undirected path that does not contain a full forward or empty backward edge, and let  $C_t$  be the remaining vertices. Then, as in the proof of PROPOSITION F,  $(C_s, C_t)$  is a minimum *st*-cut. ALGORITHM 6.3 provides an `inCut()` method that identifies the vertices on the  $s$ -side of the mincut. It accomplishes this by using the information left over in `marked[]` from the last call to `hasAugmentingPath()`.

THE PRACTICAL APPLICATION of maxflow algorithms remains both an art and a science. The art lies in picking the strategy that is most effective for a given practical situation; the science lies in understanding the essential nature of the problem. Are there new data structures and algorithms that can solve the maxflow problem in linear time, or can we prove that none exist?

algorithm	worst-case order of growth of running time for $V$ vertices and $E$ edges with integral capacities (max $C$ )
<i>Ford-Fulkerson</i> <i>shortest augmenting path</i>	$VE^2$
<i>Ford-Fulkerson</i> <i>maximum-capacity augmenting path</i>	$E^2 \log C$
<i>preflow-push</i>	$EV \log(E/V^2)$
<i>possible ?</i>	$V + E ?$
<b>Performance characteristics of maxflow algorithms</b>	

**Reduction** Throughout this book, we have focused on articulating specific problems, then developing algorithms and data structures to solve them. In several cases (many of which are listed below), we have found it convenient to solve a problem by formulating it as an instance of another problem that we have already solved. Formalizing this notion is a worthwhile starting point for studying relationships among the diverse problems and algorithms that we have studied.

**Definition.** We say that a problem  $A$  *reduces to* another problem  $B$  if we can use an algorithm that solves  $B$  to develop an algorithm that solves  $A$ .

This concept is certainly a familiar one in software development: when you use a library method to solve a problem, you are reducing your problem to the one solved by the library method. In this book, we have informally referred to problems that we can reduce to a given problem as *applications*.

**Sorting reductions.** We first encountered reduction in CHAPTER 2, to express the idea that an efficient sorting algorithm is useful for efficiently solving many other problems, that may not seem to be at all related to sorting. For example, we considered the following problems, among many others:

*Finding the median.* Given a set of numbers, find the median value.

*Distinct values.* Determine the number of distinct values in a set of numbers.

*Scheduling to minimize average completion time.* Given a set of jobs of specified duration to be completed, how can we schedule the jobs on a single processor so as to minimize their average completion time?

**Proposition H.** The following problems reduce to sorting:

- Finding the median
- Counting distinct values
- Scheduling to minimize average completion time

**Proof:** See page 345 and EXERCISE 2.5.12.

Now, we have to pay attention to cost when doing a reduction. For example, we can find the median of a set of numbers in linear time, but using the reduction to sorting will

end up costing linearithmic time. Even so, such extra cost might be acceptable, since we can use an existing sort implementation. Sorting is valuable for three reasons:

- It is useful in its own right.
- We have efficient algorithms for solving it.
- Many problems reduce to it.

Generally, we refer to a problem with these properties as a *problem-solving model*. Like well-engineered software libraries, well-designed problem-solving models can greatly expand the universe of problems that we can efficiently address. One pitfall in focusing on problem-solving models is known as *Maslow's hammer*, an idea widely attributed to A. Maslow in the 1960s: *If all you have is a hammer, everything seems to be a nail*. By focusing on a few problem-solving models, we may use them like Maslow's hammer to solve every problem that comes along, depriving ourselves of the opportunity to discover better algorithms to solve the problem, or even new problem-solving models. While the models we consider are important, powerful, and broadly useful, it is also wise to consider other possibilities.

**Shortest-paths reductions.** In SECTION 4.4, we revisited the idea of reduction in the context of shortest-paths algorithms. We considered the following problems, among many others:

**Single-source shortest paths in undirected graphs.** Given an edge-weighted *undirected* graph with nonnegative weights and a source vertex  $s$ , support queries of the form *Is there a path from  $s$  to a given target vertex  $v$ ?* If so, find a *shortest* such path (one whose total weight is minimal).

**Parallel precedence-constrained scheduling.** Given a set of jobs of specified duration to be completed, with precedence constraints that specify that certain jobs have to be completed before certain other jobs are begun, how can we schedule the jobs on identical processors (as many as needed) such that they are all completed in the minimum amount of time while still respecting the constraints?

**Arbitrage.** Find an arbitrage opportunity in a given table of currency-conversion rates.

Again, the latter two problems do not seem to be directly related to shortest-paths problems, but we saw that shortest paths is an effective way to address them. These examples, while important, are merely indicative. A large number of important problems, too many to survey here, are known to reduce to shortest paths—it is an effective and important problem-solving model.

**Proposition I.** The following problems reduce to shortest paths in weighted digraphs:

- Single-source shortest paths in undirected graphs with nonnegative weights
- Parallel precedence-constrained scheduling
- Arbitrage
- [many other problems]

**Proof examples:** See page 654, page 665, and page 680.

**Maxflow reductions.** Maxflow algorithms are also important in a broad context. We can remove various restrictions on the flow network and solve related flow problems; we can solve other network- and graph-processing problems; and we can solve problems that are not network problems at all. For example, consider the following problems.

**Job placement.** A college's job-placement office arranges interviews for a set of students with a set of companies; these interviews result in a set of job offers. Assuming that an interview followed by a job offer represents mutual interest in the student taking a job at the company, it is in everyone's best interests to maximize the number of job placements. Is it possible to match every student with a job? What is the maximum number of jobs that can be filled?

**Product distribution.** A company that manufactures a single product has factories, where the product is produced; distribution centers, where the product is stored temporarily; and retail outlets, where the product is sold. The company must distribute the product from factories through distribution centers to retail outlets on a regular basis, using distribution channels that have varying capacities. Is it possible to get the product from the warehouses to the retail outlets such that supply meets demand everywhere?

**Network reliability.** A simplified model considers a computer network as consisting of a set of trunk lines that connect computers through switches such that there is the possibility of a switched path through trunk lines connecting any two given computers. What is the minimum number of trunk lines that can be cut to disconnect some pair of computers?

Again, these problems seem to be unrelated to one another and to flow networks, but they all reduce to maxflow.

**Proposition J.** The following problems reduce to the maxflow problem:

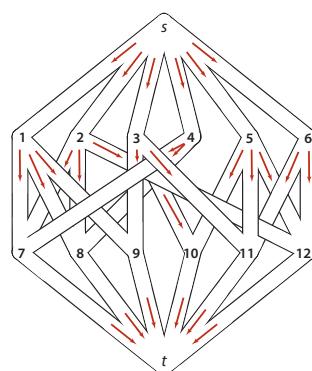
- Job placement
- Product distribution
- Network reliability
- [many other problems]

**Proof example:** We prove the first (which is known as the *maximum bipartite matching problem*) and leave the others for exercises. Given a job-placement problem, construct an instance of the maxflow problem by directing all edges from students to companies, adding a source vertex with edges directed to all the students and adding a sink vertex with edges directed from all the companies. Assign each edge capacity 1. Now, any integral solution to the maxflow problem for this network provides a solution to the corresponding bipartite matching problem (see the corollary to PROPOSITION F). The matching corresponds exactly to those edges between vertices in the two sets that are filled to capacity by the maxflow algorithm. First, the network flow always gives a legal matching: since each vertex has an edge of capacity 1 either coming in (from the source) or going out (to the sink), at most 1 unit of flow can go through each vertex, implying in turn that each vertex will be included at most once in the matching. Second, no matching can have more edges, since any such matching would lead directly to a better flow than that produced by the maxflow algorithm.

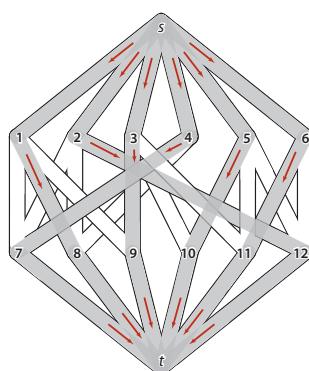
#### bipartite matching problem

1 Alice	7 Adobe
Adobe	Alice
Amazon	Bob
Facebook	Dave
2 Bob	8 Amazon
Adobe	Alice
Amazon	Bob
Yahoo	Dave
3 Carol	9 Facebook
Facebook	Alice
Google	Carol
IBM	10 Google
4 Dave	Carol
Adobe	Eliza
Amazon	11 IBM
5 Eliza	Carol
Google	Eliza
IBM	Frank
Yahoo	12 Yahoo
6 Frank	Bob
IBM	Eliza
Yahoo	Frank

#### network-flow formulation



#### maximum flow



#### matching (solution)

Alice	— Amazon
Bob	— Yahoo
Carol	— Facebook
Dave	— Adobe
Eliza	— Google
Frank	— IBM

Example of reducing maximum bipartite matching to network flow

For example, as illustrated in the figure at right, an augmenting-path max-flow algorithm might use the paths  $s \rightarrow 1 \rightarrow 7 \rightarrow t$ ,  $s \rightarrow 2 \rightarrow 8 \rightarrow t$ ,  $s \rightarrow 3 \rightarrow 9 \rightarrow t$ ,  $s \rightarrow 5 \rightarrow 10 \rightarrow t$ ,  $s \rightarrow 6 \rightarrow 11 \rightarrow t$ , and  $s \rightarrow 4 \rightarrow 7 \rightarrow 1 \rightarrow 8 \rightarrow 2 \rightarrow 12 \rightarrow t$  to compute the matching 1-8, 2-12, 3-9, 4-7, 5-10, and 6-11. Thus, there is a way to match all the students to jobs in our example. Each augmenting path fills one edge from the source and one edge into the sink. Note that these edges are never used as back edges, so there are at most  $V$  augmenting paths and a total running time proportional to  $VE$ .

SHORTEST PATHS AND MAXFLOW ARE IMPORTANT problem-solving models because they have the same properties that we articulated for sorting:

- They are useful in their own right.
- We have efficient algorithms for solving them.
- Many problems reduce to them.

This short discussion serves only to introduce the idea. If you take a course in operations research, you will learn many other problems that reduce to these and many other problem-solving models.

**Linear programming.** One of the cornerstones of operations research is *linear programming* (LP). It refers to the idea of reducing a given problem to the following mathematical formulation:

**Linear programming.** Given a set of  $M$  linear inequalities and linear equations involving  $N$  variables, and a linear objective function of the  $N$  variables, find an assignment of values to the variables that maximizes the objective function, or report that no feasible assignment exists.

Maximize  $f + h$   
subject to the constraints

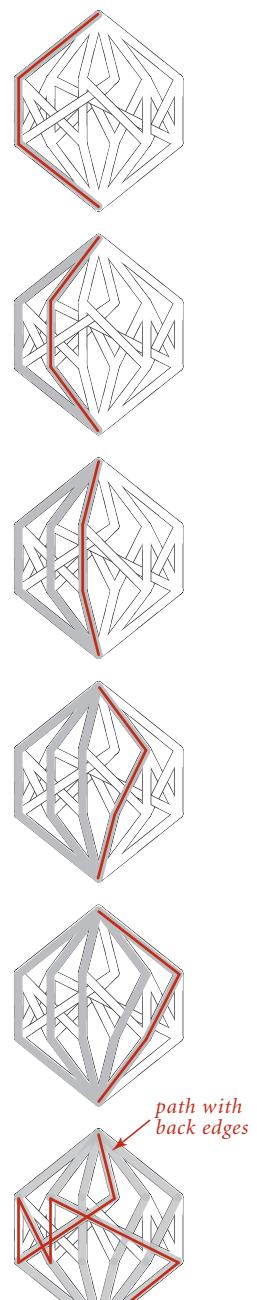
$$\begin{aligned} 0 &\leq a \leq 2 \\ 0 &\leq b \leq 3 \\ 0 &\leq c \leq 3 \\ 0 &\leq d \leq 1 \\ 0 &\leq e \leq 1 \\ 0 &\leq f \leq 1 \\ 0 &\leq g \leq 2 \\ 0 &\leq h \leq 3 \\ a &= c+d \\ b &= e+f \\ c+e &= g \\ d+f &= h \end{aligned}$$

LP example

Linear programming is an extremely important problem-solving model because

- A great many important problems reduce to linear programming
- We have efficient algorithms for solving linear-programming problems

The “useful in its own right” phrase is not needed in this litany that we have stated for other problem-solving models because *so many* practical problems reduce to linear programming.



Augmenting paths for bipartite matching

**Proposition K.** The following problems reduce to linear programming

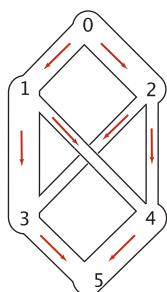
- Maxflow
- Shortest paths
- [many, many other problems]

**Proof example:** We prove the first and leave the second to EXERCISE 6.50. We consider a system of inequalities and equations that involve one variable corresponding to each edge, two inequalities corresponding to each edge, and one equation corresponding to each vertex (except the source and the sink). The value of the variable is the edge flow, the inequalities specify that the edge flow must be between 0 and the edge's capacity, and the equations specify that the total flow on the edges that go into each vertex must be equal to the total flow on the edges that go out of that vertex. Any max flow problem can be converted into an instance of a linear programming problem in this way, and the solution is easily converted to a solution of the maxflow problem. The illustration below gives the details for our example.

maxflow problem

V	6	E
0	1	2.0
0	2	3.0
1	3	3.0
1	4	1.0
2	3	1.0
2	4	1.0
3	5	2.0
4	5	3.0

↑  
capacities



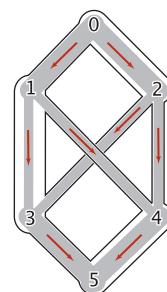
LP formulation

$$\begin{aligned}
 & \text{Maximize } x_{35} + x_{45} \\
 & \text{subject to the constraints} \\
 & 0 \leq x_{01} \leq 2 \\
 & 0 \leq x_{02} \leq 3 \\
 & 0 \leq x_{13} \leq 3 \\
 & 0 \leq x_{14} \leq 1 \\
 & 0 \leq x_{23} \leq 1 \\
 & 0 \leq x_{24} \leq 1 \\
 & 0 \leq x_{35} \leq 2 \\
 & 0 \leq x_{45} \leq 3 \\
 & x_{01} = x_{13} + x_{14} \\
 & x_{02} = x_{23} + x_{24} \\
 & x_{13} + x_{23} = x_{35} \\
 & x_{14} + x_{24} = x_{45}
 \end{aligned}$$

maxflow solution

Max flow from 0 to 5
0->2 3.0 2.0
0->1 2.0 2.0
1->4 1.0 1.0
1->3 3.0 1.0
2->3 1.0 1.0
2->4 1.0 1.0
3->5 2.0 2.0
4->5 3.0 2.0

Max flow value: 4.0



Example of reducing network flow to linear programming

The “many, many other problems” in the statement of PROPOSITION K refers to three ideas. First, *it is very easy to extend a model and to add constraints*. Second, *reduction is transitive*, so all the problems that reduce to shortest paths and maximum flow also reduce to linear programming. Third, and more generally, *optimization problems of all sorts can be directly formulated as linear programming problems*. Indeed, the term *linear programming* means “formulate an optimization problem as a linear programming problem.” This use predates the use of the word *programming* for computers. Equally important as the idea that a great many problems reduce to linear programming is the fact that efficient algorithms have been known for linear programming for many decades. The most famous, developed by G. Dantzig in the 1940s, is known as the *simplex algorithm*. Simplex is not difficult to understand (see the bare-bones implementation on the booksite). More recently, the *ellipsoid algorithm* presented by L. G. Khachian in 1979 led to the development of *interior point methods* in the 1980s that have proven to be an effective complement to the simplex algorithm for the huge linear programming problems that people are solving in modern applications. Nowadays, linear programming solvers are robust, extensively tested, efficient, and critical to the basic operation of modern corporations. Uses in scientific contexts and even in applications programming are also greatly expanding. If you can model your problem as a linear programming problem, you are likely to be able to solve it.

IN A VERY REAL SENSE, LINEAR PROGRAMMING IS THE PARENT of problem-solving models, since so many problems reduce to it. Naturally, this idea leads to the question of whether there is an even more powerful problem-solving model than linear programming. What sorts of problems do *not* reduce to linear programming? Here is an example of such a problem:

***Load balancing.*** Given a set of jobs of specified duration to be completed, how can we schedule the jobs on two identical processors so as to minimize the completion time of all the jobs?

Can we articulate a more general problem-solving model and solve instances of problems within that model efficiently? This line of thinking leads to the idea of *intractability*, our last topic.

**Intractability** The algorithms that we have studied in this book generally are used to solve practical problems and therefore consume reasonable amounts of resources. The practical utility of most of the algorithms is obvious, and for many problems we have the luxury of several efficient algorithms to choose from. Unfortunately, many other problems arise in practice that do not admit such efficient solutions. What's worse, for a large class of such problems we cannot even tell whether or not an efficient solution exists. This state of affairs has been a source of extreme frustration for programmers and algorithm designers, who cannot find any efficient algorithm for a wide range of practical problems, and for theoreticians, who have been unable to find any proof that these problems are difficult. A great deal of research has been done in this area and has led to the development of mechanisms by which new problems can be classified as being “hard to solve” in a particular technical sense. Though much of this work is beyond the scope of this book, the central ideas are not difficult to learn. We introduce them here because every programmer, when faced with a new problem, should have some understanding of the possibility that there exist problems for which no one knows any algorithm that is guaranteed to be efficient.

**Groundwork.** One of the most beautiful and intriguing intellectual discoveries of the 20th century, developed by A. Turing in the 1930s, is the *Turing machine*, a simple model of computation that is general enough to embody any computer program or computing device. A Turing machine is a finite-state machine that can read inputs, move from state to state, and write outputs. Turing machines form the foundation of theoretical computer science, starting with the following two ideas:

- *Universality*. All physically realizable computing devices can be simulated by a Turing machine. This idea is known as the *Church-Turing thesis*. This is a statement about the natural world and cannot be proven (but it can be falsified). The evidence in favor of the thesis is that mathematicians and computer scientists have developed numerous models of computation, but they all have been proven equivalent to the Turing machine.
- *Computability*. There exist problems that cannot be solved by a Turing machine (or by any other computing device, by universality). This is a mathematical truth. The halting problem (no program can guarantee to determine whether a given program will halt) is a famous example of such a problem.

In the present context, we are interested in a third idea, which speaks to the efficiency of computing devices:

- *Extended Church-Turing thesis*. The order of growth of the running time of a program to solve a problem on any computing device is within a polynomial factor of some program to solve the problem on a Turing machine (or any other computing device).

Again, this is a statement about the natural world, buttressed by the idea that all known computing devices can be simulated by a Turing machine, with at most a polynomial factor increase in cost. In recent years, the idea of *quantum computing* has given some researchers reason to doubt the extended Church-Turing thesis. Most agree that, from a practical point of view, it is probably safe for some time, but many researchers are hard at work on trying to falsify the thesis.

**Exponential running time.** The purpose of the theory of intractability is to separate problems that can be solved in polynomial time from problems that (probably) require *exponential* time to solve in the worst case. It is useful to think of an exponential-time algorithm as one that, for some input of size  $N$ , takes time proportional to  $2^N$  (at least). The substance of the argument does not change if we replace 2 by any number  $\alpha > 1$ . We generally take as granted that an exponential-time algorithm cannot be guaranteed to solve a problem of size 100 (say) in a reasonable amount of time, because no one can wait for an algorithm to take  $2^{100}$  steps, regardless of the speed of the computer. Exponential growth dwarfs technological changes: a supercomputer may be a trillion times faster than an abacus, but neither can come close to solving a problem that requires  $2^{100}$  steps. Sometimes the line between “easy” and “hard” problems is a fine one. For example, we studied an algorithm in SECTION 4.1 that can solve the following problem:

**Shortest-path length.** What is the length of the shortest path from a given vertex  $s$  to a given vertex  $t$  in a given graph?

But we did not study algorithms for the following problem, which seems to be virtually the same:

**Longest-path length.** What is the length of the longest simple path from a given vertex  $s$  to a given vertex  $t$  in a given graph?

```
public class LongestPath
{
    private boolean[] marked;
    private int max;

    public LongestPath(Graph G, int s, int t)
    {
        marked = new boolean[G.V()];
        dfs(G, s, t, 0);
    }

    private void dfs(Graph G, int v, int t, int i)
    {
        if (v == t && i > max) max = i;
        if (v == t) return;
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w, t, i+1);
        marked[v] = false;
    }

    public int maxLength()
    {
        return max;
    }
}
```

Finding the length of the longest path in a graph

The crux of the matter is this: as far as we know, these problems are nearly at opposite ends of the spectrum with respect to difficulty. Breadth-first search yields a solution for the first problem in *linear* time, but all known algorithms for the second problem take *exponential* time in the worst case. The code at the bottom of the previous page shows a variant of depth-first search that accomplishes the task. It is quite similar to depth-first search, but it examines *all* simple paths between  $s$  and  $t$  in the graph to find the longest one.

**Search problems.** The great disparity between problems that can be solved with “efficient” algorithms of the type we have been studying in this book and problems where we need to look for a solution among a potentially huge number of possibilities makes it possible to study the interface between them with a simple formal model. The first step is to characterize the type of problem that we study:

**Definition.** A *search problem* is a problem having solutions with the property that the time needed to *certify* that any solution is correct is bounded by a polynomial in the size of the input. We say that an algorithm *solves* a search problem if, given any input, it either produces a solution or reports that none exists.

Four particular problems that are of interest in our discussion of intractability are shown at the top of the facing page. These problems are known as *satisfiability* problems. Now, all that is required to establish that a problem is a search problem is to show that any solution is sufficiently well-characterized that you can efficiently certify that it is correct. Solving a search problem is like searching for a “needle in a haystack” with the sole proviso that you can recognize the needle when you see it. For example, if you are given an assignment of values to variables in each of the satisfiability problems at the top of page 913, you easily can certify that each equality or inequality is satisfied, but searching for such an assignment is a totally different task. The name **NP** is commonly used to describe search problems—we will describe the reason for the name on page 914:

**Definition.** **NP** is the set of all search problems.

**NP** is nothing more than a precise characterization of all the problems that scientists, engineers, and applications programmers *aspire to solve* with programs that are guaranteed to finish in a feasible amount of time.

**Linear equation satisfiability.** Given a set of  $M$  linear equations involving  $N$  variables, find an assignment of values to the variables that satisfies all of the equations, or report that none exists.

**Linear inequality satisfiability (search formulation of linear programming).** Given a set of  $M$  linear inequalities involving  $N$  variables, find an assignment of values to the variables that satisfies all of the inequalities, or report that none exists.

**0-1 integer linear inequality satisfiability (search formulation of 0-1 integer linear programming).** Given a set of  $M$  linear inequalities involving  $N$  integer variables, find an assignment of the values 0 or 1 to the variables that satisfies all of the inequalities, or report that none exists.

**Boolean satisfiability.** Given a set of  $M$  equations involving *and* and *or* operations on  $N$  boolean variables, find an assignment of values to the variables that satisfies all of the equations, or report that none exists.

#### Selected search problems

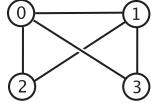
**Other types of problems.** The concept of search problems is one of many ways to characterize the set of problems that form the basis of the study of intractability. Other possibilities are *decision* problems (does a solution exist?) and *optimization* problems (what is the best solution?). For example, the longest-paths length problem on page 911 is an optimization problem, not a search problem (given a solution, we have no way to verify that it is a longest-path length). A search version of this problem is to *find* a simple path connecting all the vertices (this problem is known as the *Hamiltonian path problem*). A decision version of the problem is to ask whether *there exists* a simple path connecting all the vertices. Arbitrage, boolean satisfiability, and Hamiltonian path are search problems; to ask whether a solution exists to any of these problems is a decision problem; and shortest/longest paths, maxflow, and linear programming are all optimization problems. While not technically equivalent, search, decision, and optimization problems typically reduce to one another (see EXERCISE 6.58 and 6.59) and the main conclusions we draw apply to all three types of problems.

**Easy search problems.** The definition of **NP** says nothing about the difficulty of *finding* the solution, just certifying that it is a solution. The second of the two sets of problems that form the basis of the study of intractability, which is known as **P**, is concerned with the difficulty of finding the solution. In this model, the efficiency of an algorithm is a function of the number of bits used to encode the input.

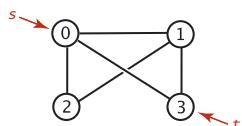
**Definition.** **P** is the set of all search problems that can be solved in polynomial time.

Implicit in the definition is the idea that the polynomial time bound is a *worst-case* bound. For a problem to be in **P**, there must exist an algorithm that can *guarantee* to solve it in polynomial time. Note that the polynomial is not specified at all. Linear, linearithmic, quadratic, and cubic are all polynomial time bounds, so this definition certainly covers the standard algorithms we have studied so far. The time taken by an algorithm depends on the computer used, but the extended Church-Turing thesis renders that point moot—it says that a polynomial-time solution on any computing device implies the existence of a polynomial-time solution on any other computing device. Sorting belongs to **P** because (for example) insertion sort runs in time proportional to  $N^2$  (the existence of linearithmic sorting algorithms is not relevant in this context), as does shortest paths, linear equation satisfiability, and many others. Having an efficient algorithm to solve a problem is a proof that the problem is in **P**. In other words, **P** is nothing more than a precise characterization of all the problems that scientists, engineers, and applications programmers *do solve* with programs that are guaranteed to finish in a feasible amount of time.

**Nondeterminism.** The **N** in **NP** stands for *nondeterminism*. It represents the idea that one way (in theory) to extend the power of a computer is to endow it with the power of nondeterminism: to assert that when an algorithm is faced with a choice of several options, it has the power to “guess” the right one. For the purposes of our discussion, we can think of an algorithm for a nondeterministic machine as “guessing” the solution to a problem, then certifying that the solution is valid. In a Turing machine, nondeterminism is as simple as defining two different successor states for a given state and a given input and characterizing solutions as all legal paths to the desired result. Nondeterminism may be a mathematical fiction, but it is a useful idea. For example, in SECTION 5.4, we used nondeterminism as a tool for algorithm design—our regular expression pattern-matching algorithm is based on efficiently simulating a nondeterministic machine.

problem	input	description	poly-time algorithm	instance	solution
<i>Hamiltonian path</i>	graph $G$	find a simple path that visits every vertex	?		0-2-1-3
<i>factoring</i>	integer $x$	find a nontrivial factor of $x$	?	97605257271	8784561
<i>0-1 linear inequality satisfiability</i>	$N$ 0-1 variables $M$ inequalities	assign values to the variables that satisfy the inequalities	?	$\begin{aligned}x - y &\leq 1 \\ 2x - z &\leq 2 \\ x + y &\geq 2 \\ z &\geq 0\end{aligned}$	$\begin{aligned}x &= 1 \\ y &= 1 \\ z &= 0\end{aligned}$
<i>all problems in P</i>		see table below			

### Examples of problems in NP

problem	input	description	poly-time algorithm	instance	solution
<i>shortest st-path</i>	graph $G$ vertices $s, t$	find the shortest path from $s$ to $t$	BFS		0-3
<i>sorting</i>	array $a$	find a permutation that puts $a$ in ascending order	mergesort	2.8 8.5 4.1 1.3	3 0 2 1
<i>linear equation satisfiability</i>	$N$ variables $M$ equations	assign values to the variables that satisfy the equations	Gaussian elimination	$\begin{aligned}x + y &= 1.5 \\ 2x - y &= 0\end{aligned}$	$\begin{aligned}x &= 0.5 \\ y &= 1\end{aligned}$
<i>linear inequality satisfiability</i>	$N$ variables $M$ inequalities	assign values to the variables that satisfy the inequalities	ellipsoid	$\begin{aligned}x - y &\leq 1.5 \\ 2x - z &\leq 0 \\ x + y &\geq 3.5 \\ z &\geq 4.0\end{aligned}$	$\begin{aligned}x &= 2.0 \\ y &= 1.5 \\ z &= 4.0\end{aligned}$

### Examples of problems in P

**The main question.** Nondeterminism is such a powerful notion that it seems almost absurd to consider it seriously. Why bother considering an imaginary tool that makes difficult problems seem trivial? The answer is that, powerful as nondeterminism may seem, no one has been able to *prove* that it helps for any particular problem! Put another way, no one has been able to find a single problem that can be proven to be in **NP** but not in **P** (or even prove that one exists), leaving the following question open:

*Does **P** = **NP**?*

This question was first posed in a famous letter from K. Gödel to J. von Neumann in 1950 and has completely stumped mathematicians and computer scientists ever since. Other ways of posing the question shed light on its fundamental nature:

- Are there *any* hard-to-solve search problems?
- Would we be able to solve some search problems more efficiently if we could build a nondeterministic computing device?

Not knowing the answers to these questions is extremely frustrating because many important practical problems belong to **NP** but may or may not belong to **P** (the best known deterministic algorithms could take exponential time). If we could prove that a problem does not belong to **P**, then we could abandon the search for an efficient solution to it. In the absence of such a proof, there is the possibility that some efficient algorithm has gone undiscovered. In fact, given the current state of our knowledge, there could be some efficient algorithm for *every* problem in **NP**, which would imply that many efficient algorithms have gone undiscovered. Virtually no one believes that **P** = **NP**, and a considerable amount of effort has gone into proving the contrary, but this remains the outstanding open research problem in computer science.

**Poly-time reductions.** Recall from page 903 that we show that a problem *A* *reduces to* another problem *B* by demonstrating that we can solve any instance of *A* in three steps:

- Transform it to an instance of *B*.
- Solve that instance of *B*.
- Transform the solution of *B* to be a solution of *A*.

As long as we can perform the transformations (and solve *B*) efficiently, we can solve *A* efficiently. In the present context, for *efficient* we use the weakest conceivable definition: to solve *A* we solve at most a polynomial number of instances of *B*, using transformations that require at most polynomial time. In this case, we say that *A* *poly-time reduces* to *B*. Before, we used reduction to introduce the idea of problem-solving models that can significantly expand the range of problems that we can solve with efficient algorithms. Now, we use reduction in another sense: *to prove a problem to be hard to solve*. If a problem *A* is known to be hard to solve, and *A* poly-time reduces to *B*, then *B* must be hard to solve, too. Otherwise, a guaranteed polynomial-time solution to *B* would give a guaranteed polynomial-time solution to *A*.

**Proposition L.** Boolean satisfiability poly-time reduces to 0-1 integer linear inequality satisfiability.

**Proof:** Given an instance of boolean satisfiability, define a set of inequalities with one 0-1 variable corresponding to each boolean variable and one 0-1 variable corresponding to each clause, as illustrated in the example at right. With this construction, we can transform a solution to the integer 0-1 linear inequality satisfiability problem to a solution to the boolean satisfiability problem by assigning each boolean variable to be *true* if the corresponding integer variable is 1 and *false* if it is 0.

**Corollary.** If satisfiability is hard to solve, then so is integer linear programming.

This statement is a meaningful statement about the relative difficulty of solving these two problems even in the absence of a precise definition of *hard to solve*. In the present context, by “hard to solve,” we mean “not in **P**.” We generally use the word *intractable* to refer to problems that are not in **P**. Starting with the seminal work of R. Karp in 1972, researchers have shown literally tens of thousands of problems from a wide variety of applications areas to be related by reduction relationships of this sort. Moreover, these relationships imply much more than just relationships between the individual problems, a concept that we now address.

**NP-completeness.** Many, many problems are known to belong to **NP** but probably do not belong to **P**. That is, we can easily *certify* that any given solution is valid, but, despite considerable effort, no one has been able to develop an efficient algorithm to *find* a solution. Remarkably, all of these many, many problems have an additional property that provides convincing evidence that **P** ≠ **NP**:

#### boolean satisfiability problem

$$(x'_1 \text{ or } x_2 \text{ or } x_3) \text{ and} \\ (x_1 \text{ or } x'_2 \text{ or } x_3) \text{ and} \\ (x'_1 \text{ or } x'_2 \text{ or } x'_3) \text{ and} \\ (x'_1 \text{ or } x'_2 \text{ or } x_3)$$

#### 0-1 integer linear inequality satisfiability formulation

*c<sub>1</sub> is 1  
if and only if  
first clause is  
satisfiable*

$$\begin{aligned} c_1 &\geq 1 - x_1 \\ c_1 &\geq x_2 \\ c_1 &\geq x_3 \\ c_1 &\leq (1 - x_1) + x_2 + x_3 \end{aligned}$$

$$\begin{aligned} c_2 &\geq x_1 \\ c_2 &\geq 1 - x_2 \\ c_2 &\geq x_3 \\ c_2 &\leq x_1 + (1 - x_2) + x_3 \end{aligned}$$

$$\begin{aligned} c_3 &\geq 1 - x_1 \\ c_3 &\geq 1 - x_2 \\ c_3 &\geq 1 - x_3 \\ c_3 &\leq (1 - x_1) + 1 - x_2 + (1 - x_3) \end{aligned}$$

$$\begin{aligned} c_4 &\geq 1 - x_1 \\ c_4 &\geq 1 - x_2 \\ c_4 &\geq x_3 \\ c_4 &\leq (1 - x_1) + (1 - x_2) + x_3 \end{aligned}$$

*s ≤ c<sub>1</sub>  
s ≤ c<sub>2</sub>  
s ≤ c<sub>3</sub>  
s ≤ c<sub>4</sub>  
s is 1  
if and only if  
c's are all 1*

$$s \geq c_1 + c_2 + c_3 + c_4 - 3$$

#### Example of reducing boolean satisfiability to 0-1 integer linear inequality satisfiability

**Definition.** A search problem  $A$  is said to be  **$NP$ -complete** if all problems in  $NP$  poly-time reduce to  $A$ .

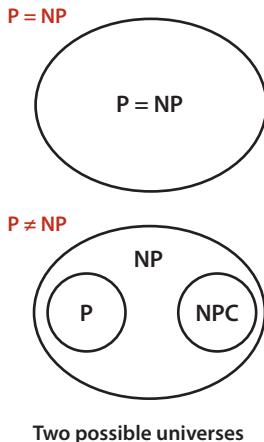
This definition enables us to upgrade our definition of “hard to solve” to mean “intractable unless  $P = NP$ .” If *any*  $NP$ -complete problem can be solved in polynomial time on a deterministic machine, then so can *all problems in NP* (i.e.,  $P = NP$ ). That is, the collective failure of all researchers to find efficient algorithms for all of these problems might be viewed as a collective failure to prove that  $P = NP$ . Since most computer scientists believe that  $P \neq NP$ , we cannot expect to find guaranteed polynomial-time algorithms for  $NP$ -complete problems.

**Cook-Levin theorem.** Reduction uses the  $NP$ -completeness of one problem to imply the  $NP$ -completeness of another. But reduction cannot be used in one case: how was the *first* problem proven to be  $NP$ -complete? This was done independently by S. Cook and L. Levin in the early 1970s.

**Proposition M. (Cook-Levin theorem)** Boolean satisfiability is  $NP$ -complete.

**Extremely brief proof sketch:** The goal is to show that if there is a polynomial time algorithm for boolean satisfiability, then all problems in  $NP$  can be solved in polynomial time. Now, a nondeterministic Turing machine can solve any problem in  $NP$ , so the first step in the proof is to describe each feature of the machine in terms of logical formulas such as appear in the boolean satisfiability problem. This construction establishes a correspondence between every problem in  $NP$  (which can be expressed as a program on the nondeterministic Turing machine) and some instance of satisfiability (the translation of that program into a logical formula). Now, the solution to the satisfiability problem essentially corresponds to a simulation of the machine running the given program on the given input, so it produces a solution to an instance of the given problem. Further details of this proof are well beyond the scope of this book. Fortunately, only one such proof is really necessary: it is much easier to use reduction to prove  $NP$ -completeness.

The Cook-Levin theorem, in conjunction with the thousands and thousands of poly-time reductions from  $NP$ -complete problems that have followed it, leaves us with two possible universes: either  $P = NP$  and no intractable search problems exist (all search problems can be solved in polynomial time); or  $P \neq NP$ , there do exist intractable search problems (some search problems cannot be solved in polynomial time).  $NP$ -complete



problems arise frequently in important natural practical applications, so there has been strong motivation to find good algorithms to solve them. The fact that no good algorithm has been found for any of these problems is surely strong evidence that  $P \neq NP$ , and most researchers certainly believe this to be the case. On the other hand, the fact that no one has been able to prove that any of these problems do not belong to  $P$  could be construed to comprise a similar body of circumstantial evidence on the other side. Whether or not  $P = NP$ , the practical fact is that the best known algorithm for any of the **NP**-complete problems takes exponential time in the worst case.

**Classifying problems.** Most practical search problems are known to be either in **P** or **NP**-complete. To prove that a search problem is in **P**, we need to exhibit a polynomial-time algorithm for solving it, perhaps by reducing it to a problem known to be in **P**. To prove that a problem in **NP** is **NP**-complete, we need to show that some known **NP**-complete problem is poly-time reducible to it: that is, that a polynomial-time algorithm for the new problem could be used to solve the **NP**-complete problem, and then could, in turn, be used to solve all problems in **NP**. Thousands and thousands of problems have been shown to be **NP**-complete in this way, as we did for integer linear programming in PROPOSITION L. The list on page 920, which includes several of the problems addressed by Karp, is representative, but contains only a tiny fraction of the known **NP**-complete problems. Classifying problems as being easy to solve (in **P**) or hard to solve (**NP**-complete) can be:

- *Straightforward.* For example, the venerable Gaussian elimination algorithm proves that linear equation satisfiability is in **P**.
- *Tricky but not difficult.* For example, developing a proof like the proof of PROPOSITION L takes some experience and practice, but it is easy to understand.
- *Extremely challenging.* For example, linear programming was long unclassified, but Khachian's ellipsoid algorithm proves that linear programming is in **P**.
- *Open.* For example, *graph isomorphism* (given two graphs, find a way to rename the vertices of one to make it identical to the other) and *factor* (given an integer, find a nontrivial factor) are still unclassified.

This is a rich and active area of current research, still involving thousands of research papers per year. As indicated by the last few entries on the list on page 920, all areas of scientific inquiry are affected. Recall that our definition of **NP** encompasses the problems that scientists, engineers, and applications programmers *aspire to solve* feasibly—all such problems certainly need to be classified!

**Boolean satisfiability.** Given a set of  $M$  equations involving  $N$  boolean variables, find an assignment of values to the variables that satisfies all of the equations, or report that none exists.

**Integer linear programming.** Given a set of  $M$  linear inequalities involving  $N$  integer variables, find an assignment of values to the variables that satisfies all of the inequalities, or report that none exists.

**Load balancing.** Given a set of jobs of specified duration to be completed and a time bound  $T$ , how can we schedule the jobs on two identical processors so as to complete them all by time  $T$ ?

**Vertex cover.** Given a graph and a integer  $C$ , find a set of  $C$  vertices such that each edge of the graph is incident to at least one vertex of the set.

**Hamiltonian path.** Given a graph, find a simple path that visits each vertex exactly once, or report that none exists.

**Protein folding.** Given energy level  $M$ , find a folded three-dimensional conformation of a protein having potential energy less than  $M$ .

**Ising model.** Given an Ising model on a lattice of dimension three and an energy threshhold  $E$ , is there a subgraph with free energy less than  $E$ ?

**Risk portfolio of a given return.** Given an investment portfolio with a given total cost, a given return, risk values assigned to each investment, and a threshold  $M$ , find a way to allocate the investments such that the risk is less than  $M$ .

### Some famous NP-complete problems

**Coping with NP-completeness.** Some sort of solution to this vast panoply of problems must be found in practice, so there is intense interest in finding ways to address them. It is impossible to do justice to this vast field of study in one paragraph, but we can briefly describe various approaches that have been tried. One approach is to change the problem and find an “approximation” algorithm that finds not the best solution but a solution guaranteed to be close to the best. For example, it is easy to find a solution to the Euclidean traveling salesperson problem that is within a factor of 2 of the optimal. Unfortunately, this approach is often not sufficient to ward off **NP**-completeness, when seeking improved approximations. Another approach is to develop an algorithm that solves efficiently virtually all of the instances that do arise in practice, even though there exist worst-case inputs for which finding a solution is infeasible. The most famous example of this approach are the integer linear programming solvers, which have been workhorses for many decades in solving huge optimization problems in countless industrial applications. Even though they could require exponential time, the inputs that arise in practice evidently are not worst-case inputs. A third approach is to work with “efficient” exponential algorithms, using a technique known as *backtracking* to avoid having to check all possible solutions. Finally, there is quite a large gap between polynomial and exponential time that is not addressed by the theory. What about an algorithm that runs in time proportional to  $N^{\log N}$  or  $2^{\sqrt{N}}$ ?

ALL THE APPLICATIONS AREAS we have studied in this book are touched by **NP**-completeness: **NP**-complete problems arise in elementary programming, in sorting and searching, in graph processing, in string processing, in scientific computing, in systems programming, in operations research, and in any conceivable area where computing plays a role. The most important practical contribution of the theory of **NP**-completeness is that it provides a mechanism to discover whether a new problem from any of these diverse areas is “easy” or “hard.” If one can find an efficient algorithm to solve a new problem, then there is no difficulty. If not, a proof that the problem is **NP**-complete tells us that developing an efficient algorithm would be a stunning achievement (and suggests that a different approach should perhaps be tried). The scores of efficient algorithms that we have examined in this book are testimony that we have learned a great deal about efficient computational methods since Euclid, but the theory of **NP**-completeness shows that, indeed, we still have a great deal to learn.

## EXERCISES on collision simulation

**6.1** Complete the implementation of `Particle` as described in the text. There are three equations governing the elastic collision between a pair of hard discs: (a) conservation of linear momentum, (b) conservation of kinetic energy, and (c) upon collision, the normal force acts perpendicular to the surface at the collision point (assuming no friction or spin). See the booksite for more details.

**6.2** Develop a version of `CollisionSystem`, `Particle`, and `Event` that handles multi-particle collisions. Such collisions are important when simulating the break in a game of billiards. (This is a difficult exercise!)

**6.3** Develop a version of `CollisionSystem`, `Particle`, and `Event` that works in three dimensions.

**6.4** Explore the idea of improving the performance of `simulate()` in `CollisionSystem` by dividing the region into rectangular cells and adding a new event type so that you only need to predict collisions with particles in one of nine adjacent cells in any time quantum. This approach reduces the number of predictions to calculate at the cost of monitoring the movement of particles from cell to cell.

**6.5** Introduce the concept of *entropy* to `CollisionSystem` and use it to confirm classical results.

**6.6** *Brownian motion.* In 1827, the botanist Robert Brown observed the motion of wildflower pollen grains immersed in water using a microscope. He observed that the pollen grains were in a random motion, following what would become known as Brownian motion. This phenomenon was discussed, but no convincing explanation was provided until Einstein provided a mathematical one in 1905. Einstein's explanation: the motion of the pollen grain particles was caused by millions of tiny molecules colliding with the larger particles. Run a simulation that illustrates this phenomenon.

**6.7** *Temperature.* Add a method `temperature()` to `Particle` that returns the product of its mass and the square of the magnitude of its velocity divided by  $dk_B$  where  $d=2$  is the dimension and  $k_B = 1.3806488 \times 10^{-23}$  is Boltzmann's constant. The temperature of the system is the average value of these quantities. Then add a method `temperature()` to `CollisionSystem` and write a driver that plots the temperature periodically, to check that it is constant.

**6.8 Maxwell-Boltzmann.** The distribution of velocity of particles in the hard disc model obeys the *Maxwell-Boltzmann distribution* (assuming that the system has thermalized and particles are sufficiently heavy that we can discount quantum-mechanical effects), which is known as the Rayleigh distribution in two dimensions. The distribution shape depends on temperature. Write a driver that computes a histogram of the particle velocities and test it for various temperatures.

**6.9 Arbitrary shape.** Molecules travel very quickly (faster than a speeding jet) but diffuse slowly because they collide with other molecules, thereby changing their direction. Extend the model to have a boundary shape where two vessels are connected by a pipe containing two different types of particles. Run a simulation and measure the fraction of particles of each type in each vessel as a function of time.

**6.10 Rewind.** After running a simulation, negate all velocities and then run the system backward. It should return to its original state! Measure roundoff error by measuring the difference between the final and original states of the system.

**6.11 Pressure.** Add a method `pressure()` to `Particle` that measures pressure by accumulating the number and magnitude of collisions against walls. The pressure of the system is the sum of these quantities. Then add a method `pressure()` to `CollisionSystem` and write a client that validates the equation  $pv = nRT$ .

**6.12 Index priority queue implementation.** Develop a version of `CollisionSystem` that uses an index priority queue to guarantee that the size of the priority queue is at most linear in the number of particles (instead of quadratic or worse).

**6.13 Priority queue performance.** Instrument the priority queue and test `Pressure` at various temperatures to identify the computational bottleneck. If warranted, try switching to a different priority-queue implementation for better performance at high temperatures.

## EXERCISES on B-Trees

**6.14** Suppose that, in a three-level tree, we can afford to keep  $a$  links in internal memory, between  $b$  and  $2b$  links in pages representing internal nodes, and between  $c$  and  $2c$  items in pages representing external nodes. What is the maximum number of items that we can hold in such a tree, as a function of  $a$ ,  $b$ , and  $c$ ?

**6.15** Develop an implementation of `Page` that represents each B-tree node as a `BinarySearchST` object.

**6.16** Extend `BTreeSET` to develop a `BTreeST` implementation that associates keys with values and supports our full ordered symbol table API that includes `min()`, `max()`, `floor()`, `ceiling()`, `deleteMin()`, `deleteMax()`, `select()`, `rank()`, and the two-argument versions of `size()` and `get()`.

**6.17** Write a program that uses `StdDraw` to visualize B-trees as they grow, as in the text.

**6.18** Estimate the average number of probes per search in a B-tree for  $S$  random searches, in a typical cache system, where the  $T$  most-recently-accessed pages are kept in memory (and therefore add 0 to the probe count). Assume that  $S$  is much larger than  $T$ .

**6.19** *Web search.* Develop an implementation of `Page` that represents B-tree nodes as text files on web pages, for the purposes of indexing (building a concordance for) the web. Use a file of search terms. Take web pages to be indexed from standard input. To keep control, take a command-line parameter  $m$ , and set an upper limit of  $10^m$  internal nodes (check with your system administrator before running for large  $m$ ). Use an  $m$ -digit number to name your internal nodes. For example, when  $m$  is 4, your nodes names might be `BTreeNode0000`, `BTreeNode0001`, `BTreeNode0002`, and so forth. Keep pairs of strings on pages. Add a `close()` operation to the API, to sort and write. To test your implementation, look for yourself and your friends on your university's website.

**6.20**  *$B^*$  trees.* Consider the sibling split (or  $B^*$ -tree) heuristic for B-trees: When it comes time to split a node because it contains  $M$  entries, we combine the node with its sibling. If the sibling has  $k$  entries with  $k < M-1$ , we reallocate the items giving the sibling and the full node each about  $(M+k)/2$  entries. Otherwise, we create a new node and give each of the three nodes about  $2M/3$  entries. Also, we allow the root to grow to hold about  $4M/3$  items, splitting it and creating a new root node with two entries when it reaches that bound. State bounds on the number of probes used for a search or an insertion in a  $B^*$ -tree of order  $M$  with  $N$  items. Compare your bounds with the

corresponding bounds for B-trees (see PROPOSITION B). Develop an *insert* implementation for B\*-trees.

**6.21** Write a program to compute the average number of external pages for a B-tree of order  $M$  built from  $N$  random insertions into an initially empty tree. Run your program for reasonable values of  $M$  and  $N$ .

**6.22** If your system supports virtual memory, design and conduct experiments to compare the performance of B-trees with that of binary search, for random searches in a huge symbol table.

**6.23** For your internal-memory implementation of Page in EXERCISE 6.15, run experiments to determine the value of  $M$  that leads to the fastest search times for a B-tree implementation supporting random search operations in a huge symbol table. Restrict your attention to values of  $M$  that are multiples of 100.

**6.24** Run experiments to compare search times for internal B-trees (using the value of  $M$  determined in the previous exercise), linear probing hashing, and red-black trees for random search operations in a huge symbol table.

## EXERCISES on suffix arrays

**6.25** Give, in the style of the figure on page 878, the suffixes, sorted suffixes, `index()` and `lcp()` tables for the following strings:

- a. abacadaba
- b. mississippi
- c. abcdefghij
- d.aaaaaaaaaa

**6.26** Identify the problem with the following code fragment to suffix sort a string `s`:

```
int N = s.length();
String[] suffixes = new String[N];
for (int i = 0; i < N; i++)
    suffixes[i] = s.substring(i, N);
Quick3way.sort(suffixes);
```

*Answer:* There is no problem if the `substring()` method takes constant time and space (as is typical in Java 6 implementations). If the `substring()` method takes linear time and space (as is typical in Java 7 implementations), then the loop takes quadratic time and space.

**6.27** Some applications require a sort of *cyclic rotations* of a text: for  $i$  from 0 to  $N - 1$ , the  $i$ th cyclic rotation of a text of length  $N$  is the last  $N - i$  characters followed by the first  $i$  characters. Develop a `CircularSuffixArray` data type that has the same API as `SuffixArray` (page 879) but using the  $N$  cyclic rotations of the text instead of the  $N$  suffixes.

**6.28** Complete the implementation of `SuffixArray` (ALGORITHM 6.2).

**6.29** Under the assumptions described in SECTION 1.4, give the memory usage of a `SuffixArray` object with a string of length  $N$ .

**6.30** Modify the implementation to avoid using the helper class `Suffix`; instead make your instance variables a `char` array (to store the text characters) and an `int` array (to store the indices of the first characters in each of the sorted suffixes) and use a customized version of 3-way string quicksort to sort the suffixes. Compare the running time and memory usage of this implementation versus the one that uses the `Suffix` class.

**6.31 Longest common substring.** Write a `SuffixArray` client `LCS` that take two file-names as command-line arguments, reads the two text files, and finds the longest substring that appears in both in linear time. (In 1970, D. Knuth conjectured that this task was impossible.) *Hint:* Create a suffix array for  $s\#t$  where  $s$  and  $t$  are the two text strings and  $\#$  is a character that does not appear in either.

**6.32 Burrows-Wheeler transform.** The *Burrows-Wheeler transform* (BWT) is a transformation that is used in data compression algorithms, including `bzip2` and in high-throughput sequencing in genomics. Write a `CircularSuffixArray` client that computes the BWT in linear time, as follows: Given a string of length  $N$  (terminated by a special end-of-file character  $\$$  that is smaller than any other character), consider the  $N$ -by- $N$  matrix in which each row contains a different cyclic rotation of the original text string. Sort the rows lexicographically. The Burrows-Wheeler transform is the rightmost column in the sorted matrix. For example, the BWT of `mississippi$` is `ipssm$pissii`. The *Burrows-Wheeler inverse transform* (BWI) inverts the BWT. For example, the BWI of `ipssm$pissii` is `mississippi$`. Also write a client that, given the BWT of a text string, computes the BWI in linear time.

**6.33 Circular string linearization.** Write a `CircularSuffixArray` client that, given a string, finds the cyclic rotation that is the smallest lexicographically in linear time. This problem arises in chemical databases for circular molecules, where each molecule is represented as a circular string, and a canonical representation (smallest cyclic rotation) is used to support search with any rotation as key. (See EXERCISE 6.27.)

**6.34 Longest repeated substrings.** Write a `SuffixArray` client that, given a string and an integer  $k$ , find the longest substring that is repeated  $k$  or more times. Write a `SuffixArray` client that, given a string and an integer  $L$ , finds all repeated substrings of length  $L$  or more.

**6.35  $k$ -gram frequency counts.** Develop and implement an ADT for preprocessing a string to support efficiently answering queries of the form *How many times does a given  $k$ -gram appear?* Each query should take time proportional to  $k \log N$  in the worst case, where  $N$  is the length of the string.

## EXERCISES on maxflow

**6.36** If capacities are positive integers less than  $M$ , what is the maximum possible flow value for any  $st$ -network with  $V$  vertices and  $E$  edges? Give two answers, depending on whether or not parallel edges are allowed.

**6.37** Give an algorithm to solve the maxflow problem for the case that the network forms a tree if the sink is removed.

**6.38** *True or false.* If true provide a short proof, if false give a counterexample:

- a. In any max flow, there is no directed cycle on which every edge carries positive flow
- b. There exists a max flow for which there is no directed cycle on which every edge carries positive flow
- c. If all edge capacities are distinct, the max flow is unique
- d. If all edge capacities are increased by an additive constant, the min cut remains unchanged
- e. If all edge capacities are multiplied by a positive integer, the min cut remains unchanged

**6.39** Complete the proof of PROPOSITION G: Show that each time an edge is a critical edge, the length of the augmenting path through it must increase by 2.

**6.40** Find a large network online that you can use as a vehicle for testing flow algorithms on realistic data. Possibilities include transportation networks (road, rail, or air), communications networks (telephone or computer connections), or distribution networks. If capacities are not available, devise a reasonable model to add them. Write a program that uses the interface to implement flow networks from your data. If warranted, develop additional private methods to clean up the data.

**6.41** Write a random-network generator for sparse networks with integer capacities between 0 and  $2^{20}$ . Use a separate class for capacities and develop two implementations: one that generates uniformly distributed capacities and another that generates capacities according to a Gaussian distribution. Implement client programs that generate random networks for both capacity distributions with a well-chosen set of values of  $V$  and  $E$  so that you can use them to run empirical tests.

**6.42** Write a program that generates  $V$  random points in the plane, then builds a flow network with edges (in both directions) connecting all pairs of points within a given distance  $d$  of each other, setting each edge's capacity using one of the random models described in the previous exercise.

**6.43** *Basic reductions.* Develop `FordFulkerson` clients for finding a maxflow in each of the following types of flow networks:

- Undirected
- No constraint on the number of sources or sinks or on either edges pointing to the sources or pointing from the sinks
- Lower bounds on capacities
- Capacity constraints on vertices

**6.44** *Product distribution.* Suppose that a flow represents products to be transferred by trucks between cities, with the flow on edge  $u-v$  representing the amount to be taken from city  $u$  to city  $v$  in a given day. Write a client that prints out daily orders for truckers, telling them how much and where to pick up and how much and where to drop off. Assume that there are no limits on the supply of truckers and that nothing leaves a given distribution point until everything has arrived.

**6.45** *Job placement.* Develop a `FordFulkerson` client that solves the job-placement problem, using the reduction in PROPOSITION J. Use a symbol table to convert symbolic names into integers for use in the flow network.

**6.46** Construct a family of bipartite matching problems where the average length of the augmenting paths used by any augmenting-path algorithm to solve the corresponding maxflow problem is proportional to  $E$ .

**6.47** *st-connectivity.* Develop a `FordFulkerson` client that, given an undirected graph  $G$  and vertices  $s$  and  $t$ , finds the minimum number of edges in  $G$  whose removal will disconnect  $t$  from  $s$ .

**6.48** *Disjoint paths.* Develop a `FordFulkerson` client that, given an undirected graph  $G$  and vertices  $s$  and  $t$ , finds the maximum number of edge-disjoint paths from  $s$  to  $t$ .

## EXERCISES on reductions and intractability

**6.49** Find a nontrivial factor of 37703491.

**6.50** Prove that the shortest-paths problem reduces to linear programming.

**6.51** Could there be an algorithm that solves an **NP**-complete problem in an average time of  $N^{\log N}$ , if  $P \neq NP$ ? Explain your answer.

**6.52** Suppose that someone discovers an algorithm that is guaranteed to solve the boolean satisfiability problem in time proportional to  $1.1^N$ . Does this imply that we can solve other **NP**-complete problems in time proportional to  $1.1^N$ ?

**6.53** What would be the significance of a program that could solve the integer linear programming problem in time proportional to  $1.1^N$ ?

**6.54** Give a poly-time reduction from vertex cover to 0-1 integer linear inequality satisfiability.

**6.55** Prove that the problem of finding a Hamiltonian path in a *directed* graph is **NP**-complete, using the **NP**-completeness of the Hamiltonian-path problem for undirected graphs.

**6.56** Suppose that two problems are known to be **NP**-complete. Does this imply that there is a poly-time reduction from one to the other?

**6.57** Suppose that  $X$  is **NP**-complete,  $X$  poly-time reduces to  $Y$ , and  $Y$  poly-time reduces to  $X$ . Is  $Y$  necessarily **NP**-complete?

**6.58** Suppose that we have an algorithm to solve the decision version of boolean satisfiability, which indicates that there exists an assignment of truth values to the variables that satisfies the boolean expression. Show how to find the assignment.

**6.59** Suppose that we have an algorithm to solve the decision version of the vertex cover problem, which indicates that there exists a vertex cover of a given size. Show how to solve the optimization version of finding the vertex cover of minimum cardinality.

**6.60** Explain why the optimization version of the vertex cover problem is not necessarily a search problem.

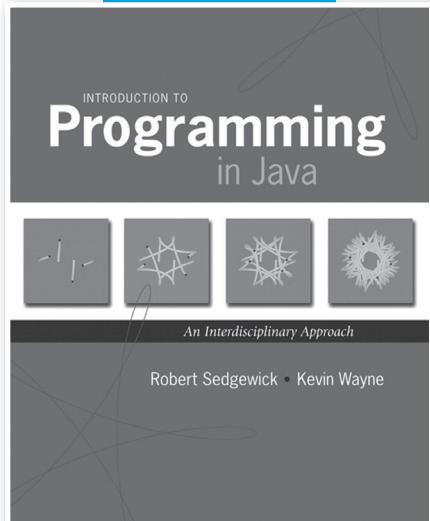
**6.61** Suppose that  $X$  and  $Y$  are two search problems and that  $X$  poly-time reduces to  $Y$ . Which of the following can we infer?

- a. If  $Y$  is  $\text{NP}$ -complete then so is  $X$ .
- b. If  $X$  is  $\text{NP}$ -complete then so is  $Y$ .
- c. If  $X$  is in  $P$ , then  $Y$  is in  $P$ .
- d. If  $Y$  is in  $P$ , then  $X$  is in  $P$ .

**6.62** Suppose that  $P \neq NP$ . Which of the following can we infer?

- a. If  $X$  is  $\text{NP}$ -complete, then  $X$  cannot be solved in polynomial time.
- b. If  $X$  is in  $\text{NP}$ , then  $X$  cannot be solved in polynomial time.
- c. If  $X$  is in  $\text{NP}$  but not  $\text{NP}$ -complete, then  $X$  can be solved in polynomial time.
- d. If  $X$  is in  $P$ , then  $X$  is not  $\text{NP}$ -complete.

*This page intentionally left blank*



## *Introduction to Programming in Java: An Interdisciplinary Approach*

Robert Sedgewick/Kevin Wayne  
©2008 • 736 pp • ISBN: 0-321-49805-4

*Introduction to Programming in Java* takes an interdisciplinary approach to teaching programming with the Java programming language.

### Features

This book thoroughly covers the field and is ideal for introductory programming courses. It can also be used for courses that integrate programming with mathematics, science, or engineering.

Students learn basic computer science concepts in the context of interesting applications in science, engineering, and commercial computing, leveraging familiar science and math while preparing students to use computers effectively in later courses. This serves to demonstrate that computation is not merely a tool, but an integral part of the modern world that pervades scientific inquiry and commercial development.

The book takes an “objects in the middle” approach where students learn basic control structures and functions, then how to use, create, and design classes.

A full programming model includes standard libraries for input, graphics, sound, and image processing that students can immediately put to use.

An integrated Companion Website features extensive Java coding examples, additional exercises, and Web links.

Instructors, contact your Pearson representative to receive an exam copy, or email [PearsonEd.CS@Pearson.com](mailto:PearsonEd.CS@Pearson.com).



# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

# informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE



**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

▼Addison-Wesley

Cisco Press

EXAM/CRAM

IBM  
Press..

QUE®

PRENTICE  
HALL

SAMS

| Safari®  
Books Online

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.  
Visit [informit.com/newsletters](http://informit.com/newsletters).
- Access FREE podcasts from experts at [informit.com/podcasts](http://informit.com/podcasts).
- Read the latest author articles and sample chapters at [informit.com/articles](http://informit.com/articles).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](http://safari.informit.com).
- Get tips from expert blogs at [informit.com/blogs](http://informit.com/blogs).

Visit [informit.com/learn](http://informit.com/learn) to discover all the ways you can access the hottest technology content.

### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](http://informit.com/socialconnect).



# Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos



**FREE TRIAL—GET STARTED TODAY!**  
[www.informit.com/safaritrial](http://www.informit.com/safaritrial)

➤ **Find trusted answers, fast**  
Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.

➤ **Master the latest tools and techniques**  
In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

## WAIT, THERE'S MORE!

➤ **Keep your competitive edge**  
With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.

➤ **Stay current with emerging technologies**  
Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.



AdobePress



Cisco Press



Microsoft  
Press



O'REILLY



que®



SAMS

Sas Publishing



Wharton School  
Publishing

